Бесплатная электронная книга

учусь common-lisp

Free unaffiliated eBook created from
Stack Overflow contributors.

#common-lisp

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: common-lisp

It is an unofficial and free common-lisp ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official common-lisp.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с общими

замечания

Это простая функция hello world в Common Lisp. Примеры напечатают текст `Hello, World!` (без кавычек, за которым следует новая строка) до стандартного вывода.

Common Lisp - это язык программирования, который в основном используется интерактивно с использованием интерфейса, известного как REPL. REPL (Read Eval Print Loop) позволяет вводить код, оценивать (запускать) и сразу видеть результаты. Запрос для REPL (в какой момент один из них вводит код, который должен быть запущен) обозначается `CL-USER>` . Иногда перед `>` должен появляться нечто иное, кроме `CL-USER` , но это все еще REPL.

После запроса появляется код, обычно либо одно слово (то есть имя переменной), либо форма (список слов / форм, заключенных между `(` и `)`) (т. Е. Вызов функции или объявление и т. Д.). На следующей строке будет любой вывод, который программа печатает (или ничего, если программа ничего не печатает), а затем значения, возвращаемые путем вычисления выражения. Обычно выражение возвращает одно значение, но если оно возвращает несколько значений, они появляются один раз в строке.

Версии

Версия	Дата выхода
Общий Лисп	1984-01-01
ANSI Common Lisp	1994-01-01

Examples

Привет, мир

Ниже следует отрывок из сеанса REPL с Common Lisp, в котором «Hello, World!» функция определена и выполнена. См. Замечания в нижней части этой страницы для более подробного описания REPL.

```
CL-USER> (defun hello ()
           (format t "Hello, World!~%"))
HELLO
CL-USER> (hello)
Hello, World!
NIL
```

```
CL-USER>
```

Это определяет «функцию» нулевых аргументов с именем `hello` , которая будет писать строку `"Hello, World!"` за которым следует новая строка для стандартного вывода и возвращаем `NIL` .

Чтобы определить функцию, напишем

```
(defun name (parameters...)
  code...)
```

В этом случае функция называется `hello` , не принимает никаких параметров, а код, который он запускает, - это один вызов функции. Возвращаемое значение из функции lisp - это последний бит кода в функции для запуска, так что `hello` возвращается `(format t "Hello, World!~%")` возвращается `(format t "Hello, World!~%")` .

В lisp для вызова функции write `(function-name arguments...)` где `function-name` - это имя функции, а `arguments...` - это список аргументов (разделенных пробелами) для вызова. Существуют некоторые особые случаи, которые выглядят как вызовы функций, но не являются, например, в приведенном выше коде не существует функции `defun` , вызываемой, она обрабатывается специально и определяет функцию вместо этого.

Во втором приглашении REPL, после того как мы определили функцию `hello` , мы вызываем его без параметров, записывая `(hello)` . Это, в свою очередь, вызовет функцию `format` с параметрами `t` и `"Hello, World!~%"` . Функция `format` производит форматированный вывод на основе аргументов, которые он задает (немного похож на расширенную версию `printf` в C). Первый аргумент указывает, куда выводить, с `t` значением standard-output. Второй аргумент говорит, что печатать (и как интерпретировать любые дополнительные параметры). Директива (специальный код во втором аргументе) `~%` сообщает формату печати новой строки (т.е. в UNIX она может писать `\n` и на windows `\r\n`). Формат обычно возвращает `NIL` (немного как `NULL` на других языках).

После второго запроса мы видим, что `Hello, World` был напечатан, а на следующей строке возвращаемое значение было `NIL` .

Привет, Имя

Это немного более продвинутый пример, который показывает еще несколько функций общего lisp. Начнем с простого `Hello, World!` функции и продемонстрировать некоторые интерактивные разработки на REPL. Обратите внимание, что любой текст из точки с запятой `;` , остальная часть строки - комментарий.

```
CL-USER> (defun hello ()
       (format t "Hello, World!~%")) ;We start as before
HELLO
CL-USER> (hello)
```

```
Hello, World!
NIL
CL-USER> (defun hello-name (name) ;A function to say hello to anyone
       (format t "Hello, ~a~%" name)) ;~a prints the next argument to format
HELLO-NAME
CL-USER> (hello-name "Jack")
Hello, Jack
NIL
CL-USER> (hello-name "jack") ;doesn't capitalise names
Hello, jack
NIL
CL-USER> (defun hello-name (name) ;format has a feature to convert to title case
       (format t "Hello, ~:(~a~)~%" name)) ;anything between ~:( and ~) gets it
WARNING: redefining COMMON-LISP-USER::HELLO-NAME in DEFUN
HELLO-NAME
CL-USER> (hello-name "jack")
Hello, Jack
NIL
CL-USER> (defun hello-name (name)
       (format t "Hello, ~:(~a~)!~%" name))
WARNING: redefining COMMON-LISP-USER::HELLO-NAME in DEFUN
HELLO-NAME
CL-USER> (hello-name "jack") ;now this works
Hello, Jack!
NIL
CL-USER> (defun hello (&optional (name "world")) ;we can take an optional argument
       (hello-name name)) ;name defaults to "world"
WARNING: redefining COMMON-LISP-USER::HELLO in DEFUN
HELLO
CL-USER> (hello)
Hello, World!
NIL
CL-USER> (hello "jack")
Hello, Jack!
NIL
CL-USER> (hello "john doe") ;note that this capitalises both names
Hello, John Doe!
NIL
CL-USER> (defun hello-person (name &key (number))
       (format t "Hello, ~a ~r" name number)) ;~r prints a number in English
HELLO-PERSON
CL-USER> (hello-person "Louis" :number 16) ;this doesn't quite work
Hello, Louis sixteen
NIL
CL-USER> (defun hello-person (name &key (number))
       (format t "Hello, ~:(~a ~:r~)!" name number)) ;~:r prints an ordinal
WARNING: redefining COMMON-LISP-USER::HELLO-PERSON in DEFUN
HELLO-PERSON
CL-USER> (hello-person "Louis" :number 16)
Hello, Louis Sixteenth!
NIL
CL-USER> (defun hello-person (name &key (number))
       (format t "Hello, ~:(~a ~@r~)!" name number)) ;~@r prints Roman numerals
WARNING: redefining COMMON-LISP-USER::HELLO-PERSON in DEFUN
HELLO-PERSON
CL-USER> (hello-person "Louis" :number 16)
Hello, Louis Xvi!
NIL
CL-USER> (defun hello-person (name &key (number)) ;capitalisation was wrong
       (format t "Hello, ~:(~a~) ~:@r!" name number))
WARNING: redefining COMMON-LISP-USER::HELLO-PERSON in DEFUN
```

```
HELLO-PERSON
CL-USER> (hello-person "Louis" :number 16) ;thats better
Hello, Louis XVI!
NIL
CL-USER> (hello-person "Louis") ;we get an error because NIL is not a number
Hello, Louis ; Evaluation aborted on #<SB-FORMAT:FORMAT-ERROR {1006641AB3}>.
CL-USER> (defun say-person (name &key (number 1 number-p)
                                      (title nil) (roman-number t))
       (let ((number (if number-p
                 (typecase number
                   (integer
                (format nil (if roman-number " ~:@r" " ~:(~:r~)") number))
                   (otherwise
                (format nil " ~:(~a~)" number)))
                 "")) ; here we define a variable called number
         (title (if title
                (format nil "~:(~a~) " title)
                ""))) ; and here one called title
         (format nil "~a~:(~a~)~a" title name number))) ;we use them here

SAY-PERSON
CL-USER> (say-person "John") ;some examples
"John"
CL-USER> (say-person "john doe")
"John Doe"
CL-USER> (say-person "john doe" :number "JR")
"John Doe Jr"
CL-USER> (say-person "john doe" :number "Junior")
"John Doe Junior"
CL-USER> (say-person "john doe" :number 1)
"John Doe I"
CL-USER> (say-person "john doe" :number 1 :roman-number nil) ;this is wrong
"John Doe First"
CL-USER> (defun say-person (name &key (number 1 number-p)
                                      (title nil) (roman-number t))
       (let ((number (if number-p
                 (typecase number
                   (integer
                (format nil (if roman-number " ~:@r" " the ~:(~:r~)") number))
                   (otherwise
                (format nil " ~:(~a~)" number)))
                 ""))
         (title (if title
                (format nil "~:(~a~) " title)
                "")))
         (format nil "~a~:(~a~)~a" title name number)))
WARNING: redefining COMMON-LISP-USER::SAY-PERSON in DEFUN
SAY-PERSON
CL-USER> (say-person "john doe" :number 1 :roman-number nil) ;thats better
"John Doe the First"
CL-USER> (say-person "louis" :title "king" :number 16 :roman-number nil)
"King Louis the Sixteenth"
CL-USER> (say-person "louis" :title "king" :number 16 :roman-number t)
"King Louis XVI"
CL-USER> (defun hello (&optional (name "World") &rest arguments) ;now we will just
       (apply #'hello-name name arguments)) ;pass all arguments to hello-name
WARNING: redefining COMMON-LISP-USER::HELLO in DEFUN
HELLO
CL-USER> (defun hello-name (name &rest arguments) ;which will now just use
       (format t "Hello, ~a!" (apply #'say-person name arguments))) ;say-person
WARNING: redefining COMMON-LISP-USER::HELLO-NAME in DEFUN
```

```
HELLO-NAME
CL-USER> (hello "louis" :title "king" :number 16) ;this works now
Hello, King Louis XVI!
NIL
CL-USER>
```

Это освещает некоторые дополнительные функции функции `format` Common Lisp, а также некоторые функции, такие как необязательные параметры и аргументы ключевых слов (например, `:number`). Это также дает пример интерактивной разработки в REPL в общем lisp.

Простая программа Hello World в REPL

Common Lisp REPL - это интерактивная среда. Каждая форма, написанная после запроса, оценивается, и ее значение впоследствии печатается в результате оценки. Итак, самая простая программа Hello, World! В Common Lisp:

```
CL-USER> "Hello, World!"
"Hello, World!"
CL-USER>
```

Что происходит здесь, так это то, что строковый costant задается во вводе REPL, он вычисляется и результат печатается. Из этого примера видно, что строки, такие как числа, специальные символы, такие как `NIL` и `T` и несколько других литералов, являются *самооценками* : они сами оценивают.

Основные выражения

Попробуем некоторое базовое выражение в REPL:

```
CL-USER> (+ 1 2 3)
6
CL-USER> (- 3 1 1)
1
CL-USER> (- 3)
-3
CL-USER> (+ 5.3 (- 3 2) (* 2 2))
10.3
CL-USER> (concatenate 'string "Hello, " "World!")
"Hello, World!"
CL-USER>
```

Основной строительный блок Common Lisp - это *форма* . В этих примерах мы имеем *формы функций* , то есть выражения, написанные как список, в которых первый элемент является оператором (или функцией), а остальные элементы являются операндами (это называется «Префиксная нотация» или «Польская нотация» «). Написание форм в REPL вызывает их оценку. В примерах вы можете увидеть простые выражения, аргументы которых являются постоянными числами, строками и символами (в случае `'string` , которая является именем типа). Вы также можете видеть, что арифметические операторы могут принимать любое

количество аргументов.

Важно отметить, что круглые скобки являются неотъемлемой частью синтаксиса и не могут использоваться свободно, как в других языках программирования. Например, следующая ошибка:

```
(+ 5 ((+ 2 4)))
> Error: Car of ((+ 2 4)) is not a function name or lambda-expression. ...
```

В общих формах Лиспа также могут быть данные, символы, макроформы, специальные формы и формы лямбда. Они могут быть записаны для оценки, возврата нуля, одного или нескольких значений или могут быть заданы во вводе макроса, которые преобразуют их в другие формы.

Сумма списка целых чисел

```
(defun sum-list-integers (list)
    (reduce '+ list))

; 10
(sum-list-integers '(1 2 3 4))

; 55
(sum-list-integers '(1 2 3 4 5 6 7 8 9 10))
```

Лямбда-выражения и анонимные функции

Анонимная функция может быть определена без имени через выражение Lambda . Для определения этих типов функций вместо ключевого слова `defun` используется ключевое слово `lambda` . Следующие строки эквивалентны и определяют анонимные функции, которые выводят сумму двух чисел:

```
(lambda (x y) (+ x y))
(function (lambda (x y) (+ x y)))
#'(lambda (x y) (+ x y))
```

Их полезность заметна при создании Лямбда-форм , то есть в форме, которая представляет собой список, в котором первым элементом является лямбда-выражение, а остальные элементы являются аргументами анонимной функции. Примеры (онлайн-исполнение):

```
(print ((lambda (x y) (+ x y)) 1 2)) ; >> 3

(print (mapcar (lambda (x y) (+ x y)) '(1 2 3) '(2 -5 0))) ; >> (3 -3 3)
```

Общие учебные ресурсы Lisp

Интернет-книги

Это книги, которые свободно доступны в Интернете.

- Практический общий Lisp от Peter Seibel - хорошее введение в CL для опытных программистов, которое с самого начала пытается подчеркнуть, что делает CL отличным от других языков.
- Common Lisp: Нежное введение в символические вычисления Дэвида С. Туретцкого - хорошее введение для людей, новых для программирования.
- Common Lisp: интерактивный подход Стюарта Шапиро был использован в качестве учебника курса, и примечания к курсу сопровождают книгу на веб-сайте.
- Common Lisp, Language by Guy L. Steele - это описание языка Common Lisp. Согласно CLiki, он устарел, но содержит более подробные описания макроса цикла и формата, чем это делает Common Lisp Hyperspec.
- На Lisp by Paul Graham - отличная книга для опытных мастеров Лисперса.
- Let Over Lambda от Doug Hoyte - это передовая книга о Lisp Macros. Несколько человек рекомендовали вам комфортно работать с Lisp перед тем, как прочитать эту книгу, и что начало происходит медленно.

Интернет-ссылки

- Common Lisp Hyperspec - *это* справочный документ для Common Lisp.
- Common Lisp Cookbook - это список полезных рецептов Lisp. Также содержит список других онлайн-источников информации CL.
- В Common Lisp Quick Reference есть листы для печати Lisp.
- Lispdoc.com ищет несколько источников информации о Lisp (Practical Common Lisp, Successful Lisp, On Lisp, HyperSpec) для документации.
- L1sp.org - это служба переадресации для документации.

Не в сети

Это книги, которые вам, вероятно, придется купить или получить в библиотеке.

- ANSI Common Lisp от Paul Graham .
- Общие рецепты Лиспа от Эдмунда Вейца .
- В парадигмах программирования искусственного интеллекта есть много интересных приложений Lisp, но это не является хорошей ссылкой для AI.

Интернет-сообщества

- CLiki имеет отличную начальную страницу . Отличный ресурс для всех вещей CL. Имеет обширный список книг Лиспа .
- Common Lisp subreddit имеет множество полезных ссылок и справочных документов на боковой панели.
- IRC: #lisp, #ccl, #sbcl и другие на Freenode .

- Common-Lisp.net предоставляет хостинг для многих общих проектов lisp и групп пользователей.

Библиотеки

- Quicklisp является библиотечным менеджером для Common Lisp и имеет длинный список поддерживаемых библиотек .
- QuickDocs размещает библиотечную документацию для многих библиотек CL.
- Awesome CL - это кураторский список библиотек, фреймворков и других блестящих материалов, отсортированных по категориям.

Предварительно упакованные среды Lisp

Это среды редактирования Lisp, которые легко установить и начать работу, потому что все, что вам нужно, предварительно упаковано и предварительно настроено.

- Portacle - это переносная и многоплатформенная среда Common Lisp. Он отправляет слегка настроенные Emacs со Slime, SBCL (популярная реализация Common Lisp), Quicklisp и Git. Никакой установки не требуется, так что это очень быстрый и простой способ добиться успеха.
- Lispbox - это среда IDE (Emacs + SLIME), Common Lisp (Clozure Common Lisp) и менеджер библиотек (Quicklisp), предварительно упакованные в виде архивов для Windows, Mac OSX и Linux. Потомок «Лиспа в коробке», рекомендованный в книге «Практические общие Лиспы».
- Не предварительно упакован, но SLIME превращает Emacs в общую среду Lisp IDE и содержит руководство пользователя, которое поможет вам начать работу. Требуется отдельная реализация Common Lisp.

Общие реализации Lisp

В этом разделе перечислены некоторые общие реализации CL и их руководства. Если не указано иное, это реализация программного обеспечения. См. Также список бесплатных программ Clic's Common Lisp и список коммерческих реализаций Common Lisp в Википедии .

- Allegro Common Lisp (ACL) и руководство пользователя . Коммерческий, но имеет бесплатное Express Edition и учебные видеоролики на Youtube .
- CLISP и руководства .
- Clozure Common Lisp (CCL) и руководство .
- Университет Carnegie Mellon Common Lisp (CMUCL) имеет справочную и другую полезную страницу информации .
- Embeddable Common Lisp (ECL) и руководство .
- LispWorks и руководство . Коммерческий, но имеет личную версию с некоторыми ограничениями .
- Steel Bank Common Lisp (SBCL) и руководство .

- Scieneer Common Lisp (SCL) и руководство - это коммерческая реализация Linux и Unix, но имеет неограниченную бесплатную версию для оценки и некоммерческого использования .

Прочитайте Начало работы с общими онлайн: https://riptutorial.com/ru/common-lisp/topic/534/начало-работы-с-общими

глава 2: ANSI Common Lisp, языковой стандарт и его документация

Examples

Общий Lisp HyperSpec

Common Lisp имеет стандарт, который был первоначально опубликован в 1994 году как стандарт ANSI.

Common Lisp HyperSpec , короткий CLHS, предоставленный LispWorks, является часто используемой HTML-документацией, которая получена из стандартного документа. HyperSpec также можно загружать и локально использовать .

Общие среды разработки Lisp обычно позволяют искать документацию HyperSpec для символов Lisp.

- Для GNU Emacs есть clhs.el.
- SLIME для GNU Emacs предоставляет версию hyperspec.el .

Смотрите также: cliki на CLHS

Объявления синтаксиса EBNF в документации

Стандарт ANSI CL использует расширенную синтаксическую нотацию EBNF. Документация, дублируемая в Stackoverflow, должна использовать одну и ту же синтаксическую нотацию, чтобы уменьшить путаницу.

Пример:

```
specialized-lambda-list::=
     ({var | (var parameter-specializer-name)}*
      [&optional {var | (var [initform [supplied-p-parameter] ])}*]
      [&rest var]
      [&key{var | ({var | (keywordvar)} [initform [supplied-p-parameter] ])}*
           [&allow-other-keys] ]
      [&aux {var | (var [initform] )}*] )
```

Обозначения:

- `[foo]` -> ноль или один `foo`
- `{foo}*` -> ноль или больше `foo`
- `foo | bar` -> `foo` или `bar`

Common Lisp the Language, 2-е издание, автор Guy L. Steele Jr.

Эта книга известна как CLtL2.

Это второе издание книги «Common Lisp the Language». Он был опубликован в 1990 году, прежде чем стандарт ANSI CL был окончательным. Первоначальное определение языка из первого издания (опубликованное в 1984 году) было рассмотрено и описано все изменения в процессе стандартизации до 1990 года, а также некоторые расширения (например, итерация SERIES).

Примечание: CLTL2 описывает версию Common Lisp, которая немного отличается от опубликованного стандарта с 1994 года. Таким образом, всегда используйте стандартный, а не CLtL2, в качестве ссылки.

CLtL2 все еще может быть полезен, поскольку он предоставляет информацию, не найденную в документе спецификации Common Lisp.

Существует версия HTML Common Lisp the Language, 2nd Edition .

CLiki - Предлагаемые изменения и разъяснения ANSI

В CLiki, Wiki для Common Lisp и *бесплатном* программном обеспечении Common Lisp, поддерживается список предлагаемых ANSI-исправлений и разъяснений .

Поскольку стандарт Common Lisp не изменился с 1994 года, пользователи обнаружили несколько проблем со спецификационным документом. Они задокументированы на странице CLiki.

Общая краткая справочная информация Lisp

Common Lisp Quick Reference - это документ, который можно распечатать и связать как буклет в различных макетах, чтобы иметь распечатанную краткую ссылку для Common Lisp.

Стандарт ANSI Common Lisp в формате Texinfo (особенно полезен для GNU Emacs)

GNU Emacs использует специальный формат для документации: *info* .

Стандарт Common Lisp был преобразован в формат Texinfo, который можно использовать для создания документации, доступной для просмотра с помощью устройства чтения *информации* в GNU Emacs.

См. Здесь: dpans2texi.el преобразует источники TeX проекта стандарта ANSI Common Lisp (dpANS) в формат Texinfo.

Другая версия была выполнена для GCL: gcl.info.tgz .

Прочитайте ANSI Common Lisp, языковой стандарт и его документация онлайн: https://riptutorial.com/ru/common-lisp/topic/2900/ansi-common-lisp--языковой-стандарт-и-его-документация

глава 3: ASDF - другой механизм определения системы

замечания

ASDF - другой механизм определения системы

ASDF - это инструмент для определения того, как системы программного обеспечения Common Lisp состоят из компонентов (подсистем и файлов) и как работать с этими компонентами в правильном порядке, чтобы их можно было компилировать, загружать, тестировать и т. Д.

Examples

Простая система ASDF с плоской структурой каталогов

Рассмотрим этот простой проект с плоской структурой каталогов:

```
example
|-- example.asd
|-- functions.lisp
|-- main.lisp
|-- packages.lisp
`-- tools.lisp
```

Файл `example.asd` - это просто еще один файл Lisp, содержащий немного больше, чем вызов функции ASDF. Предполагая, что ваш проект зависит от систем `drakma` и `clsql` , его содержимое может быть примерно таким:

```
(asdf:defsystem :example
    :description "a simple example project"
    :version "1.0"
    :author "TheAuthor"
    :depends-on (:clsql
                 :drakma)
    :components ((:file "packages")
                 (:file "tools" :depends-on ("packages"))
                 (:file "functions" :depends-on ("packages"))
                 (:file "main" :depends-on ("packages"
                                            "functions"))))
```

Когда вы загружаете этот файл Lisp, вы указываете ASDF о своей системе `:example` , но вы еще не загружаете систему. Это делается либо `(asdf:require-system :example)` либо `(ql:quickload :example)` .

И когда вы загружаете систему, ASDF будет:

1. Загрузите зависимости - в этом случае системы `clsql` и `drakma`
2. *Компилируйте и загрузите* компоненты вашей системы, то есть файлы Lisp, основанные на данных зависимостях
    1. `packages` сначала (без зависимостей)
    2. `functions` после `packages` (поскольку это зависит только от `packages`), но до `main` (что зависит от него)
    3. `main` после `functions` (поскольку это зависит от `packages` и `functions`)
    4. `tools` любое время после `packages`

Иметь ввиду:

- Введите зависимости по мере необходимости (например, для использования требуется определение макросов). Если вы этого не сделаете, ASDF будет ошибочно при загрузке вашей системы.
- Все перечисленные файлы заканчиваются на `.lisp` но этот постфикс должен быть `.lisp` в сценарии asdf
- Если ваша система названа так же, как и ее `.asd` файл, и вы перемещаете (или символизируете) ее папку в папку `quicklisp/local-projects/` , вы можете загрузить проект, используя `(ql:quickload "example")` .
- Библиотеки, зависящие от вашей системы, должны быть известны либо ASDF (через `ASDF:*CENTRAL-REGISTRY`), либо Quicklisp (либо через `QUICKLISP-CLIENT:*LOCAL-PROJECT-DIRECTORIES*` либо доступны в любом из ее дисках)

Как определить тестовую операцию для системы

```
(in-package #:asdf-user)

(defsystem #:foo
  :components ((:file "foo"))
  :in-order-to ((asdf:test-op (asdf:load-op :foo)))
  :perform (asdf:test-op (o c)
                    (uiop:symbol-call :foo-tests 'run-tests)))

(defsystem #:foo-tests
  :name "foo-test"
  :components ((:file "tests")))

;; Afterwards to run the tests we type in the REPL
(asdf:test-system :foo)
```

Заметки:

- Мы предполагаем, что *система* : foo-tests определяет *пакет под* названием «FOO-TESTS»
- `run-tests` - это точка входа для тестового бегуна
- uoip: symbol-call позволяет определить метод, который вызывает функцию, которая еще не была прочитана. Пакет, определяемый функцией, не существует, когда мы определяем систему

В каком пакете я должен определить свою систему ASDF?

ASDF предоставляет пакет `ASDF-USER` разработчикам для определения своих пакетов.

Прочитайте ASDF - другой механизм определения системы онлайн:
https://riptutorial.com/ru/common-lisp/topic/670/asdf---другой-механизм-определения-системы

глава 4: CLOS - общая система объектов Lisp

Examples

Создание базового класса CLOS без родителей

Класс CLOS описывается:

- имя
- список суперклассов
- список слотов
- дополнительные параметры, такие как документация

Каждый слот имеет:

- имя
- форма инициализации (необязательно)
- аргумент инициализации (необязательно)
- тип (необязательно)
- строка документации (необязательно)
- функции доступа, чтения и / или записи (необязательно)
- дополнительные варианты, такие как распределение

Пример:

```
(defclass person ()
  ((name
    :initform      "Erika Mustermann"
    :initarg       :name
    :type          string
    :documentation "the name of a person"
    :accessor      person-name)
   (age
    :initform      25
    :initarg       :age
    :type          number
    :documentation "the age of a person"
    :accessor      person-age))
  (:documentation "a CLOS class for persons with name and age"))
```

Метод печати по умолчанию:

```
(defmethod print-object ((p person) stream)
  "The default print-object method for a person"
  (print-unreadable-object (p stream :type t :identity t)
    (with-slots (name age) p
```

```
      (format stream "Name: ~a, age: ~a" name age))))
```

Создание экземпляров:

```
CL-USER > (make-instance 'person)
#<PERSON Name: Erika Mustermann, age: 25 4020169AB3>

CL-USER > (make-instance 'person :name "Max Mustermann" :age 24)
#<PERSON Name: Max Mustermann, age: 24 4020169FEB>
```

Миксины и интерфейсы

Common Lisp не имеет интерфейсов в том смысле, что некоторые языки (например, Java) работают, и меньше необходимости в этом типе интерфейса, учитывая, что Common Lisp поддерживает множественные наследования и общие функции. Тем не менее, тот же тип шаблонов может быть легко реализован с использованием классов mixin. В этом примере показана спецификация интерфейса коллекции с несколькими соответствующими универсальными функциями.

```
;; Specification of the COLLECTION "interface"

(defclass collection () ()
  (:documentation "A collection mixin."))

(defgeneric collection-elements (collection)
  (:documentation "Returns a list of the elements in the collection."))

(defgeneric collection-add (collection element)
  (:documentation "Adds an element to the collection."))

(defgeneric collection-remove (collection element)
  (:documentation "Removes the element from the collection, if it is present."))

(defgeneric collection-empty-p (collection)
  (:documentation "Returns whether the collection is empty or not."))

(defmethod collection-empty-p ((c collection))
  "A 'default' implementation of COLLECTION-EMPTY-P that tests
whether the list returned by COLLECTION-ELEMENTS is the empty
list."
  (endp (collection-elements c)))
```

Реализация интерфейса - это просто класс, который имеет mixin как один из его суперклассов и определения соответствующих общих функций. (На этом этапе обратите внимание, что класс mixin действительно предназначен только для сигнализации о том, что класс реализует «интерфейс». Этот пример будет работать также с несколькими универсальными функциями и документацией, в которых говорится, что существуют методы для функции класс.)

```
;; Implementation of a sorted-set class
```

```
(defclass sorted-set (collection)
  ((predicate
    :initarg :predicate
    :reader sorted-set-predicate)
   (test
    :initarg :test
    :initform 'eql
    :reader sorted-set-test)
   (elements
    :initform '()
    :accessor sorted-set-elements
    ;; We can "implement" the COLLECTION-ELEMENTS function, that is,
    ;; define a method on COLLECTION-ELEMENTS, simply by making it
    ;; a reader (or accessor) for the slot.
    :reader collection-elements)))

(defmethod collection-add ((ss sorted-set) element)
  (unless (member element (sorted-set-elements ss)
                  :test (sorted-set-test ss))
    (setf (sorted-set-elements ss)
          (merge 'list
                 (list element)
                 (sorted-set-elements ss)
                 (sorted-set-predicate ss)))))

(defmethod collection-remove ((ss sorted-set) element)
  (setf (sorted-set-elements ss)
        (delete element (sorted-set-elements ss))))
```

Наконец, мы видим, что использование экземпляра класса **отсортированного набора** выглядит при использовании «интерфейсных» функций:

```
(let ((ss (make-instance 'sorted-set :predicate '<)))
  (collection-add ss 3)
  (collection-add ss 4)
  (collection-add ss 5)
  (collection-add ss 3)
  (collection-remove ss 5)
  (collection-elements ss))
;; => (3 4)
```

Прочитайте CLOS - общая система объектов Lisp онлайн: https://riptutorial.com/ru/common-lisp/topic/673/clos---общая-система-объектов-lisp

глава 5: LOOP, общий макрос Lisp для итерации

Examples

Ограниченные петли

Мы можем повторить действие несколько раз, используя `repeat` .

```
CL-USER> (loop repeat 10 do (format t "Hello!~%"))
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
NIL
CL-USER> (loop repeat 10 collect (random 50))
(28 46 44 31 5 33 43 35 37 4)
```

Пересечение последовательностей

```
(loop for i in '(one two three four five six)
      do (print i))
(loop for i in '(one two three four five six) by #'cddr
      do (print i)) ;prints ONE THREE FIVE

(loop for i on '(a b c d e f g)
      do (print (length i))) ;prints 7 6 5 4 3 2 1
(loop for i on '(a b c d e f g) by #'cddr
      do (print (length i))) ;prints 7 5 3 1
(loop for i on '(a b c)
      do (print i)) ;prints (a b c) (b c) (c)

(loop for i across #(1 2 3 4 5 6)
      do (print i)) ; prints 1 2 3 4 5 6
(loop for i across "foo"
      do (print i)) ; prints #\f #\o #\o
(loop for element across "foo"
      for i from 0
      do (format t "~a ~a~%" i element)) ; prints 0 f\n1 o\n1 o
```

Вот краткое описание ключевых слов

Ключевое слово	Тип последовательности	Переменный тип
в	список	элемент списка
на	список	некоторый cdr списка
через	вектор	элемент вектора

Пересечение таблиц Hash

```
(defvar *ht* (make-hash-table))
(loop for (sym num) on
        '(one 1 two 2 three 3 four 4 five 5 six 6 seven 7 eight 8 nine 9 ten 10)
        by #'cddr
      do (setf (gethash sym *ht*) num))

(loop for k being each hash-key of *ht*
      do (print k)) ; iterate over the keys
(loop for k being the hash-keys in *ht* using (hash-value v)
      do (format t "~a=>~a~%" k v))
(loop for v being the hash-value in *ht*
      do (print v))
(loop for v being each hash-values of *ht* using (hash-key k)
      do (format t "~a=>~a~%" k v))
```

Простая форма LOOP

Простая форма LOOP без специальных ключевых слов:

```
(loop forms...)
```

Чтобы выйти из цикла, мы можем использовать `(return <return value>)` `

Некоторые примеры:

```
(loop (format t "Hello~%"))  ; prints "Hello" forever
(loop (print (eval (read)))) ; your very own REPL
(loop (let ((r (read)))
        (typecase r
         (number (return (print (* r r))))
         (otherwise (format t "Not a number!~%")))))
```

Заполнение пакетов

```
(loop for s being the symbols in 'cl
      do (print s))
(loop for s being the present-symbols in :cl
      do (print s))
(loop for s being the external-symbols in (find-package "COMMON LISP")
      do (print s))
(loop for s being each external-symbols of "COMMON LISP"
      do (print s))
```

```
(loop for s being each external-symbol in pack ;pack is a variable containing a package
      do (print s))
```

Арифметические петли

```
(loop for i from 0 to 10
      do (print i)) ; prints 0 1 2 3 4 5 6 7 8 9 10
(loop for i from 0 below 10
      do (print i)) ; prints 0 1 2 3 4 5 6 7 8 9 10
(loop for i from 10 above 0
      do (print i)) ; prints 10 9 8 7 6 5 4 3 2 1
(loop for i from 10 to 0
      do (print i)) ; prints nothing
(loop for i from 10 downto 0
      do (print i)) ; prints 10 9 8 7 6 5 4 3 2 1 0
(loop for i downfrom 10 to 0
      do (print i)) ; same as above
(loop for i from 1 to 100 by 10
      do (print i)) ; prints 1 11 21 31 41 51 61 71 81 91
(loop for i from 100 downto 0 by 10
      do (print i)) ; prints 100 90 80 70 60 50 40 30 20 10 0
(loop for i from 1 to 10 by (1+ (random 3))
      do (print i)) ; note that (random 3) is evaluated only once
(let ((step (random 3)))
  (loop for i from 1 to 10 by (+ step 1)
        do (print i))) ; equivalent to the above
(loop for i from 1 to 10
      for j from 11 by 11
      do (format t "~2d ~3d~%" i j)) ;prints 1 11\n2 22\n...10 110
```

Разрушение в операциях FOR

Мы можем разрушить списки сложных объектов

```
CL-USER> (loop for (a . b) in '((1 . 2) (3 . 4) (5 . 6)) collect a)
(1 3 5)
CL-USER> (loop for (a . b) in '((1 . 2) (3 . 4) (5 . 6)) collect b)
(2 4 6)
CL-USER> (loop for (a b c) in '((1 2 3) (4 5 6) (7 8 9) (10 11 12)) collect b)
(2 5 8 11)
```

Мы также можем разрушить список

```
CL-USER> (loop for (a . b) on '(1 2 3 4 5 6) collect a)
(1 2 3 4 5 6)
CL-USER> (loop for (a . b) on '(1 2 3 4 5 6) collect b)
((2 3 4 5 6) (3 4 5 6) (4 5 6) (5 6) (6) NIL)
```

Это полезно, когда мы хотим перебирать только определенные элементы

```
CL-USER> (loop for (a . b) on '(1 2 3 4 5 6) by #'cddr collect a)
(1 3 5)
CL-USER> (loop for (a . b) on '(1 2 3 4 5 6) by #'cdddr collect a)
(1 4)
```

Использование `NIL` для игнорирования термина:

```
(loop for (a nil . b) in '((1 2 . 3) (4 5 . 6) (7 8 . 9))
      collect (list a b)) ;=> ((1 3) (4 6) (7 9))
(loop for (a b) in '((1 2) (3 4) (5 6)) ;(a b) == (a b . nil)
      collect (+ a b)) ;=> (3 7 11)

; iterating over a window in a list
(loop for (pre x post) on '(1 2 3 4 5 3 2 1 2 3 4)
      for nth from 1
      while (and x post) ; checks that we have three elements of the list
      if (and (<= post x) (<= pre x)) collect (list :max x nth)
      if (and (>= post x) (>= pre x)) collect (list :min x nth))
; The above collects local minima/maxima
```

LOOP как выражение

В отличие от циклов почти на любом другом используемом сегодня языке программирования, `LOOP` в Common Lisp может использоваться как выражение:

```
(let ((doubled (loop for x from 1 to 10
                     collect (* 2 x))))
    doubled) ;; ==> (2 4 6 8 10 12 14 16 18 20)

(loop for x from 1 to 10 sum x)
```

`MAXIMIZE` заставляет `LOOP` возвращать наибольшее значение, которое было оценено. `MINIMIZE` - это противоположность `MAXIMIZE` .

```
(loop repeat 100
      for x = (random 1000)
      maximize x)
```

`COUNT` сообщает вам, сколько раз выражение оценивалось без `NIL` во время цикла:

```
(loop repeat 100
      for x = (random 1000)
      count (evenp x))
```

`LOOP` также имеет эквиваленты `some` , `every` и не `notany` функций:

```
(loop for ch across "foobar"
     thereis (eq ch #\a))

(loop for x in '(a b c d e f 1)
    always (symbolp x))

(loop for x in '(1 3 5 7)
    never (evenp x))
```

... за исключением того, что они не ограничиваются итерацией над последовательностями:

```
(loop for value = (read *standard-input* nil :eof)
   until (eq value :eof)
   never (stringp value))
```

`LOOP` -генерирующие значение глаголы также могут быть записаны с помощью суффикса -ing:

```
(loop repeat 100
      for x = (random 1000)
      minimizing x)
```

Также возможно записать значение, генерируемое этими глаголами, в переменные (которые создаются неявно макросом `LOOP`), поэтому вы можете генерировать более одного значения за раз:

```
 (loop repeat 100
     for x = (random 1000)
     maximizing x into biggest
     minimizing x into smallest
     summing x into total
     collecting x into xs
     finally (return (values biggest smallest total xs)))
```

Вы можете иметь более чем один `collect` , `count` и т.д. пункт, собирающий в том же значении выходного сигнала. Они будут выполняться последовательно.

Следующее преобразует список ассоциаций (который вы можете использовать с `assoc`) в список свойств (который вы можете использовать с `getf`):

```
(loop for (key . value) in assoc-list
      collect key
      collect value)
```

Хотя это лучший стиль:

```
(loop for (key . value) in assoc-list
      append (list key value))
```

Условное выполнение предложений LOOP

`LOOP` имеет свой собственный оператор `IF` который может управлять выполнением предложений:

```
(loop repeat 1000
      for x = (random 100)
      if (evenp x)
        collect x into evens
      else
        collect x into odds
      finally (return (values evens odds)))
```

Объединение нескольких предложений в тело IF требует специального синтаксиса:

```
 (loop repeat 1000
       for x = (random 100)
       if (evenp x)
          collect x into evens
          and do (format t "~a is even!~%" x)
       else
          collect x into odds
          and count t into n-odds
       finally (return (values evens odds n-odds)))
```

Параллельная итерация

В `LOOP` допускаются несколько предложений `FOR` . Цикл заканчивается, когда заканчивается первый из этих статей:

```
(loop for a in '(1 2 3 4 5)
      for b in '(a b c)
      collect (list a b))
;; Evaluates to: ((1 a) (2 b) (3 c))
```

Другие предложения, которые определяют, следует ли продолжать цикл, можно комбинировать:

```
(loop for a in '(1 2 3 4 5 6 7)
      while (< a 4)
      collect a)
;; Evaluates to: (1 2 3)

(loop for a in '(1 2 3 4 5 6 7)
      while (< a 4)
      repeat 1
      collect a)
;; Evaluates to: (1)
```

Определите, какой список длиннее, отрезав итерацию, как только ответ будет известен:

```
(defun longerp (list-1 list-2)
    (loop for cdr1 on list-1
          for cdr2 on list-2
          if (null cdr1) return nil
          else if (null cdr2) return t
          finally (return nil)))
```

Нумерация элементов списка:

```
(loop for item in '(a b c d e f g)
      for x from 1
      collect (cons x item))
;; Returns ((1 . a) (2 . b) (3 . c) (4 . d) (5 . e) (6 . f) (7 . g))
```

Убедитесь, что все номера в списке четные, но только для первых 100 элементов:

```
(assert
   (loop for number in list
         repeat 100
         always (evenp number)))
```

Вложенная итерация

Специальный синтаксис `LOOP NAMED foo` позволяет создать цикл, из которого вы можете выйти раньше. Выход выполняется с использованием `return-from` и может использоваться из внутри вложенных циклов.

Ниже используется вложенный цикл для поиска сложного числа в 2D-массиве:

```
(loop named top
      for x from 0 below (array-dimension *array* 1)
      do (loop for y from 0 below (array-dimension *array* 0))
               for n = (aref *array* y x)
              when (complexp n)
                do (return-from top (values n x y))))
```

RETURN и RETURN.

В пределах `LOOP` вы можете использовать форму Common Lisp `(return)` в любом выражении, что приведет к тому, что форма `LOOP` немедленно оценит значение, указанное для `return` .

`LOOP` также имеет предложение `return` , которое работает почти одинаково, с той лишь разницей, что вы не окружаете его круглыми скобками. Предложение используется в DSL `LOOP` , в то время как форма используется в выражениях.

```
(loop for x in list
      do (if (listp x) ;; Non-barewords after DO are expressions
             (return :x-has-a-list)))

;; Here, both the IF and the RETURN are clauses
(loop for x in list
     if (listp x) return :x-has-a-list)

;; Evaluate the RETURN expression and assign it to X...
;; except RETURN jumps out of the loop before the assignment
;; happens.
(loop for x = (return :nothing-else-happens)
      do (print :this-doesnt-print))
```

Вещь после `finally` должна быть выражением, поэтому необходимо использовать форму `(return)` а не предложение `return` :

```
 (loop for n from 1 to 100
       when (evenp n) collect n into evens
       else collect n into odds
```

```
      finally return (values evens odds)) ;; ERROR!

 (loop for n from 1 to 100
       when (evenp n) collect n into evens
       else collect n into odds
      finally (return (values evens odds))) ;; Correct usage.
```

Зацикливание над окном списка

Некоторые примеры для окна размером 3:

```
;; Naïve attempt:
(loop for (first second third) on '(1 2 3 4 5)
      do (print (* first second third)))
;; prints 6 24 60 then Errors on (* 4 5 NIL)

;; We will try again and put our attempt into a function
(defun loop-3-window1 (function list)
  (loop for (first second third) on list
        while (and second third)
        do (funcall function first second third)))
(loop-3-window1 (lambda (a b c) (print (* a b c))) '(1 2 3 4 5))
;; prints 6 24 60 and returns NIL
(loop-3-window1 (lambda (a b c) (print (list a b c))) '(a b c d nil nil e f))
;; prints (a b c) (b c d) then returns NIL

;; A second attempt
(defun loop-3-window2 (function list)
  (loop for x on list
        while (nthcdr 2 x) ;checks if there are at least 3 elements
        for (first second third) = x
        do (funcall function first second third)))
(loop-3-window2 (lambda (a b c) (print (list a b c))) '(a b c d nil nil e f))
;; prints (a b c) (b c d) (c d nil) (c nil nil) (nil nil e) (nil e f)

;; A (possibly) more efficient function:
(defun loop-3-window2 (function list)
  (let ((f0 (pop list))
        (s0 (pop list)))
    (loop for first = f0 then second
          and second = s0 then third
          and third in list
          do (funcall function first second third))))

;; A more general function:
(defun loop-n-window (n function list)
  (loop for x on list
        while (nthcdr (1- n) x)
        do (apply function (subseq x 0 n))))
;; With potentially efficient implementation:
(define-compiler-macro loop-n-window (n function list &whole w)
  (if (typep n '(integer 1 #.call-arguments-limit))
     (let ((vars (loop repeat n collect (gensym)))
           (vars0 (loop repeat (1- n) collect (gensym)))
           (lst (gensym)))
       `(let ((,lst ,list))
          (let ,(loop for v in vars0 collect `(,v (pop ,lst)))
            (loop for
                  ,@(loop for v0 in vars0 for (v vn) on vars
```

```
                   collect v collect '= collect v0 collect 'then collect vn
                   collect 'and)
                ,(car (last vars)) in ,lst
                do ,(if (and (consp function) (eq 'function (car function))
    w
```

Прочитайте LOOP, общий макрос Lisp для итерации онлайн:
https://riptutorial.com/ru/common-lisp/topic/1369/loop--общий-макрос-lisp-для-итерации

глава 6: Streams

Синтаксис

- `(read-char &optional stream eof-error-p eof-value recursive-p)` => character
- `(write-char character &optional stream)` => символ
- `(read-line &optional stream eof-error-p eof-value recursive-p)` => line, missing-newline-p
- `(write-line line &optional stream)` => строка

параметры

параметр	подробность
`stream`	Поток для чтения или записи.
`eof-error-p`	Если сообщение об ошибке будет сообщено, если обнаружен конец файла.
`eof-value`	Какое значение должно быть возвращено, если eof встречается, а `eof-error-p` - false.
`recursive-p`	Является ли операция чтения называется рекурсивно из `READ` . Обычно это следует оставить как `NIL` .
`character`	Персонаж для записи или символ, который был прочитан.
`line`	Строка для записи или строка, которая была прочитана.

Examples

Создание входных потоков из строк

Макрос `WITH-INPUT-FROM-STRING` может использоваться для создания потока из строки.

```
(with-input-from-string (str "Foobar")
  (loop for i from 0
        for char = (read-char str nil nil)
        while char
        do (format t "~d: ~a~%" i char)))
; 0: F
; 1: o
; 2: o
; 3: b
; 4: a
; 5: r
```

```
;=> NIL
```

То же самое можно сделать вручную, используя MAKE-STRING-INPUT-STREAM .

```
(let ((str (make-string-input-stream "Foobar")))
  (loop for i from 0
        for char = (read-char str nil nil)
        while char
        do (format t "~d: ~a~%" i char)))
```

Запись вывода в строку

Макрос WITH-OUTPUT-TO-STRING можно использовать для создания потока вывода строки и возвращать результирующую строку в конце.

```
(with-output-to-string (str)
  (write-line "Foobar!" str)
  (write-string "Barfoo!" str))
;=> "Foobar!
;   Barfoo!"
```

То же самое можно сделать вручную с помощью MAKE-STRING-OUTPUT-STREAM и GET-OUTPUT-STREAM-STRING .

```
(let ((str (make-string-output-stream)))
  (write-line "Foobar!" str)
  (write-string "Barfoo!" str)
  (get-output-stream-string str))
```

Серые потоки

Серые потоки - это нестандартное расширение, которое позволяет определять потоки, определенные пользователем. Он предоставляет классы и методы, которые пользователь может расширить. Вы должны проверить свое руководство по реализации, чтобы убедиться, что оно обеспечивает потоки Gray.

Для простого примера поток ввода символов, который возвращает случайные символы, может быть реализован следующим образом:

```
(defclass random-character-input-stream (fundamental-character-input-stream)
  ((character-table
    :initarg :character-table
    :initform "abcdefghijklmnopqrstuvwxyz
" ; The newline is necessary.
    :accessor character-table))
  (:documentation "A stream of random characters."))

(defmethod stream-read-char ((stream random-character-input-stream))
  (let ((table (character-table stream)))
    (aref table (random (length table)))))
```

```
(let ((stream (make-instance 'random-character-input-stream)))
  (dotimes (i 5)
    (print (read-line stream))))
; "gyaexyfjsqdcpciaaftoytsygdeycrrzwivwcfb"
; "gctnoxpajovjqjbkiqykdflbhfspmexjaaggonhydhayvknwpdydyiabithpt"
; "nvfxwzczfalosaqw"
; "sxeiejcovrtesbpmoppfvvjfvx"
; "hjplqgstbodbalnmxhsvxdox"
;=> NIL
```

Чтение файла

Файл можно открыть для чтения в виде потока, используя макрос `WITH-OPEN-FILE` .

```
(with-open-file (file #P"test.file")
  (loop for i from 0
        for line = (read-line file nil nil)
        while line
        do (format t "~d: ~a~%" i line)))
; 0: Foobar
; 1: Barfoo
; 2: Quuxbar
; 3: Barquux
; 4: Quuxfoo
; 5: Fooquux
;=> T
```

То же самое можно сделать вручную с помощью `OPEN` и `CLOSE` .

```
(let ((file (open #P"test.file"))
      (aborted t))
  (unwind-protect
       (progn
         (loop for i from 0
               for line = (read-line file nil nil)
               while line
               do (format t "~d: ~a~%" i line))
         (setf aborted nil))
    (close file :abort aborted)))
```

Обратите внимание, что `READ-LINE` создает новую строку для каждой строки. Это может быть медленным. Некоторые реализации предоставляют вариант, который может читать строку в строковый буфер. Пример: `READ-LINE-INTO` для Allegro CL.

Запись в файл

Файл можно открыть для записи в виде потока, используя макрос `WITH-OPEN-FILE` .

```
(with-open-file (file #P"test.file" :direction :output
                                    :if-exists :append
                                    :if-does-not-exist :create)
  (dolist (line '("Foobar" "Barfoo" "Quuxbar"
                  "Barquux" "Quuxfoo" "Fooquux"))
```

```
    (write-line line file)))
```

То же самое можно сделать вручную с помощью `OPEN` и `CLOSE` .

```
(let ((file (open #P"test.file" :direction :output
                                :if-exists :append
                                :if-does-not-exist :create)))
  (dolist (line '("Foobar" "Barfoo" "Quuxbar"
                  "Barquux" "Quuxfoo" "Fooquux"))
    (write-line line file))
  (close file))
```

Копирование файла

Копировать байт за байт файла

Следующая функция копирует файл в другой, выполняя точную копию байта за байт, игнорируя вид контента (который может быть либо строками символов в некоторых кодировках, либо двоичными данными):

```
(defun byte-copy (infile outfile)
  (with-open-file (instream infile :direction :input :element-type '(unsigned-byte 8)
                           :if-does-not-exist nil)
    (when instream
      (with-open-file (outstream outfile :direction :output :element-type '(unsigned-byte 8)
                                 :if-exists :supersede)
        (loop for byte = (read-byte instream nil)
           while byte
           do (write-byte byte outstream))))))
```

Тип `(unsigned-byte 8)` - это тип 8-разрядных байтов. Функции `read-byte` и `write-byte` работают в байтах, а не `read-char` и `write-char` которые работают с символами. `read-byte` возвращает байт, считанный из потока, или `NIL` в конце файла, если вторым необязательным параметром является `NIL` (иначе он сигнализирует об ошибке).

Массовая копия

Точная копия, более эффективная предыдущая. может быть сделано путем чтения и записи файлов с большими фрагментами данных каждый раз, вместо одиночных байтов:

```
(defun bulk-copy (infile outfile)
  (with-open-file (instream infile :direction :input :element-type '(unsigned-byte 8)
                           :if-does-not-exist nil)
    (when instream
      (with-open-file (outstream outfile :direction :output :element-type '(unsigned-byte 8)
                                 :if-exists :supersede)
        (let ((buffer (make-array 8192 :element-type '(unsigned-byte 8))))
          (loop for bytes-read = (read-sequence buffer instream)
             while (plusp bytes-read)
             do (write-sequence buffer outstream :end bytes-read)))))))
```

`read-sequence`

`write-sequence` используются здесь с буфером, который представляет собой вектор байтов (они могут работать с последовательностями байтов или символов). `read-sequence` заполняет массив байтами, считываемыми каждый раз, и возвращает количество прочитанных байтов (которое может быть меньше размера массива, когда достигнут конец файла). Обратите внимание, что массив подвергается деструктивной модификации на каждой итерации.

Точная копия строки в строке файла

Последний пример - это копия, выполняемая путем чтения каждой строки символов входного файла и записи ее в выходной файл. Обратите внимание: поскольку мы хотим получить точную копию, мы должны проверить, завершена ли последняя строка входного файла или нет символом конца строки. По этой причине мы используем два значения, возвращаемые `read-line` : новая строка, содержащая символы следующей строки, и логическое значение, которое *истинно,* если строка является последней из файла и не содержит окончательный символ новой строки (ы). В этом случае вместо `write-string` используется `write-string` `write-line` , так как первая не добавляет новую строку в конце строки.

```
(defun line-copy (infile outfile)
  (with-open-file (instream infile :direction :input :if-does-not-exist nil)
    (when instream
      (with-open-file (outstream outfile :direction :output :if-exists :supersede)
        (let (line missing-newline-p)
          (loop
             (multiple-value-setq (line missing-newline-p)
               (read-line instream nil nil))
             (cond (missing-newline-p                          ; we are at the end of file
                    (when line (write-string line outstream))  ; note `write-string`
                    (return))                                  ; exit from simple loop
                   (t (write-line line outstream)))))))))
```

Обратите внимание, что эта программа независима от платформы, так как символ новой строки (меняющийся в разных операционных системах) автоматически управляется функциями `read-line` и `write-line` .

Чтение и запись целых файлов в строки и из них

Следующая функция считывает весь файл в новую строку и возвращает его:

```
(defun read-file (infile)
  (with-open-file (instream infile :direction :input :if-does-not-exist nil)
    (when instream
      (let ((string (make-string (file-length instream))))
        (read-sequence string instream)
        string))))
```

Результатом является `NIL` если файл не существует.

Следующая функция записывает строку в файл. Параметр ключевого слова используется

для указания того, что делать, если файл уже существует (по умолчанию он вызывает ошибку, допустимые значения являются значениями макроса `with-open-file`).

```
(defun write-file (string outfile &key (action-if-exists :error))
   (check-type action-if-exists (member nil :error :new-version :rename :rename-and-delete
                                        :overwrite :append :supersede))
   (with-open-file (outstream outfile :direction :output :if-exists action-if-exists)
     (write-sequence string outstream)))
```

В этом случае `write-sequence` может быть заменена на `write-string` .

Прочитайте Streams онлайн: https://riptutorial.com/ru/common-lisp/topic/3028/streams

глава 7: Группировка форм

Examples

Когда нужно группировать?

В некоторых местах в Common Lisp ряд форм оценивается по порядку. Например, в теле **defun** или **лямбда** , или в теле **доты** . В этих случаях запись нескольких форм в порядке работает, как ожидалось. Однако в нескольких местах, таких как части *then* и *else* выражения **if** , допускается только одна форма. Конечно, можно захотеть на самом деле оценить несколько выражений в этих местах. Для этих ситуаций требуется какая-то неявная явная форма группировки.

Progn

Общего назначения специальный оператор **progn** используется для оценки ноль или более форм. Возвращается значение последней формы. Например, в следующем случае **(print 'hello)** оценивается (и его результат игнорируется), а затем **42** оценивается и возвращается его результат (**42**):

```
(progn
  (print 'hello)
  42)
;=> 42
```

Если в **запросе** нет форм, то возвращается **nil** :

```
(progn)
;=> NIL
```

Помимо группировки ряда форм, **progn** также имеет важное свойство, что если форма **progn** является формой *верхнего уровня* , то все формы внутри нее обрабатываются как формы верхнего уровня. Это может быть важно при написании макросов, которые расширяются в несколько форм, которые все должны обрабатываться как формы верхнего уровня.

Progn также ценен тем, что он возвращает *все* значения последней формы. Например,

```
(progn
  (print 'hello)
  (values 1 2 3))
;;=> 1, 2, 3
```

Напротив, некоторые выражения группировки возвращают *первичное значение* формы, создающей результат.

Неявные Прогнозы

Некоторые формы используют *неявные прогнозы* для описания их поведения. Например, **когда** и **если** макросы, которые по существу являются односторонними , **если** формы, описать их поведение с точки зрения *неявного progn.* Это означает, что такая форма, как

```
(when (foo-p foo)
  form1
  form2)
```

и условие **(foo-p foo)** истинно, тогда *форма 1* и *форма* 2 сгруппированы так, как если бы они содержались в **прогнозе** . Расширение **когда** это макрос, по существу:

```
(if (foo-p foo)
  (progn
    form1
    form2)
  nil)
```

Prog1 и Prog2

Часто бывает полезно оценить несколько выражений и вернуть результат из первой или второй формы, а не из последней. Это легко сделать, используя **let** и, например:

```
(let ((form1-result form1))
  form2
  form3
  ;; ...
  form-n-1
  form-n
  form1-result)
```

Поскольку эта форма распространена в некоторых приложениях, Common Lisp включает **prog1** и **prog2** , которые похожи на **progn** , но возвращают результат первой и второй форм, соответственно. Например:

```
(prog1
  42
  (print 'hello)
  (print 'goodbye))
;; => 42
```

```
(prog2
  (print 'hello)
  42
  (print 'goodbye))
;; => 42
```

Однако важное различие между **prog1** / **prog2** и **progn** заключается в том, что **progn**

возвращает *все* значения последней формы, тогда как **prog1** и **prog2** возвращают первичное значение первой и второй форм. Например:

```
(progn
  (print 'hello)
  (values 1 2 3))
;;=> 1, 2, 3

(prog1
  (values 1 2 3)
  (print 'hello))
;;=> 1              ; not 1, 2, 3
```

Для нескольких значений с **оценкой** стиля **prog1** вместо этого используйте **multiple-value-prog1** . Нет аналогичного **многозначного prog2** , но его нетрудно реализовать, если вам это нужно.

блок

Специальный **блок** оператора позволяет группировать несколько форм Лиспа (например, неявное `progn`), а также *имя,* чтобы назвать этот блок. Когда формы внутри блока оцениваются, для выхода из блока можно использовать специальный оператор **return-from** . Например:

```
(block foo
  (print 'hello)     ; evaluated
  (return-from foo)
  (print 'goodbye))  ; not evaluated
;;=> NIL
```

return-from также может быть предоставлено возвращаемое значение:

```
(block foo
  (print 'hello)     ; evaluated
  (return-from foo 42)
  (print 'goodbye))  ; not evaluated
;;=> 42
```

Именованные блоки полезны, когда кусок кода имеет значимое имя или когда блоки вложены. В каком-то контексте важна только возможность возврата из блока раньше. В этом случае вы можете использовать **nil в** качестве имени блока и **вернуться** . **Возврат** точно так же, как **return-from** , за исключением того, что имя блока всегда равно **нулю** .

Примечание: закрытые формы не являются формами верхнего уровня. Это отличается от `progn` , где вложенные формы с верхним уровнем `progn` формы по - прежнему считаются формами *верхнего уровня.*

Tagbody

Для большого контроля в групповых формах специальный оператор **tagbody** может быть очень полезным. Формы внутри формы **тегов** - это либо *теги go* (которые являются просто символами или целыми числами), либо формы для выполнения. Внутри **тега** используется специальный оператор **go** для передачи выполнения в новое место. Этот тип программирования можно считать довольно низким, так как он допускает произвольные пути выполнения. Ниже приведен подробный пример того, как может выглядеть цикл for при реализации в качестве **тега** :

```
(let (x)                ; for (x = 0; x < 5; x++) { print(hello); }
  (tagbody
     (setq x 0)
   prologue
     (unless (< x 5)
       (go end))
   begin
     (print (list 'hello x))
   epilogue
     (incf x)
     (go prologue)
   end))
```

Хотя **tagbody** и **go** обычно не используются, возможно, из-за «GOTO считается вредным», но могут быть полезны при реализации сложных структур управления, таких как государственные машины. Многие итерационные конструкции также расширяются в *неявный тег* . Например, тело **топимов** указано как серия тегов и форм.

Какую форму использовать?

При написании макросов, которые расширяются в формы, которые могут включать группировку, стоит потратить некоторое время на рассмотрение того, какая структура группировки будет расширяться.

Для форм стиля определения, например, макроса **define-widget** , который обычно отображается как форма верхнего уровня, и что несколько **defun** s, **defstruct** s и т. Д., Обычно имеет смысл использовать **progn** , так что дочерние формы обрабатываются как формы верхнего уровня. Для итерационных форм чаще всего используется неявный **тег** .

Например, тело **DOTIMES** , **DOLIST** , и **сделать** каждый расшириться в неявном **tagbody.**

Для форм, которые определяют именованный «кусок» кода, часто используется неявный **блок** . Например, в то время как тело **defun** находится внутри неявного **прогноза** , этот неявный **progn** находится внутри блока, совместно использующего имя функции. Это означает, что **return-from** может использоваться для выхода из функции. Такой комп

Прочитайте Группировка форм онлайн: https://riptutorial.com/ru/common-lisp/topic/4892/группировка-форм

глава 8: Единичное тестирование

Examples

Использование FiveAM

Загрузка библиотеки

```
(ql:quickload "fiveam")
```

Определить тестовый пример

```
(fiveam:test sum-1
  (fiveam:is (= 3 (+ 1 2))))

;; We'll also add a failing test case
(fiveam:test sum2
  (fiveam:is (= 4 (+ 1 2))))
```

Выполнить тесты

```
(fiveam:run!)
```

который сообщает

```
Running test suite NIL
 Running test SUM2 f
 Running test SUM1 .
 Did 2 checks.
    Pass: 1 (50%)
    Skip: 0 ( 0%)
    Fail: 1 (50%)
 Failure Details:
 --------------------------------
 SUM2 []:

(+ 1 2)

 evaluated to

3

 which is not

=
```

```
 to

4

..
 --------------------------------
NIL
```

Заметки

- Тесты сгруппированы по наборам тестов
- По умолчанию тесты добавляются в глобальный набор тестов

Вступление

Существует несколько библиотек для модульного тестирования в Common Lisp

- FiveAM
- Докажите , с несколькими уникальными функциями, такими как обширные тестовые репортеры, цветной вывод, отчет о продолжительности теста и интеграция asdf.
- Lisp-Unit2 , аналогичный JUnit
- Fiasco , уделяя особое внимание предоставлению хорошего опыта тестирования REPL. Преемник hu.dwim.stefil

Прочитайте Единичное тестирование онлайн: https://riptutorial.com/ru/common-lisp/topic/2349/единичное-тестирование

глава 9: Консоли и списки

Examples

Списки как конвенция

Некоторые языки включают структуру данных списка. Common Lisp и другие языки семейства Lisp широко используют списки (а имя Lisp основано на идее LISt-процессора). Однако Common Lisp фактически не содержит примитивный тип данных списка. Вместо этого списки существуют по соглашению. Конвенция зависит от двух принципов:

1. Символ **nil** - пустой список.
2. Непустым списком является *ячейка cons,* чей *автомобиль* является первым элементом списка и чей *cdr* является остальной частью списка.

Это все, что есть в списках. Если вы прочитали пример, называемый *What is cons cell?* , то вы знаете, что ячейка cons, автомобиль которой X, а cdr - Y, может быть записана как **(X, Y)** . Это означает, что мы можем написать несколько списков на основе вышеизложенных принципов. Список элементов 1, 2 и 3 просто:

```
(1 . (2 . (3 . nil)))
```

Однако, поскольку списки настолько распространены в семействе языков Lisp, существуют специальные соглашения о печати за пределами простой пунктирной пары для cons-ячеек.

1. Символ **nil** также может быть записан как **()** .
2. Когда cdr одной cons-ячейки является другим списком (либо **(),** либо ячейкой cons), вместо того, чтобы писать одну cons-ячейку с точечной парной нотацией, используется «запись списка».

Обозначение списка показано наиболее четко несколькими примерами:

```
(x . (y . z))   === (x y . z)
(x . NIL)       === (x)
(1 . (2 . NIL)) === (1 2)
(1 . ())        === (1)
```

Идея состоит в том, что элементы списка записываются в последовательном порядке в круглых скобках до тех пор, пока не будет достигнут окончательный cdr в списке. Если конечный cdr равен **nil** (пустой список), тогда записывается окончательная скобка. Если конечный cdr не равен **nil** (в этом случае список называется *неправильным списком*), то записывается точка, а затем записывается окончательный cdr.

Что такое ячейка cons?

Ячейка cons, также известная как пунктирная пара (из-за ее напечатанного представления), представляет собой просто пару из двух объектов. Ячейка cons создается функцией `cons` , а элементы в паре извлекаются с помощью функций `car` и `cdr` .

```
(cons "a" 4)
```

Например, это возвращает пару, чей первый элемент (который может быть извлечен `car`) является `"a"` , а второй элемент (который может быть извлечен `cdr`) равен `4` .

```
(car (cons "a" 4))
;;=> "a"

(cdr (cons "a" 4))
;;=> 3
```

Символьные ячейки могут быть напечатаны в виде *точечной пары* :

```
(cons 1 2)
;;=> (1 . 2)
```

Консервные ячейки также могут быть прочитаны в виде точечных парных обозначений, так что

```
(car '(x . 5))
;;=> x

(cdr '(x . 5))
;;=> 5
```

(Печатная форма cons-ячеек также может быть немного сложнее. Подробнее об этом см. В примере о cons-ячейках в виде списков.)

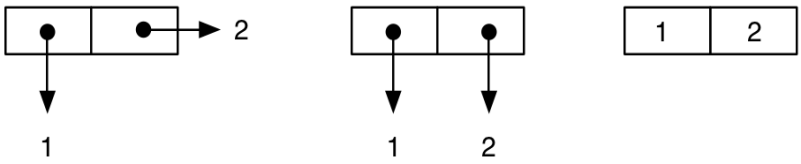
Это оно; cons cells - это только пары элементов, созданных функцией `cons` , и элементы могут быть извлечены `car` и `cdr` . Из-за их простоты ячейки cons могут быть полезным строительным блоком для более сложных структур данных.

Эскизные экраны

Для лучшего понимания семантики conses и списков часто используется графическое представление таких структур. Ячейка cons обычно представлена двумя контактами, содержащими две стрелки, указывающие на значения `car` и `cdr` , или непосредственно значения. Например, результат:

```
(cons 1 2)
;; -> (1 . 2)
```

могут быть представлены одним из этих чертежей:

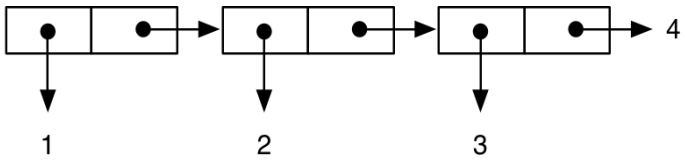


Обратите внимание, что эти представления являются чисто концептуальными и не означают, что значения *содержатся* в ячейке или *указываются* из ячейки: в общем, это зависит от реализации, типа значений, уровня оптимизации и т. Д. . В остальной части примера мы будем использовать первый вид чертежа, который является наиболее часто используемым.

Так, например:

```
(cons 1 (cons 2 (cons 3 4)))   ; improper “dotted” list
;; -> (1 2 3 . 4)
```

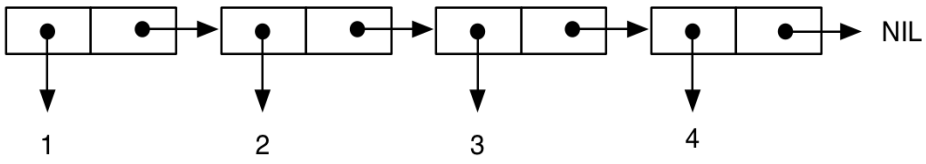
представлен как:



в то время как:

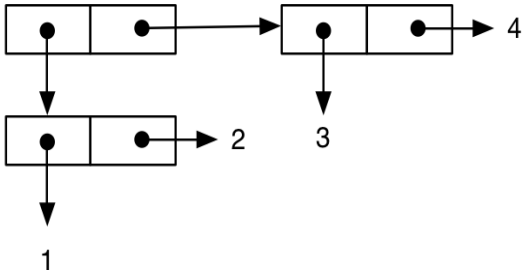
```
(cons 1 (cons 2 (cons 3 (cons 4 nil))))  ;; proper list, equivalent to: (list 1 2 3 4)
;; -> (1 2 3 4)
```

представлен как:



Вот древовидная структура:

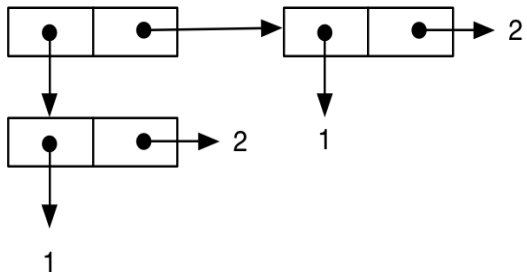
```
(cons (cons 1 2) (cons 3 4))
;; -> ((1 . 2) 3 . 4)         ; note the printing as an improper list
```



В последнем примере показано, как это обозначение может помочь нам понять важные семантические аспекты языка. Во-первых, мы пишем выражение, подобное предыдущему:

```
(cons (cons 1 2) (cons 1 2))
;; -> ((1 . 2) 1 . 2)
```

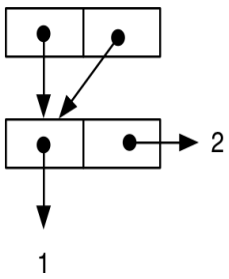
которые могут быть представлены обычным образом:



Затем мы пишем другое выражение, которое, по-видимому, эквивалентно предыдущему, и это подтверждается печатным представлением результата:

```
(let ((cell-a (cons 1 2)))
  (cons cell-a cell-a))
;; -> ((1 . 2) 1 . 2)
```

Но, если мы рисуем диаграмму, мы можем видеть, что семантика выражения различна, поскольку `cell-a` та *же* ячейка является значением *как* части `car` части `cdr` внешних `cons` (это - `cell-a` является *общей*) :



и тот факт, что семантика двух результатов на самом деле отличается на уровне языка, может быть проверена следующими тестами:

```
(let ((c1 (cons (cons 1 2) (cons 1 2)))
      (c2 (let ((cell-a (cons 1 2)))
            (cons cell-a cell-a))))
  (list (eq (car c1) (cdr c1))
        (eq (car c2) (cdr c2)))
;; -> (NIL T)
```

Первый `eq` является *ложным,* так как `car` и `cdr c1` структурно равны (это *верно* `equal`), но не являются «идентичными» (т. Е. «Одна и та же общая структура»), а во втором тесте

результат *верен,* поскольку `car` и `cdr c2` *идентичны* , то есть они имеют *одинаковую структуру* .

Прочитайте Консоли и списки онлайн: https://riptutorial.com/ru/common-lisp/topic/2622/консоли-и-списки

глава 10: Контрольные структуры

Examples

Условные конструкции

В Common Lisp, `if` это простейшая условная конструкция. Она имеет вид `(if test then [else])` , `then` `test else (if test then [else])` и оценивается на `then` , если `test` верно и `else` иначе. Часть else может быть опущена.

```
(if (> 3 2)
    "Three is bigger!"
    "Two is bigger!")
;;=> "Three is bigger!"
```

Одно очень важное различие между `if` в Common Lisp и, `if` во многих других языках программирования, это CL, `if` это выражение, а не утверждение. Таким образом, `if` формы возвращают значения, которые могут быть назначены переменным, используемые в списках аргументов и т. Д .:

```
;; Use a different format string depending on the type of x
(format t (if (numberp x)
              "~x~%"
              "~a~%")
          x)
```

Общий Lisp, `if` можно считать эквивалентным тернарному оператору?: В C # и других языках «фигурных скобок».

Например, следующее выражение C #:

```
year == 1990 ? "Hammertime" : "Not Hammertime"
```

Является эквивалентом следующего кода Common Lisp, считая, что `year` содержит целое число:

```
(if (eql year 1990) "Hammertime" "Not Hammertime")
```

`cond` - еще одна условная конструкция. Он несколько похож на цепочку операторов `if` и имеет вид:

```
(cond (test-1 consequent-1-1 consequent-2-1 ...)
      (test-2)
      (test-3 consequent-3-1 ...)
      ... )
```

Точнее, `cond` имеет нулевые или более *предложения* , и каждое предложение имеет один тест, за которым следуют ноль или более последовательных. Вся конструкция `cond` выбирает первое предложение, тест которого не оценивается в `nil` и оценивает его последующие порядки. Он возвращает значение последней формы в последующих.

```
(cond ((> 3 4) "Three is bigger than four!")
      ((> 3 3) "Three is bigger than three!")
      ((> 3 2) "Three is bigger than two!")
      ((> 3 1) "Three is bigger than one!"))
;;=> "Three is bigger than two!"
```

Чтобы предоставить предложение по умолчанию, чтобы оценить, не оценивает ли другое предложение значение `t` , вы можете добавить предложение, которое истинно по умолчанию, используя `t` . Это очень похоже на концепцию SQL `CASE...ELSE` , но для выполнения задачи используется буквальное логическое значение true, а не ключевое слово.

```
(cond
    ((= n 1) "N equals 1")
    (t "N doesn't equal 1")
)
```

Конструкция `if` может быть записана как `cond` конструкция. `(if test then else)` и `(cond (test then) (t else))` эквивалентны.

Если вам нужно только одно предложение, используйте, `when` или `unless` :

```
(when (> 3 4)
  "Three is bigger than four.")
;;=> NIL

(when (< 2 5)
  "Two is smaller than five.")
;;=> "Two is smaller than five."

(unless (> 3 4)
  "Three is bigger than four.")
;;=> "Three is bigger than four."

(unless (< 2 5)
  "Two is smaller than five.")
;;=> NIL
```

Цикл do

Большинство циклов и условных конструкций в Common Lisp на самом деле являются макросами, которые скрывают более основные конструкции. Например, `dotimes` и `dolist` построены на `do` макрос. Форма для `do` выглядит так:

```
(do (varlist)
```

```
    (endlist)
   &body)
```

- `varlist` состоит из переменных, определенных в цикле, их начальных значений и того, как они меняются после каждой итерации. Часть «изменения» оценивается в конце цикла.
- `endlist` содержит конечные условия и значения, возвращаемые в конце цикла. Конечное условие оценивается в начале цикла.

Вот один, который начинается с 0 и идет вверх (не включая) 10.

```
;;same as (dotimes (i 10))
(do (( i (+ 1 i))
    ((< i 10) i)
   (print i))
```

И вот тот, который перемещается по списку:

```
;;same as (dolist (item given-list)
(do ((item (car given-list))
     (temp list (cdr temp))
   (print item))
```

Часть `varlist` похожа на `varlist` в инструкции `let` . Вы можете связывать более одной переменной, и они существуют только внутри цикла. Каждая объявленная переменная находится в своем собственном наборе скобок. Вот один из них, который подсчитывает количество 1 и 2 в списке.

```
(let ((vars (list 1 2 3 2 2 1)))
  (do ((ones 0)
       (twos 0)
       (temp vars (cdr temp)))
      ((not temp) (list ones twos))
    (when (= (car temp) 1)
      (setf ones (+ 1 ones)))
    (when (= (car temp) 2)
      (setf twos (+ 1 twos)))))
-> (2 3)
```

И если макрос цикла while не был реализован:

```
(do ()
    (t)
  (when task-done
    (break)))
```

Для наиболее распространенных приложений более конкретные `dotimes` и `doloop` макросы гораздо более кратки.

Прочитайте Контрольные структуры онлайн: https://riptutorial.com/ru/common-lisp/topic/3229/

контрольные-структуры

глава 11: котировка

Синтаксис

- (объект цитаты) -> объект

замечания

Есть некоторые объекты (например, символы ключевых слов), которые не нужно указывать, поскольку они сами оценивают.

Examples

Пример простой цитаты

Цитата - это **специальный оператор,** который предотвращает оценку его аргумента. Он возвращает свой аргумент, неоценимый.

```
CL-USER> (quote a)
A

CL-USER> (let ((a 3))
           (quote a))
A
```

'является псевдонимом для специального оператора QUOTE

Обозначение `'thing` равно `(quote thing)` .

Читатель будет делать расширение:

```
> (read-from-string "'a")
(QUOTE A)
```

Цитирование используется для предотвращения дальнейшей оценки. Выделенный объект оценивается сам по себе.

```
> 'a
A

> (eval '+ 1 2)
3
```

Если цитируемые объекты деструктивно изменены, последствия не

определены!

Избегайте разрушительных операций с цитируемыми объектами. Объекты с котировкой - это литеральные объекты. Возможно, они каким-то образом встроены в код. Как это работает, и последствия изменений не указаны в стандарте Common Lisp, но могут иметь нежелательные последствия, такие как изменение общих данных, попытка изменить данные, защищенные от записи, или создание непреднамеренных побочных эффектов.

~~(delete 5 '(1 2 3 4 5))~~

Цитата и самооценка объектов

Обратите внимание, что многие типы данных не нужно указывать, так как они сами оценивают. `QUOTE` особенно полезна для символов и списков, чтобы предотвратить оценку в виде форм Лиспа.

Пример для других типов данных, которые не нужно указывать для предотвращения оценки: строки, числа, символы, объекты CLOS, ...

Вот пример для строк. Результаты оценки - это строки, независимо от того, указаны они в источнике или нет.

```
> (let ((some-string-1 "this is a string")
        (some-string-2 '"this is a string with a quote in the source")
        (some-string-3 (quote "this is another string with a quote in the source")))
    (list some-string-1 some-string-2 some-string-3))

("this is a string"
 "this is a string with a quote in the source"
 "this is another string with a quote in the source")
```

Таким образом, предложение для объектов необязательно.

Прочитайте котировка онлайн: https://riptutorial.com/ru/common-lisp/topic/1315/котировка

глава 12: Лексические и специальные переменные

Examples

Глобальные специальные переменные везде

Таким образом, эти переменные будут использовать динамическое связывание.

```
(defparameter count 0)
;; All uses of count will refer to this one

(defun handle-number (number)
  (incf count)
  (format t "~&~d~%" number))

(dotimes (count 4)
  ;; count is shadowed, but still special
  (handle-number count))

(format t "~&Calls: ~d~%" count)
==>
0
2
Calls: 0
```

Дайте специальным переменным разные имена, чтобы избежать этой проблемы:

```
(defparameter *count* 0)

(defun handle-number (number)
  (incf *count*)
  (format t "~&~d~%" number))

(dotimes (count 4)
  (handle-number count))

(format t "~&Calls: ~d~%" *count*)
==>
0
1
2
3
Calls: 4
```

Примечание 1: невозможно сделать глобальную переменную неспециальной в определенной области. Нет объявления, чтобы сделать переменную *лексической* .

Примечание 2: можно объявить переменную *специальный* в местном контексте , используя `special` декларацию. Если глобальная специальная декларация для этой переменной

отсутствует, декларация будет только локальной и может быть затенена.

```
(defun bar ()
  (declare (special a))
  a)                        ; value of A is looked up from the dynamic binding

(defun foo ()
  (let ((a 42))             ; <- this variable A is special and
                            ;    dynamically bound
    (declare (special a))
    (list (bar)
          (let ((a 0))      ; <- this variable A is lexical
            (bar)))))


> (foo)
(42 42)
```

Прочитайте Лексические и специальные переменные онлайн:
https://riptutorial.com/ru/common-lisp/topic/3362/лексические-и-специальные-переменные

глава 13: Логические и обобщенные булевы

Examples

Правда и ложь

Специальный символ `T` представляет значение *true* в Common Lisp, а специальный символ `NIL` представляет *false* :

```
CL-USER> (= 3 3)
T
CL-USER> (= 3 4)
NIL
```

Они называются «константными переменными» (sic!) В стандарте, поскольку они являются переменными, значение *которых не может* быть изменено. Как следствие, вы не можете использовать свои имена для обычных переменных, например, в следующем, неверном, например:

```
CL-USER> (defun my-fun(t)
           (+ t 1))
While compiling MY-FUN :
Can't bind or assign to constant T.
```

Собственно, их можно рассматривать просто как константы или как самооцененные символы. `T` и `NIL` - это специальные вещи в других смыслах. Например, `T` также является типом (супертипом любого другого типа), тогда как `NIL` также является пустым списком:

```
CL-USER> (eql NIL '())
T
CL-USER> (cons 'a (cons 'b nil))
(A B)
```

Обобщенные булевы

Фактически любое значение, отличное от `NIL` , считается *истинным* значением в Common Lisp. Например:

```
CL-USER> (let ((a (+ 2 2)))
           (if a
               a
               "Oh my! 2 + 2 is equal to NIL!"))
4
```

Этот факт можно комбинировать с булевыми операторами, чтобы сделать программы более краткими. Например, приведенный выше пример эквивалентен:

```
CL-USER> (or (+ 2 2) "Oh my! 2 + 2 is equal to NIL!")
4
```

Макрос `OR` оценивает свои аргументы в порядке слева направо и останавливается, как только он находит значение, отличное от NIL, и возвращает его. Если все они являются `NIL` , возвращаемое значение равно `NIL` :

```
CL-USER> (or (= 1 2) (= 3 4) (= 5 6))
NIL
```

Аналогично, макрос `AND` оценивает свои аргументы слева направо и возвращает значение последнего, если все они оцениваются не-NIL, в противном случае останавливает оценку, как только находит `NIL` , возвращая его:

```
CL-USER> (let ((a 2)
               (b 3))
           (and (/= b 0) (/ a b)))
2/3
CL-USER> (let ((a 2)
               (b 0))
           (and (/= b 0) (/ a b)))
NIL
```

По этим причинам `AND` и `OR` можно считать более похожими на управляющие структуры других языков, а не на логические операторы.

Прочитайте Логические и обобщенные булевы онлайн: https://riptutorial.com/ru/common-lisp/topic/3292/логические-и-обобщенные-булевы

глава 14: макрос

замечания

Цель макросов

Макросы предназначены для генерации кода, преобразования кода и предоставления новых обозначений. Эти новые обозначения могут быть более подходящими для лучшего выражения программы, например, путем создания конструкций на уровне домена или целых новых внедренных языков.

Макросы могут сделать исходный код более понятным, но отладка может быть затруднена. Как правило, нельзя использовать макросы, когда будет выполняться регулярная функция. Когда вы их используете, избегайте обычных ловушек, старайтесь придерживаться обычно используемых шаблонов и соглашений об именах.

Заказ Macroexpansion

По сравнению с функциями макросы расширяются в обратном порядке; крайний крайний, самый последний. Это означает, что по умолчанию нельзя использовать внутренний макрос для генерации синтаксиса, необходимого для внешнего макроса.

Порядок оценки

Иногда макросы должны перемещать пользовательские формы. Нужно следить за тем, чтобы не изменять порядок их оценки. Пользователь может полагаться на побочные эффекты, происходящие по порядку.

Оценить только один раз

Расширение макроса часто требует использования значения одной и той же пользовательской формы более одного раза. Возможно, что форма имеет побочные эффекты или может вызвать дорогостоящую функцию. Таким образом, макрос должен обязательно оценивать только такие формы один раз. Обычно это делается путем назначения значения локальной переменной (имя которой `GENSYM` ed).

Функции, используемые макросами, используя EVAL-WHEN

Сложные макросы часто имеют части своей логики, реализованные в отдельных функциях. Однако следует помнить, что макросы расширяются до того, как будет скомпилирован фактический код. При компиляции файла по умолчанию функции и переменные, определенные в одном файле, не будут доступны во время выполнения макроса. Все определения функций и переменных в том же файле, которые используются макросом, должны быть обернуты внутри формы `EVAL-WHEN` . `EVAL-WHEN` должно быть указано все три раза, когда прилагаемый код также должен быть оценен во время загрузки и времени выполнения.

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (defun foobar () ...))
```

Это не относится к функциям, вызванным расширением макроса, только те, которые вызываются самим макросом.

Examples

Общие шаблоны макросов

TODO: Возможно, переместите пояснения к замечаниям и добавьте примеры отдельно

FOOF

В Common Lisp существует концепция обобщенных ссылок . Они позволяют программисту устанавливать значения в разные «места», как если бы они были переменными. Макросы, которые используют эту способность, часто имеют имя `F` -postfix. Обычно это первый аргумент макроса.

Примеры из стандарта: `INCF` , `DECF` , `ROTATEF` , `SHIFTF` , `REMF` .

Глупый пример, макрос, который переворачивает знак магазина чисел в месте:

```
(defmacro flipf (place)
  `(setf ,place (- ,place)))
```

C-FOO

Макросы, которые приобретают и безопасно выпускают ресурс, обычно называются `WITH-` -prefix. Обычно макрос должен использовать синтаксис, например:

```
(with-foo (variable details-of-the-foo...)
  body...)
```

Примеры из стандарта: `WITH-OPEN-FILE` , `WITH-OPEN-STREAM` , `WITH-INPUT-FROM-STRING` , `WITH-OUTPUT-TO-STRING` .

Один из подходов к реализации этого типа макросов, который может избежать некоторых ошибок в загрязнении имен и непреднамеренной множественной оценки, заключается в том, что сначала внедряется функциональная версия. Например, первый шаг в реализации макроса `with-widget` который безопасно создает виджет и очищает после него, может быть функцией:

```
(defun call-with-widget (args function)
  (let ((widget (apply #'make-widget args))) ; obtain WIDGET
    (unwind-protect (funcall function widget)  ; call FUNCTION with WIDGET
      (cleanup widget)                         ; cleanup
```

Поскольку это функция, нет никаких проблем в отношении объема имен внутри **функции** или **поставщика** , и это упрощает запись соответствующего макроса:

```
(defmacro with-widget ((var &rest args) &body body)
  `(call-with-widget (list ,@args) (lambda (,var) ,@body)))
```

DO-FOO

Макросы, которые перебирают что-то, часто называются `DO` -prefix. Макро синтаксис обычно должен быть в форме

```
(do-foo (variable the-foo-being-done return-value)
  body...)
```

Примеры из стандарта: `DOTIMES` , `DOLIST` , `DO-SYMBOLS` .

ЛОКАЗА, ЭФОКАЗА, CFOOCASE

Макросы, соответствующие входным данным в некоторых случаях, часто называются `CASE` -postfix. Часто существует `E...CASE` -вариант, который сигнализирует об ошибке, если вход не соответствует ни одному из случаев, и `C...CASE` , который сигнализирует о продолжающейся ошибке. Они должны иметь такой синтаксис, как

```
(foocase input
```

```
  (case-to-match-against (optionally-some-params-for-the-case)
   case-body-forms...)
  more-cases...
  [(otherwise otherwise-body)])
```

Примеры из стандарта: `CASE` , `TYPECASE` , `TYPECASE HANDLER-CASE` .

Например, макрос, который сопоставляет строку с регулярными выражениями и связывает группы регистров с переменными. Использует CL-PPCRE для регулярных выражений.

```
(defmacro regexcase (input &body cases)
  (let ((block-sym (gensym "block"))
        (input-sym (gensym "input")))
    `(let ((,input-sym ,input))
       (block ,block-sym
         ,@(loop for (regex vars . body) in cases
                 if (eql regex 'otherwise)
                   collect `(return-from ,block-sym (progn ,vars ,@body))
                 else
                   collect `(cl-ppcre:register-groups-bind ,vars
                                (,regex ,input-sym)
                              (return-from ,block-sym
                                (progn ,@body)))))))))

(defun test (input)
  (regexcase input
    ("(\\d+)-(\\d+)" (foo bar)
      (format t "Foo: ~a, Bar: ~a~%" foo bar))
    ("Foo: (\\w+)$" (foo)
      (format t "Foo: ~a.~%" foo))
    (otherwise (format t "Didn't match.~%"))))

(test "asd 23-234 qwe")
; Foo: 23, Bar: 234
(test "Foo: Foobar")
; Foo: Foobar.
(test "Foo: 43 - 23")
; Didn't match.
```

DEFINE-FOO, DEFFOO

Макросы, которые определяют вещи, обычно называются либо `DEFINE-` либо `DEF` -prefix.

Примеры из стандарта: `DEFUN` , `DEFMACRO` , `DEFINE-CONDITION` .

Анафорические макросы

Anaphoric Macro - это макрос, который вводит переменную (часто `IT`), которая захватывает результат предоставленной пользователем формы. Общим примером является Anaphhoric If, который похож на обычный `IF` , но также определяет переменную `IT` для ссылки на результат тестовой формы.

```
(defmacro aif (test-form then-form &optional else-form)
  `(let ((it ,test-form))
     (if it ,then-form ,else-form)))

(defun test (property plist)
  (aif (getf plist property)
       (format t "The value of ~s is ~a.~%" property it)
       (format t "~s wasn't in ~s!~%" property plist)))

(test :a '(:a 10 :b 20 :c 30))
; The value of :A is 10.
(test :d '(:a 10 :b 20 :c 30))
; :D wasn't in (:A 10 :B 20 :C 30)!
```

MACROEXPAND

Расширение макросов - это процесс превращения макросов в фактический код. Обычно это происходит как часть процесса компиляции. Компилятор будет расширять все макроформы до фактического компиляции кода. Макро расширение также происходит во время *интерпретации* кода Lisp.

Можно вручную вызвать `MACROEXPAND` чтобы узнать, к чему расширяется `MACROEXPAND` .

```
CL-USER> (macroexpand '(with-open-file (file "foo")
                        (do-something-with file)))
(LET ((FILE (OPEN "foo")) (#:G725 T))
  (UNWIND-PROTECT
      (MULTIPLE-VALUE-PROG1 (PROGN (DO-SOMETHING-WITH FILE)) (SETQ #:G725 NIL))
    (WHEN FILE (CLOSE FILE :ABORT #:G725))))
```

`MACROEXPAND-1` тот же, но только один раз расширяется. Это полезно при попытке понять макроскопическую форму, которая расширяется до другой макроформы.

```
CL-USER> (macroexpand-1 '(with-open-file (file "foo")
                          (do-something-with file)))
(WITH-OPEN-STREAM (FILE (OPEN "foo")) (DO-SOMETHING-WITH FILE))
```

Обратите внимание, что ни `MACROEXPAND` ни `MACROEXPAND-1` расширяют код Lisp на всех уровнях. Они только расширяют макрос верхнего уровня. Для макроэкспонирования полной формы на всех уровнях для этого нужен *ходок кода* . Это средство не предусмотрено стандартом Common Lisp.

Backquote - создание шаблонов кода для макросов

Код возврата макросов. Поскольку код в Lisp состоит из списков, для его создания можно использовать обычные функции управления списком.

```
;; A pointless macro
(defmacro echo (form)
  (list 'progn
        (list 'format t "Form: ~a~%" (list 'quote form))
```

```
        form))
```

Это часто очень трудно читать, особенно в более длинных макросах. Макрос чтения Backquote позволяет писать шаблоны с кавычками , которые заполняются выборочной оценкой элементов.

```
(defmacro echo (form)
  `(progn
     (format t "Form: ~a~%" ',form)
     ,form))

(macroexpand '(echo (+ 3 4)))
;=> (PROGN (FORMAT T "Form: ~a~%" '(+ 3 4)) (+ 3 4))
```

Эта версия выглядит почти как обычный код. Запятые используются для оценки `FORM` ; все остальное возвращается как есть. Обратите внимание, что в `',form` одинарной кавычки находится за запятой, поэтому она будет возвращена.

Можно также использовать `,@` для сращивания списка в позиции.

```
(defmacro echo (&rest forms)
  `(progn
     ,@(loop for form in forms collect `(format t "Form: ~a~%" ,form))
     ,@forms))

(macroexpand '(echo (+ 3 4)
                    (print "foo")
                    (random 10)))
;=> (PROGN
;     (FORMAT T "Form: ~a~%" (+ 3 4))
;     (FORMAT T "Form: ~a~%" (PRINT "foo"))
;     (FORMAT T "Form: ~a~%" (RANDOM 10))
;     (+ 3 4)
;     (PRINT "foo")
;     (RANDOM 10))
```

Backquote можно использовать и вне макросов.

Уникальные символы для предотвращения конфликтов имен в макросах

Расширение макроса часто требует использования символов, которые не были переданы в качестве аргументов пользователем (например, имена локальных переменных). Нужно убедиться, что такие символы не могут конфликтовать с символом, который пользователь использует в окружающем коде.

Обычно это достигается с помощью функции `GENSYM` , которая возвращает новый символ без символа.

Плохой

Рассмотрим макрос ниже. Он делает `DOTIMES` -loop, который также собирает результат тела

в список, который возвращается в конце.

```
(defmacro dotimes+collect ((var count) &body body)
  `(let ((result (list)))
     (dotimes (,var ,count (nreverse result))
       (push (progn ,@body) result))))

(dotimes+collect (i 5)
  (format t "~a~%" i)
  (* i i))
; 0
; 1
; 2
; 3
; 4
;=> (0 1 4 9 16)
```

Кажется, что это работает в этом случае, но если у пользователя произошло переменное имя `RESULT` , которое они используют в теле, результаты, вероятно, не будут тем, что ожидает пользователь. Рассмотрим эту попытку написать функцию, которая собирает список сумм всех целых чисел до `N` :

```
(defun sums-upto (n)
  (let ((result 0))
    (dotimes+collect (i n)
      (incf result i))))

(sums-upto 10) ;=> Error!
```

Хорошо

Чтобы решить эту проблему, мы должны использовать `GENSYM` для создания уникального имени для `RESULT` -переменного в макроподстановках.

```
(defmacro dotimes+collect ((var count) &body body)
  (let ((result-symbol (gensym "RESULT")))
    `(let ((,result-symbol (list)))
       (dotimes (,var ,count (nreverse ,result-symbol))
         (push (progn ,@body) ,result-symbol)))))

(sums-upto 10) ;=> (0 1 3 6 10 15 21 28 36 45)
```

TODO: Как сделать символы из строк

TODO: устранение проблем с символами в разных пакетах

если-let, когда-let, -let-макросы

Эти макросы объединяют поток управления и связывание. Это улучшение по сравнению с анафорическими анафорическими макросами, потому что они позволяют разработчику передавать смысл посредством именования. Поэтому их использование рекомендуется по сравнению с анафорическими аналогами.

```
(if-let (user (get-user user-id))
  (show-dashboard user)
  (redirect 'login-page))
```

`FOO-LET` связывают одну или несколько переменных, а затем используют эти переменные в качестве тестовой формы для соответствующего условного (`IF` , `WHEN`). Множественные переменные объединяются с `AND` . Выбранная ветвь выполняется с действующими связями. Простая реализация переменной `IF-LET` одной переменной может выглядеть примерно так:

```
(defmacro if-let ((var test-form) then-form &optional else-form)
  `(let ((,var ,test-form))
     (if ,var ,then-form ,else-form)))

(macroexpand '(if-let (a (getf '(:a 10 :b 20 :c 30) :a))
                (format t "A: ~a~%" a)
                (format t "Not found.~%")))
; (LET ((A (GETF '(:A 10 :B 20 :C 30) :A)))
;   (IF A
;       (FORMAT T "A: ~a~%" A)
;       (FORMAT T "Not found.~%")))
```

Версия, поддерживающая несколько переменных, доступна в библиотеке Александрии .

Использование макросов для определения структур данных

Общее использование макросов заключается в создании шаблонов для структур данных, которые подчиняются общим правилам, но могут содержать разные поля. Записывая макрос, вы можете разрешить детальное конфигурирование структуры данных без необходимости повторять шаблонный код и не использовать менее эффективную структуру (например, хеш) в памяти, чтобы упростить программирование.

Например, предположим, что мы хотим определить несколько классов, которые имеют ряд разных свойств, каждый из которых имеет геттер и сеттер. Кроме того, для некоторых (но не всех) этих свойств мы хотим, чтобы setter вызывал метод на объекте, уведомляющем его о том, что свойство было изменено. Хотя Common LISP уже имеет сокращение для написания геттеров и сеттеров, написание стандартного настраиваемого сеттера таким образом обычно требует дублирования кода, который вызывает метод уведомления в каждом сетевом устройстве, что может быть больно, если имеется большое количество свойств , Однако, определяя макрос, это становится намного проще:

```
(defmacro notifier (class slot)
  "Defines a setf method in (class) for (slot) which calls the object's changed method."
   `(defmethod (setf ,slot) (val (item ,class))
     (setf (slot-value item ',slot) val)
     (changed item ',slot)))

(defmacro notifiers (class slots)
  "Defines setf methods in (class) for all of (slots) which call the object's changed method."
  `(progn
     ,@(loop for s in slots collecting `(notifier ,class ,s))))
```

```
(defmacro defclass-notifier-slots (class nslots slots)
  "Defines a class with (nslots) giving a list of slots created with notifiers, and (slots)
giving a list of slots created with regular accessors."
  `(progn
     (defclass ,class ()
       ( ,@(loop for s in nslots collecting `(,s :reader ,s))
         ,@(loop for s in slots collecting `(,s :accessor ,s))))
     (notifiers ,class ,nslots)))
```

Теперь мы можем написать `(defclass-notifier-slots foo (bar baz qux) (waldo))` и сразу определить класс `foo` с обычным слотом `waldo` (созданным второй частью макроса со спецификацией `(waldo :accessor waldo)`) , и слоты `bar` , `baz` и `qux` с сеттерами, которые вызывают `changed` метод (где геттер определяется первой частью макроса `(bar :reader bar)` и сеттер с помощью макроса вызываемого `notifier`).

В дополнение к тому, что мы можем быстро определить несколько классов, которые ведут себя таким образом, с большим количеством свойств без повторения, мы получаем обычное преимущество повторного использования кода: если позже мы решим изменить способ работы методов уведомления, мы можем просто изменить макрос, и структура каждого класса, использующего его, изменится.

Прочитайте макрос онлайн: https://riptutorial.com/ru/common-lisp/topic/1257/макрос

глава 15: настройка

Examples

Дополнительные возможности для Read-Eval-Print-Loop (REPL) в терминале

CLISP имеет интеграцию с GNU Readline.

Дополнительные улучшения для других реализаций см. В разделе: Как настроить SBCL REPL .

Файлы инициализации

Большинство распространенных реализаций Lisp попытаются загрузить *файл инициализации* при запуске:

Реализация	Исходный файл	Файл Site / System Init
ABCL	`$HOME/.abclrc`	
Allegro CL	`$HOME/.clinit.cl`	
ECL	`$HOME/.eclrc`	
пожатие	`$HOME/.clasprc`	
CLISP	`$HOME/.clisprc.lisp`	
Clozure CL	`home:ccl-init.lisp` **ИЛИ** `home:ccl-init.fasl` **ИЛИ** `home:.ccl-init.lisp`	
CMUCL	`$HOME/.cmucl-init.lisp`	
LispWorks	`$HOME/.lispworks`	
MKCL	`$HOME/.mkclrc`	
SBCL	`$HOME/.sbclrc`	`$SBCL_HOME/sbclrc` **ИЛИ** `/etc/sbclrc`
SCL	`$HOME/.scl-init.lisp`	

Примеры файлов инициализации:

Реализация	Пример файла Init
LispWorks	`Library/lib/7-0-0-0/config/a-dot-lispworks.lisp`

Настройки оптимизации

Common Lisp имеет способ влиять на стратегии компиляции. Имеет смысл определить ваши предпочтительные значения.

Значения оптимизации находятся между 0 (несущественными) и 3 (чрезвычайно важными). **1 - нейтральное значение.**

Полезно всегда использовать безопасный код (безопасность = 3) с включенными проверками времени выполнения.

Обратите внимание, что интерпретация значений является специфичной для реализации. Большинство распространенных реализаций Lisp используют некоторые значения этих значений.

настройка	объяснение	полезное значение по умолчанию	полезная стоимость доставки
`compilation-speed`	скорость процесса компиляции	2	0
`debug`	легкость отладки	2	1 или 0
`safety`	проверка ошибок во время выполнения	3	2
`space`	как размер кода, так и время выполнения	2	2
`speed`	скорость объектного кода	2	3

`optimize` объявление для использования с `declaim` , `declare` и `proclaim` :

```
(optimize (compilation-speed 2)
          (debug 2)
          (safety 3)
          (space 2)
          (speed 2))
```

Обратите внимание, что вы также можете применять специальные настройки оптимизации к частям кода в функции с помощью макроса `LOCALLY` .

Прочитайте настройка онлайн: https://riptutorial.com/ru/common-lisp/topic/5679/настройка

глава 16: Основные петли

Синтаксис

- (do ({var | (var [init-form [step-form]])} *) (end-test-form result-form *) Объявление * {tag | statement} *)
- (do * ({var | (var [init-form [step-form]])} *) (end-test-form result-form *) Объявление * {tag | statement} *)
- (объявление dolist (var list-form [result-form]) * {tag | statement} *)
- (dotimes (var count-form [result-form]) декларация * {tag | statement} *)

Examples

DOTIMES

`dotimes` - это макрос для целочисленной итерации по одной переменной от 0 ниже некоторого значения параметра. Один из примеров:

```
CL-USER> (dotimes (i 5)
           (print i))

0
1
2
3
4
NIL
```

Обратите внимание, что `NIL` - это возвращаемое значение, так как мы сами не предоставили его; переменная начинается с 0 и по всему циклу становится значениями от 0 до N-1. После цикла переменная становится N:

```
CL-USER> (dotimes (i 5 i))
5

CL-USER> (defun 0-to-n (n)
           (let ((list ()))
             (dotimes (i n (nreverse list))
               (push i list))))
0-TO-N
CL-USER> (0-to-n 5)
(0 1 2 3 4)
```

DOLIST

`dolist` - это `dolist` макрос, созданный для простой `dolist` списков. Одним из самых простых применений было бы:

```
CL-USER> (dolist (item '(a b c d))
           (print item))

A
B
C
D
NIL ; returned value is NIL
```

Обратите внимание, что поскольку мы не предоставили возвращаемое значение, возвращается `NIL` (a A, B, C, D печатаются на `*standard-output*`).

`dolist` также может возвращать значения:

```
;;This may not be the most readable summing function.
(defun sum-list (list)
  (let ((sum 0))
    (dolist (var list sum)
      (incf sum var))))

CL-USER> (sum-list (list 2 3 4))
9
```

Простая петля

Макрос **цикла** имеет две формы: «простую» форму и «расширенную» форму. Расширенная форма рассматривается в другом разделе документации, но простой цикл полезен для очень простого цикла.

Простая форма **цикла** принимает несколько форм и повторяет их до тех пор, пока цикл не будет завершен с использованием **возврата** или какого-либо другого выхода (например, **throw**).

```
(let ((x 0))
  (loop
     (print x)
     (incf x)
     (unless (< x 5)
       (return))))

0
1
2
3
4
NIL
```

Прочитайте Основные петли онлайн: https://riptutorial.com/ru/common-lisp/topic/2053/основные-петли

глава 17: Отображение функций по спискам

Examples

обзор

Набор функций отображения высокого уровня доступен в Common Lisp, чтобы применить функцию к элементам одного или нескольких списков. Они отличаются тем, как функция применяется к спискам и как получается конечный результат. Следующая таблица суммирует различия и показывает для каждой из них эквивалентную форму LOOP. *f* - применяемая функция, которая должна иметь число аргументов, равное количеству списков; «Применяется к машине» означает, что оно применяется в свою очередь к элементам списков, «применяется к cdr» означает, что оно применяется в свою очередь к спискам, их cdr, их cddr и т. Д .; столбец «возвращает» показывает, является ли глобальный результат полученным путем перечисления результатов, объединяя их (поэтому они должны быть списками!) или просто используются для побочных эффектов (и в этом случае возвращается первый список).

функция	Применительно к	Возвращает	Эквивалентный LOOP
(mapcar fl $_1$ ... l $_n$)	автомобиль	список результатов	(цикл для x $_1$ в l $_1$ ... для x $_n$ в l $_n$ собирать (fx $_1$ ... x $_n$))
(maplist fl $_1$ ... l $_n$)	корд	список результатов	(цикл для x $_1$ на l $_1$ ... для x $_n$ на l $_n$ собирать (fx $_1$ ... x $_n$))
(mapcan fl $_1$ ... l $_n$)	автомобиль	конкатенация результатов	(цикл для x $_1$ в l $_1$ ... для x $_n$ в l $_n$ nconc (fx $_1$ ... x $_n$))
(mapcon fl $_1$ ... l $_n$)	корд	конкатенация результатов	(цикл для x $_1$ на l $_1$ ... для x $_n$ на l $_n$ nconc (fx $_1$ ... x $_n$))
(mapc fl $_1$ ... l $_n$)	автомобиль	л $_1$	(цикл для x $_1$ в l $_1$ ... для x $_n$ в l $_n$ do (fx $_1$ ... x $_n$) наконец (return l $_1$))
(mapl fl $_1$ ... l $_n$)	корд	л $_1$	(цикл для x $_1$ на l $_1$ ... для x $_n$ на l $_n$ do (fx $_1$ ... x $_n$) наконец (return l $_1$))

Обратите внимание, что во всех случаях списки могут иметь разную длину, и приложение заканчивается при завершении кратчайшего списка.

Доступна еще одна пара функций карты: `map` , которая может быть применена к последовательностям (строки, векторы, списки), аналогичные `mapcar` , и которые могут возвращать любой тип последовательности, указанный как первый аргумент, и `map-into` , аналогичный `map` , но это разрушительно изменяет свой первый аргумент последовательности, чтобы сохранить результаты применения функции.

Примеры MAPCAR

MAPCAR является наиболее часто используемой функцией семьи:

```
CL-USER> (mapcar #'1+ '(1 2 3))
(2 3 4)
CL-USER> (mapcar #'cons '(1 2 3) '(a b c))
((1 . A) (2 . B) (3 . C))
CL-USER> (mapcar (lambda (x y z) (+ (* x y) z))
                 '(1 2 3)
                 '(10 20 30)
                 '(100 200 300))
(110 240 390)
CL-USER> (let ((list '(a b c d e f g h i))) ; randomize this list
           (mapcar #'cdr
                   (sort (mapcar (lambda (x)
                                   (cons (random 100) x))
                                 list)
                         #'<=
                         :key #'car)))
(I D A G B H E C F)
```

`mapcar` использование `mapcar` заключается в транспонировании матрицы, представленной в виде списка списков:

```
CL-USER> (defun transpose (list-of-lists)
           (apply #'mapcar #'list list-of-lists))
ROTATE
CL-USER> (transpose '((a b c) (d e f) (g h i)))
((A D G) (B E H) (C F I))

;  +---+---+---+              +---+---+---+
;  | A | B | C |              | A | D | G |
;  +---+---+---+              +---+---+---+
;  | D | E | F |    becomes   | B | E | H |
;  +---+---+---+              +---+---+---+
;  | G | H | I |              | C | F | I |
;  +---+---+---+              +---+---+---+
```

Для пояснения см. Этот ответ .

Примеры MAPLIST

```
CL-USER> (maplist (lambda (list) (cons 0 list)) '(1 2 3 4))
((0 1 2 3 4) (0 2 3 4) (0 3 4) (0 4))
CL-USER> (maplist #'append
                  '(a b c d -)
                  '(1 2 3))
((A B C D - 1 2 3) (B C D - 2 3) (C D - 3))
```

Примеры MAPCAN и MAPCON

MAPCAN:

```
CL-USER> (mapcan #'reverse '((1 2 3) (a b c) (100 200 300)))
(3 2 1 C B A 300 200 100)
CL-USER> (defun from-to (min max)
           (loop for i from min to max collect i))
FROM-TO
CL-USER> (from-to 1 5)
(1 2 3 4 5)
CL-USER> (mapcan #'from-to '(1 2 3) '(5 5 5))
(1 2 3 4 5 2 3 4 5 3 4 5)
```

Одним из видов использования MAPCAN является создание списка результатов без значений NIL:

```
CL-USER> (let ((l1 '(10 20 40)))
           (mapcan (lambda (x)
                     (if (member x l1)
                         (list x)
                       nil))
                   '(2 4 6 8 10 12 14 16 18 20
                     18 16 14 12 10 8 6 4 2)))
(10 20 10)
```

MAPCON:

```
CL-USER> (mapcon #'copy-list '(1 2 3))
(1 2 3 2 3 3)
CL-USER> (mapcon (lambda (l1 l2) (list (length l1) (length l2))) '(a b c d) '(d e f))
(4 3 3 2 2 1)
```

Примеры MAPC и MAPL

MAPC:

```
CL-USER> (mapc (lambda (x) (print (* x x))) '(1 2 3 4))

1
4
9
16
(1 2 3 4)
CL-USER> (let ((sum 0))
           (mapc (lambda (x y) (incf sum (* x y)))
```

```
                '(1 2 3)
                '(100 200 300))
          sum)
1400  ; => (1 x 100) + (2 x 200) + (3 x 300)
```

MAPL:

```
CL-USER> (mapl (lambda (list) (print (reduce #'+ list))) '(1 2 3 4 5))

15
14
12
9
5
(1 2 3 4 5)
```

Прочитайте Отображение функций по спискам онлайн: https://riptutorial.com/ru/common-lisp/topic/6064/отображение-функций-по-спискам

глава 18: последовательность - как разбить последовательность

Синтаксис

1. split regex target-string & end start end limit with-registers-p omit-unmatched-p sharedp => list
2. lispworks: последовательность разделителей-разделителей с разделительной последовательностью и ключ-ключ окончания ключа для коалесценции-разделителей => последовательности
3. последовательность разграничения последовательности и конец запуска ключа из конца списка remove-empty-subseqs test test-not key => список подпоследовательностей

Examples

Разделить строки с использованием регулярных выражений

Библиотека CL-PPCRE предоставляет функцию `split` которая позволяет нам разделить строки в подстроках, которые соответствуют регулярному выражению, отбрасывая части строки, которые этого не делают.

```
(cl-ppcre:split "\\." "127.0.0.1")
;; => ("127" "0" "0" "1")
```

SPLIT-SEQUENCE в LIspWorks

Простое разделение строки IP-номера.

```
> (lispworks:split-sequence "." "127.0.0.1")
  ("127" "0" "0" "1")
```

Простое разделение URL-адреса:

```
> (lispworks:split-sequence ".:/" "http://127.0.0.1/foo/bar.html"
                            :coalesce-separators t)
("http" "127" "0" "0" "1" "foo" "bar" "html")
```

Использование библиотеки split-sequence

Библиотека split-sequence предоставляет функцию `split-sequence` , которая позволяет разделить на элементы последовательности

```
(split-sequence:split-sequence #\Space "John Doe II")
;; => ("John" "Doe" "II")
```

Прочитайте последовательность - как разбить последовательность онлайн: https://riptutorial.com/ru/common-lisp/topic/1454/последовательность---как-разбить-последовательность

глава 19: Протокол Meta-Object CLOS

Examples

Получить имена слотов класса

Допустим, у нас есть класс, как

```
(defclass person ()
  (name email age))
```

Чтобы получить имена слотов класса, мы используем функциональные классы-слоты. Это можно найти в пакете с более узким шваброй, предоставляемом системой более близкого швабра. Чтобы загрузить его, мы используем используемое изображение lisp `(ql:quickload :closer-mop)` . Мы также должны убедиться, что класс завершен до вызова слотов класса.

```
(let ((class (find-class 'person)))
  (c2mop:ensure-finalized class)
  (c2mop:class-slots class))
```

который возвращает список *эффективных* объектов *определения слотов* :

```
(#<SB-MOP:STANDARD-EFFECTIVE-SLOT-DEFINITION S/TRANSFORMATIONS::NAME>
 #<SB-MOP:STANDARD-EFFECTIVE-SLOT-DEFINITION S/TRANSFORMATIONS::EMAIL>
 #<SB-MOP:STANDARD-EFFECTIVE-SLOT-DEFINITION S/TRANSFORMATIONS::AGE>)
```

Обновление слота при изменении другого слота

CLOS MOP предоставляет класс использования слота hook-value, который вызывается при доступе к слоту, чтении или изменении. Поскольку мы только заботимся о модификациях, в этом случае мы определяем метод для `(setf slot-value-using-class)` .

```
(defclass document ()
  ((id :reader id :documentation "A hash computed with the contents of every other slot")
   (title :initarg :title :accessor title)
   (body :initarg :body :accessor body)))

(defmethod (setf c2mop:slot-value-using-class) :after
    (new class (object document) (slot c2mop:standard-effective-slot-definition))
  ;; To avoid this method triggering a call to itself, we check that the slot
  ;; the modification occurred in is not the slot we are updating.
  (unless (eq (slot-definition-name slot) 'id)
    (setf (slot-value object 'id) (hash-slots object))))
```

Обратите внимание, что, поскольку в экземпляре создание `slot-value` не вызывается, может потребоваться дублировать код в `initialize-instance :after` method

```
(defmethod initialize-instance :after ((obj document) &key)
  (setf (slot-value obj 'id)
        (hash-slots obj)))
```

Прочитайте Протокол Meta-Object CLOS онлайн: https://riptutorial.com/ru/common-lisp/topic/2901/протокол-meta-object-clos

глава 20: Работа с SLIME

Examples

Монтаж

Лучше всего использовать последний SLIME из репозитория Emacs MELPA: пакеты могут быть немного нестабильными, но вы получаете последние функции.

Портальные и мультиплатформенные Emacs, Slime, Quicklisp, SBCL и Git

Вы можете загрузить портативную и многоплатформенную версию Emacs25, уже настроенную с помощью Slime, SBCL, Quicklisp и Git: Portacle . Это быстрый и простой способ добиться успеха. Если вы хотите узнать, как установить все самостоятельно, читайте дальше.

Ручная установка

В файле инициализации GNU Emacs (> = 24.5) (`~/.emacs` или `~/.emacs.d/init.el`) добавьте следующее:

```
;; Use Emacs package system
(require 'package)
;; Add MELPA repository
(add-to-list 'package-archives
             '("melpa" . "http://melpa.milkbox.net/packages/") t)
;; Reload package list
(package-initialize)
(unless package-archive-contents
  (package-refresh-contents))
;; List of packages to install:
(setq package-list
      '(magit                    ; git interface (OPTIONAL)
        auto-complete            ; auto complete (RECOMMENDED)
        auto-complete-pcmp       ; programmable completion
        idle-highlight-mode      ; highlight words in programming buffer (OPTIONAL)
        rainbow-delimiters       ; highlight parenthesis (OPTIONAL)
        ac-slime                 ; auto-complete for SLIME
        slime                    ; SLIME itself
        eval-sexp-fu             ; Highlight evaluated form (OPTIONAL)
        smartparens              ; Help with many parentheses (OPTIONAL)
        ))

;; Install if are not installed
(dolist (package package-list)
  (unless (package-installed-p package)
    (package-install package)))

;; Parenthesis - OPTIONAL but recommended
(show-paren-mode t)
(require 'smartparens-config)
```

```
(sp-use-paredit-bindings)
(sp-pair "(" ")" :wrap "M-(")
(define-key smartparens-mode-map (kbd "C-<right>") 'sp-forward-slurp-sexp)
(define-key smartparens-mode-map (kbd "C-<left>") 'sp-backward-slurp-sexp)
(define-key smartparens-mode-map (kbd "C-S-<right>") 'sp-forward-barf-sexp)
(define-key smartparens-mode-map (kbd "C-S-<left>") 'sp-backward-barf-sexp)

(define-key smartparens-mode-map (kbd "C-)") 'sp-forward-slurp-sexp)
(define-key smartparens-mode-map (kbd "C-(") 'sp-backward-slurp-sexp)
(define-key smartparens-mode-map (kbd "C-}") 'sp-forward-barf-sexp)
(define-key smartparens-mode-map (kbd "C-{") 'sp-backward-barf-sexp)

(sp-pair "(" ")" :wrap "M-(")
(sp-pair "[" "]" :wrap "M-[")
(sp-pair "{" "}" :wrap "M-{")

;; MAIN Slime setup
;; Choose lisp implementation:
;; The first option uses roswell with default sbcl
;; the second option - uses ccl directly
(setq slime-lisp-implementations
      '((roswell ("ros" "-L" "sbcl-bin" "run"))
        (ccl ("ccl64"
              "-K" "utf-8"))))
;; Other settings...
```

SLIME сам по себе в порядке, но он лучше работает с менеджером пакетов Quicklisp . Чтобы установить Quicklisp, следуйте инструкциям на веб-сайте (если вы используете roswell , следуйте инструкциям roswell). После установки в вашем lisp-вызове:

```
(ql:quickload :quicklisp-slime-helper)
```

и добавьте следующие строки в файл инициализации Emacs:

```
;; Find where quicklisp is installed to
;; Add your own location if quicklisp is installed somewhere else
(defvar quicklisp-directories
  '("~/.roswell/lisp/quicklisp/"          ;; default roswell location for quicklisp
    "~/quicklisp/")                       ;; default quicklisp location
  "Possible locations of QUICKLISP")

;; Load slime-helper
(let ((continue-p t)
      (dirs quicklisp-directories))
  (while continue-p
    (cond ((null dirs) (message "Cannot find slime-helper.el"))
          ((file-directory-p (expand-file-name (car dirs)))
           (message "Loading slime-helper.el from %s" (car dirs))
           (load (expand-file-name "slime-helper.el" (car dirs)))
           (setq continue-p nil))
          (t (setq dirs (cdr dirs))))))

;; Autocomplete in SLIME
(require 'slime-autoloads)
(slime-setup '(slime-fancy))

;; (require 'ac-slime)
```

```
 (add-hook 'slime-mode-hook 'set-up-slime-ac)
 (add-hook 'slime-repl-mode-hook 'set-up-slime-ac)
 (eval-after-load "auto-complete"
   '(add-to-list 'ac-modes 'slime-repl-mode))

(eval-after-load "auto-complete"
  '(add-to-list 'ac-modes 'slime-repl-mode))

;; Hooks
(add-hook 'lisp-mode-hook (lambda ()
                            (rainbow-delimiters-mode t)
                            (smartparens-strict-mode t)
                            (idle-highlight-mode t)
                            (auto-complete-mode)))

(add-hook 'slime-mode-hook (lambda ()
                             (set-up-slime-ac)
                             (auto-complete-mode)))

(add-hook 'slime-repl-mode-hook (lambda ()
                                  (rainbow-delimiters-mode t)
                                  (smartparens-strict-mode t)
                                  (set-up-slime-ac)
                                  (auto-complete-mode)))
```

После перезагрузки GNU Emacs установит и настроит все необходимые пакеты.

Запуск и завершение SLIME, специальных (запятых) команд REPL

В Emacs `Mx slime` запустит слизь с реализацией Common Lisp по умолчанию (первая). Если имеется несколько реализаций (через переменные `slime-lisp-implementations`), к другим реализациям можно получить доступ через `M-- Mx slime` , которые будут предлагать выбор доступных реализаций в мини-буфере.

`Mx slime` откроет буфер REPL, который будет выглядеть следующим образом:

```
; SLIME 2016-04-19
CL-USER>
```

Буфер SLIME REPL принимает несколько специальных команд. Все они начинаются с `,` . Один раз `,` набрав, список параметров будет показан в мини-буфере. Они включают:

- `,quit`
- `,restart-inferior-lisp`
- `,pwd` - печатает текущую директорию, откуда работает Lisp
- `,cd` - изменит текущий каталог

Использование REPL

```
CL-USER> (+ 2 3)
5
CL-USER> (sin 1.5)
0.997495
```

```
CL-USER> (mapcar (lambda (x) (+ x 2)) '(1 2 3))
(3 4 5)
```

Результат, который печатается после оценки, является не только строкой: над объектом Lisp находится полный объект, который можно проверить, щелкнув правой кнопкой мыши и выбрав «Осмотр».

Также возможен многострочный вход: используйте `Cj` для ввода новой строки. `Enter` -key приведет к оценке введенной формы и, если форма не будет завершена, вероятно, вызовет ошибку:

```
CL-USER> (mapcar (lambda (x y)
                   (declare (ignore y))
                   (* x 2))
                 '(1 2 3)
                 '(:a :b :c))
(2 4 6)
```

Обработка ошибок

Если оценка вызывает ошибку:

```
CL-USER> (/ 3 0)
```

Появится буфер отладчика со следующим содержимым (в SBSL lisp):

```
arithmetic error DIVISION-BY-ZERO signalled
Operation was /, operands (3 0).
   [Condition of type DIVISION-BY-ZERO]

Restarts:
 0: [RETRY] Retry SLIME REPL evaluation request.
 1: [*ABORT] Return to SLIME's top level.
 2: [ABORT] abort thread (#<THREAD "repl-thread" RUNNING {1004FA8033}>)

Backtrace:
  0: (SB-KERNEL::INTEGER-/-INTEGER 3 0)
  1: (/ 3 0)
  2: (SB-INT:SIMPLE-EVAL-IN-LEXENV (/ 3 0) #<NULL-LEXENV>)
  3: (EVAL (/ 3 0))
  4: (SWANK::EVAL-REGION "(/ 3 0) ..)
  5: ((LAMBDA NIL :IN SWANK-REPL::REPL-EVAL))
--- more ---
```

Перемещение курсора вниз прошло `--- more ---` приведет к дальнейшему расширению backtrace.

В каждой строке обратной линии нажмите `Enter` чтобы отобразить больше информации о конкретном вызове (если имеется).

Нажатие `Enter` в строке перезапуска приведет к вызову определенного перезапуска. В качестве альтернативы, перезапуск может быть выбран с помощью номера `0` , `1` или `2` (нажмите соответствующую клавишу в любом месте буфера). Перезапуск по умолчанию отмечен звездой и может быть вызван нажатием клавиши `q` (для «quit»). Нажатие `q` закроет отладчик и покажет следующее в REPL

```
; Evaluation aborted on #<DIVISION-BY-ZERO {10064CCE43}>.
CL-USER>
```

Наконец, довольно редко, но Lisp может столкнуться с ошибкой, которую нельзя обработать отладчиком Lisp, и в этом случае он перейдет в низкоуровневый отладчик или закончит ненормально. Чтобы увидеть причину такого рода ошибок, `*inferior-lisp*` буфер `*inferior-lisp*` .

Настройка сервера SWANK через туннель SSH.

1. Установите общую реализацию Lisp на сервере. (Например, `sbcl` , `clisp` и т. Д.)
2. Установите `quicklisp` на сервере.
3. Загрузить SWANK с помощью `(ql:quickload :swank)`
4. Запустите сервер с помощью `(swank:create-server)` . Порт по умолчанию - `4005` .
5. [На вашем локальном компьютере] Создайте туннель SSH с помощью `ssh -L4005:127.0.0.1:4005 [remote machine]`
6. Подключитесь к работающему удаленному серверу swank с `Mx slime-connect` . Хост должен быть `127.0.0.1` и порт `4005` .

Прочитайте Работа с SLIME онлайн: https://riptutorial.com/ru/common-lisp/topic/4097/работа-с-slime

глава 21: Работа с базами данных

Examples

Простое использование PostgreSQL с Postmodern

Postmodern - это библиотека для взаимодействия реляционной базы данных PostgreSQL . Он предлагает несколько уровней доступа к PostgreSQL, от выполнения SQL-запросов, представленных в виде строк или списков, к объектно-реляционному сопоставлению.

База данных, используемая в следующих примерах, может быть создана с помощью этих операторов SQL:

```
create table employees
  (empid integer not null primary key,
   name text not null,
   birthdate date not null,
   skills text[] not null);
insert into employees (empid, name, birthdate, skills) values
  (1, 'John Orange', '1991-07-26', '{C, Java}'),
  (2, 'Mary Red', '1989-04-14', '{C, Common Lisp, Hunchentoot}'),
  (3, 'Ron Blue', '1974-01-17', '{JavaScript, Common Lisp}'),
  (4, 'Lucy Green', '1968-02-02', '{Java, JavaScript}');
```

В первом примере показан результат простого запроса, возвращающего отношение:

```
CL-USER> (ql:quickload "postmodern")   ; load the system postmodern (nickname: pomo)
("postmodern")
CL-USER> (let ((parameters '("database" "dbuser" "dbpass" "localhost")))
           (pomo:with-connection parameters
             (pomo:query "select name, skills from employees")))
(("John Orange" #("C" "Java"))                  ; output manually edited!
 ("Mary Red" #("C" "Common Lisp" "Hunchentoot"))
 ("Ron Blue" #("JavaScript" "Common Lisp"))
 ("Lucy Green" #("Java" "JavaScript")))
4                                               ; the second value is the size of the result
```

Обратите внимание, что результат может быть возвращен как список alists или plists, добавляющий необязательные параметры `:alists` или `:plists` к функции запроса.

Альтернативой `query` является `doquery` , чтобы перебирать результаты запроса. Его параметрами являются `query (&rest names) &body body` , где имена привязаны к значениям в строке на каждой итерации:

```
CL-USER> (let ((parameters '("database" "dbuser" "dbpass" "localhost")))
           (pomo:with-connection parameters
             (format t "The employees that knows Java are:~%")
             (pomo:doquery "select empid, name from employees where skills @> '{Java}'" (i n)
               (format t "~a (id = ~a)~%" n i))))
```

```
The employees that knows Java are:
John Orange (id = 1)
Lucy Green (id = 4)
NIL
2
```

Когда запрос требует параметров, можно использовать подготовленные операторы:

```
CL-USER> (let ((parameters '("database" "dbuser" "dbpass" "localhost")))
           (pomo:with-connection parameters
             (funcall
               (pomo:prepare "select name, skills from employees where skills @> $1")
               #("Common Lisp"))))    ; find employees with skills including Common Lisp
(("Mary Red" #("C" "Common Lisp" "Hunchentoot"))
 ("Ron Blue" #("JavaScript" "Common Lisp")))
2
```

Функция `prepare` получает запрос с заполнителями `$1` , `$2` и т. Д. И возвращает новую функцию, которая требует одного параметра для каждого заполнителя и выполняет запрос при вызове с правильным количеством аргументов.

В случае обновлений функция `exec` возвращает количество измененных кортежей (два оператора DDL заключены в транзакцию):

```
CL-USER> (let ((parameters '("database" "dbuser" "dbpass" "localhost")))
           (pomo:with-connection parameters
             (pomo:ensure-transaction
               (values
                 (pomo:execute "alter table employees add column salary integer")
                 (pomo:execute "update employees set salary =
                                      case when skills @> '{Common Lisp}'
                                      then 100000 else 50000 end")))))
0
4
```

В дополнение к написанию SQL-запросов в виде строк можно использовать списки ключевых слов, символов и констант с синтаксисом, напоминающим lisp (S-SQL):

```
CL-USER> (let ((parameters '("database" "dbuser" "dbpass" "localhost")))
           (pomo:with-connection parameters
             (pomo:query (:select 'name :from 'employees :where (:> 'salary 60000)))))
(("Mary Red") ("Ron Blue"))
2
```

Прочитайте Работа с базами данных онлайн: https://riptutorial.com/ru/common-lisp/topic/4558/работа-с-базами-данных

глава 22: Равенство и другие предикаты сравнения

Examples

Разница между EQ и EQL

1. `EQ` проверяет, имеют ли два значения один и тот же адрес памяти: другими словами, он проверяет, являются ли эти два значения *одинаковыми* , *идентичный* объект. Таким образом, это можно считать тестом идентификации, и его следует применять *только* к структурам: conses, массивам, структурам, объектам, как правило, чтобы увидеть, действительно ли вы имеете дело с тем же объектом, «достигшим» через разные пути или с помощью различные переменные.

2. `EQL` проверяет, являются ли две структуры одним и тем же объектом (например, `EQ`) *или* являются одинаковыми неструктурированными значениями (то есть теми же численными значениями для чисел одного и того же типа или символьных значений). Поскольку он включает в себя оператор `EQ` и может использоваться также для неструктурированных значений, является наиболее важным и наиболее часто используемым оператором, и почти все примитивные функции, для которых требуется сравнение равенства, например, `MEMBER` , *используют по умолчанию этот оператор* .

Таким образом, всегда верно, что `(EQ XY)` подразумевает `(EQL XY)` , а наоборот - не выполняется.

Несколько примеров могут устранить разницу между двумя операторами:

```
(eq 'a 'a)
T ;; => since two s-expressions (QUOTE A) are “internalized” as the same symbol by the reader.
(eq (list 'a) (list 'a))
NIL ;; => here two lists are generated as different objects in memory
(let* ((l1 (list 'a))
       (l2 l1))
  (eq l1 l2))
T ;; => here there is only one list which is accessed through two different variables
(eq 1 1)
?? ;; it depends on the implementation: it could be either T or NIL if integers are “boxed”
(eq #\a #\a)
?? ;; it depends on the implementation, like for numbers
(eq 2d0 2d0)
?? ;; => dependes on the implementation, but usually is NIL, since numbers in double
   ;;    precision are treated as structures in many implementations
(let ((a1 2d0)
      (a2 2d0))
  (eq a1 a2))
?? ;; => also in this case the results depends on the implementation
```

Попробуем те же примеры с `EQL` :

```
(eql 'a 'a)
T ;; => equal because they are the same value, as for EQ
(eql (list 'a) (list 'a))
NIL ;; => different because they different objects in memory, as for EQ
(let* ((l1 (list 'a))
       (l2 l1))
  (eql l1 l2))
T ;; => as above
(eql 1 1)
T ;; they are the same number, even if integers are “boxed”
(eql #\a #\a)
T ;; they are the same character
(eql 2d0 2d0)
T ;; => they are the same number, even if numbers in double precision are treated as
   ;;   structures in many implementations
(let ((a1 2d0)
      (a2 2d0))
  (eql a1 a2))
T ;; => as before
(eql 2 2.0)
NIL;; => since the two values are of a different numeric type
```

Из примеров видно, почему оператор `EQL` следует использовать для надёжной проверки «идентичности» для всех значений, структурированных и неструктурированных, и почему многие эксперты советуют не использовать `EQ` в целом.

Структурное равенство с EQUAL, EQUALP, TREE-EQUAL

Эти три оператора реализуют структурную эквивалентность, то есть проверяют, имеют ли разные сложные объекты эквивалентную структуру с эквивалентным компонентом.

`EQUAL` ведет себя как `EQL` для неструктурированных данных, тогда как для структур, созданных conses (списки и деревья) и двух специальных типов массивов, строк и битовых векторов, он выполняет *структурную эквивалентность* , возвращая true на две структуры, которые являются изоморфными и чьи элементарные компоненты соответственно равны `EQUAL` . Например:

```
(equal (list 1 (cons 2 3)) (list 1 (cons 2 (+ 2 1))))
T ;; => since the two arguments are both equal to (1 (2 . 3))
(equal "ABC" "ABC")
T ;; => equality on strings
(equal "Abc" "ABC")
NIL ;; => case sensitive equality on strings
(equal '(1 . "ABC") '(1 . "ABC"))
T ;; => equal since it uses EQL on 1 and 1, and EQUAL on "ABC" and "ABC"
(let* ((a (make-array 3 :initial-contents '(1 2 3)))
       (b (make-array 3 :initial-contents '(1 2 3)))
       (c a))
  (values (equal a b)
          (equal a c)))
NIL ;; => the structural equivalence is not used for general arrays
T   ;; => a and c are alias for the same object, so it is like EQL
```

`EQUALP` возвращает true во всех случаях, когда `EQUAL` является истинным, но он использует также структурную эквивалентность для массивов любого вида и размерности, для структур и для хэш-таблиц (но не для экземпляров класса!). Более того, для строк используется нечувствительность к регистру.

```
(equalp "Abc" "ABC")
T ;; => case insensitive equality on strings
(equalp (make-array 3 :initial-contents '(1 2 3))
        (make-array 3 :initial-contents (list 1 2 (+ 2 1))))
T ;; => the structural equivalence is used also for any kind of arrays
(let ((hash1 (make-hash-table))
      (hash2 (make-hash-table)))
      (setf (gethash 'key hash1) 42)
      (setf (gethash 'key hash2) 42)
      (print (equalp hash1 hash2))
      (setf (gethash 'another-key hash1) 84)
      (equalp hash1 hash2))
T   ;; => after the first two insertions, hash1 and hash2 have the same keys and values
NIL ;; => after the third insertion, hash1 and hash2 have different keys and values
(progn (defstruct s) (equalp (make-s) (make-s)))
T ;; => the two values are structurally equal
(progn (defclass c () ()) (equalp (make-instance 'c) (make-instance 'c)))
NIL ;; => two structurally equivalent class instances returns NIL, it's up to the user to
    ;;    define an equality method for classes
```

Наконец, `TREE-EQUAL` может применяться к структурам, построенным через `cons` и проверяет, являются ли они изоморфными, как `EQUAL` , но оставляя пользователю выбор функции, которую можно использовать для сравнения листьев, т. Е. Несовпадающих (атомов) который может быть любого другого типа данных (по умолчанию тест, используемый для атома, является `EQL`). Например:

```
(let ((l1 '(1 . ("A" . 2)))
      (l2 '(1 . ("A" . 2))))
  (tree-equal l1 l2 :test #'eql))
NIL ;; => since (eql "A" "A") gives NIL
(let ((l1 '(1 . ("A" . 2)))
      (l2 '(1 . ("A" . 2))))
  (tree-equal l1 l2 :test #'equal))
T ;; since (equal "A" "A") gives T
```

Операторы сравнения по числовым значениям

Числовые значения могут сравниваться с `=` и другими операторами числового сравнения (`/=` , `<` , `<=` , `>` , `>=`), которые игнорируют разницу в физическом представлении различных типов чисел и выполняют сравнение соответствующих математических значений , Например:

```
(= 42 42)
T ;; => both number have the sme numeric type and the same value
(= 1 1.0 1d0)
T ;; => all the tree values represent the number 1, while for instance (eql 1 1d0) => NIL
  ;;    since it returns true only if the operands have the same numeric type
```

```
(= 0.0 -0.0)
T ;; => again, the value is the same, while (eql 0.0 -0.0) => NIL
(= 3.0 #c(3.0 0.0))
T ;; => a complex number with 0 imaginary part is equal to a real number
(= 0.33333333 11184811/33554432)
T ;; => since a float number is passed to RATIONAL before comparing it to another number
  ;; => and (RATIONAL 0.33333333) => 11184811/33554432 in 32-bit IEEE floats architectures
(= 0.33333333 0.33333334)
T ;; => since the result of RATIONAL on both numbers is equal in 32-bit IEEE floats
architectures
(= 0.33333333d0 0.33333334d0)
NIL ;; => since the RATIONAL of the two numbers in double precision is different
```

Из этих примеров можно сделать вывод, что `=` - это оператор, который обычно должен использоваться для сравнения между числовыми значениями, если мы не хотим быть строгими в отношении того, что два числовых значения равны, *только* если они имеют одинаковый числовой тип, в в каком случае следует использовать `EQL` .

Операторы сравнения по символам и строкам

Common Lisp имеет 12 типов специфических операторов для сравнения двух символов, 6 из которых чувствительны к регистру, а остальные - нечувствительны к регистру. Их имена имеют простой шаблон, позволяющий легко запомнить их смысл:

С учетом регистра	Без учета регистра
СИМВОЛ =	CHAR-EQUAL
СИМ / =	CHAR-NOT-РАВНО
CHAR <	СИМ-LESSP
CHAR <=	CHAR-NOT-GREATERP
CHAR>	СИМ-GREATERP
CHAR> =	CHAR-NOT-LESSP

Два символа одного и того же случая находятся в том же порядке, что и соответствующие коды, полученные `CHAR-CODE` , тогда как для нечувствительных к регистру сравнений относительный порядок между любыми двумя символами, взятыми из двух диапазонов `a..z` , `A..Z` зависит от реализации , Примеры:

```
(char= #\a #\a)
T ;; => the operands are the same character
(char= #\a #\A)
NIL ;; => case sensitive equality
(CHAR-EQUAL #\a #\A)
T ;; => case insensitive equality
(char> #\b #\a)
```

```
T ;; => since in all encodings (CHAR-CODE #\b) is always greater than (CHAR-CODE #\a)
(char-greaterp #\b \#A)
T ;; => since for case insensitive the ordering is such that A=a, B=b, and so on,
  ;;    and furthermore either 9<A or Z<0.
(char> #\b #\A)
?? ;; => the result is implementation dependent
```

Для строк конкретными операторами являются `STRING=` , `STRING-EQUAL` и т. Д. Со словом STRING вместо CHAR. Две строки равны, если они имеют одинаковое количество символов, *а* соответствующие символы равны в соответствии с `CHAR=` или `CHAR-EQUAL` если тест чувствителен к регистру или нет.

Порядок между строками является лексикографическим порядком для символов двух строк. Когда сравнение заказов выполняется успешно, результатом является не `T` , а индекс первого символа, в котором две строки отличаются (что эквивалентно true, так как каждый объект, отличный от NIL, является «обобщенным логическим» в Common Lisp).

Важным является то , что *все* операторы сравнения на строке принимают четыре параметра: ключевых слов `start1` , `end1` , `start2` , `end2` , которые могут быть использованы для ограничения сравнения только к прилежащему пробегу символов внутри одной или оба строк. Начальный индекс, если опущен, равен 0, оконечный индекс опущен, равен длине строки, а сравнение выполняется в подстроке, начинающейся с символа с индексом `:start` и заканчивается символом с индексом `:end - 1` включен.

Наконец, обратите внимание, что строка, даже с одним символом, не может сравниваться с символом.

Примеры:

```
(string= "foo" "foo")
T ;; => both strings have the same lenght and the characters are `CHAR=` in order
(string= "Foo" "foo")
NIL ;; => case sensitive comparison
(string-equal "Foo" "foo")
T ;; => case insensitive comparison
(string= "foobar" "barfoo" :end1 3 :start2 3)
T ;; => the comparison is perform on substrings
(string< "fooarr" "foobar")
3 ;; => the first string is lexicographically less than the second one and
  ;;   the first character different in the two string has index 3
(string< "foo" "foobar")
3 ;; => the first string is a prefix of the second and the result is its length
```

В качестве особого случая операторы сравнения строк также могут применяться к символам, и сравнение производится по символу `SYMBOL-NAME` . Например:

```
(string= 'a "A")
T ;; since (SYMBOL-NAME 'a) is "A"
(string-equal '|a| 'a)
T ;; since the the symbol names are "a" and "A" respectively
```

Как окончательное замечание, `EQL` для символов эквивалентно `CHAR=` ; `EQUAL` в строках эквивалентен `STRING=` , а `EQUALP` на строках эквивалентен `STRING-EQUAL` .

Overwiew

В Common Lisp существует множество разных предикатов для сравнения значений. Они могут быть классифицированы в следующих категориях:

1. Общие операторы равенства: EQ, EQL, EQUAL, EQUALP. Они могут использоваться для значений любого типа и всегда возвращать логическое значение T или NIL.
2. Тип конкретных операторов равенства: = и = для чисел, CHAR = CHAR = CHAR-EQUAL CHAR-NOT-EQUAL для символов, STRING = STRING = STRING-EQUAL STRING-NOT-EQUAL для строк, TREE-EQUAL для conses.
3. Операторы сравнения для числовых значений: <, <=,>,> =. Они могут применяться к любому типу числа и сравнивать математическое значение числа, независимо от фактического типа.
4. Операторы сравнения для символов, такие как CHAR <, CHAR-LESSP и т. Д., Которые сравнивают символы либо чувствительным к делу способом, либо нечувствительным к регистру образом в соответствии с зависимым от реализации порядком, который сохраняет естественное алфавитное упорядочение.
5. Операторы сравнения для строк, такие как STRING <, STRING-LESSP и т. Д., Которые сравнивают строки лексикографически, либо чувствительным к делу способом, либо нечувствительным к регистру образом, используя операторы сравнения символов.

Прочитайте Равенство и другие предикаты сравнения онлайн:
https://riptutorial.com/ru/common-lisp/topic/10064/равенство-и-другие-предикаты-сравнения

глава 23: Регулярные выражения

Examples

Использование с сопоставлением с образцом для привязки захваченных групп

`trivia.ppcre` шаблоны сопоставления шаблонов предоставляют систему `trivia.ppcre` которая позволяет связанным захваченным группам с помощью сопоставления с образцом

```
(trivia:match "John Doe"
  ((trivia.ppcre:ppcre "(.*)\\W+(.*)" first-name last-name)
   (list :first-name first-name :last-name last-name)))

;; => (:FIRST-NAME "John" :LAST-NAME "Doe")
```

- Примечание: библиотека Optima предоставляет аналогичное средство в системе `optima.ppcre`

Связывание регистровых групп с CL-PPCRE

`CL-PPCRE:REGISTER-GROUPS-BIND` будет соответствовать строке с регулярным выражением, и если она будет соответствовать, свяжите группы регистров в регулярном выражении с переменными. Если строка не соответствует, возвращается `NIL` .

```
(defun parse-date-string (date-string)
  (cl-ppcre:register-groups-bind
      (year month day)
      ("(\\d{4})-(\\d{2})-(\\d{2})" date-string)
    (list year month day)))

(parse-date-string "2016-07-23") ;=> ("2016" "07" "23")
(parse-date-string "foobar")     ;=> NIL
(parse-date-string "2016-7-23")  ;=> NIL
```

Прочитайте Регулярные выражения онлайн: https://riptutorial.com/ru/common-lisp/topic/2897/регулярные-выражения

глава 24: Рекурсия

замечания

Lisp часто используется в образовательных контекстах, где учащиеся учатся понимать и внедрять рекурсивные алгоритмы.

Производственный код, написанный на Common Lisp или переносимом коде, имеет несколько проблем с рекурсией: они не используют специфические для реализации функции, такие как *оптимизация хвостовых вызовов* , часто делая необходимым избегать рекурсии. В этих случаях реализации:

- Обычно имеют *ограничение глубины рекурсии* из-за ограничений в размерах стека. Таким образом, рекурсивные алгоритмы будут работать только для данных ограниченного размера.
- Не всегда обеспечивайте оптимизацию хвостовых вызовов, особенно в сочетании с операциями с динамическим охватом.
- Только оптимизировать хвостовые звонки на определенных уровнях оптимизации.
- Обычно не предусматривают *оптимизацию хвостовых вызовов* .
- Обычно не предусматривают *оптимизацию вызовов* на определенных платформах. Например, реализации JVM могут не делать этого, поскольку сама JVM не поддерживает *оптимизацию хвостового вызова* .

Замена хвостовых вызовов с помощью переходов обычно затрудняет отладку; Добавление переходов приведет к тому, что фреймы стека станут недоступными в отладчике. В качестве альтернативы Common Lisp обеспечивает:

- Конструкции итераций, такие как `DO` , `DOTIMES` , `LOOP` и другие
- Функции более `REDUCE` порядка, такие как `MAP` , `REDUCE` и другие
- Различные структуры управления, в том числе низкого уровня `go to`

Examples

Множество условий рекурсии 2

```
(defun fn (x)
     (cond (test-condition1 the-value1)
           (test-condition2 the-value2)
           ...
           ...
           ...
           (t (fn reduced-argument-x))))


CL-USER 2788 > (defun my-fib (n)
```

```
                 (cond ((= n 1) 1)
                       ((= n 2) 1)
                       (t (+
                           (my-fib (- n 1))
                           (my-fib (- n 2))))))
MY-FIB

CL-USER 2789 > (my-fib 1)
1

CL-USER 2790 > (my-fib 2)
1

CL-USER 2791 > (my-fib 3)
2

CL-USER 2792 > (my-fib 4)
3

CL-USER 2793 > (my-fib 5)
5

CL-USER 2794 > (my-fib 6)
8

CL-USER 2795 > (my-fib 7)
13
```

Рекурсивный шаблон 1 одиночная хвостовая рекурсия

```
(defun fn (x)
  (cond (test-condition the-value)
        (t (fn reduced-argument-x))))
```

Вычислить n-й номер Фибоначчи

```
;;Find the nth Fibonacci number for any n > 0.
;; Precondition: n > 0, n is an integer. Behavior undefined otherwise.
(defun fibonacci (n)
    (cond
        (                                   ;; Base case.
             ;; The first two Fibonacci numbers (indices 1 and 2) are 1 by definition.
            (<= n 2)                        ;; If n <= 2
            1                               ;; then return 1.
        )
        (t                                  ;; else
            (+                              ;; return the sum of
                                            ;; the results of calling
                (fibonacci (- n 1))         ;; fibonacci(n-1) and
                (fibonacci (- n 2))         ;; fibonacci(n-2).
                                            ;; This is the recursive case.
            )
        )
    )
)
```

Рекурсивно распечатать элементы списка

```
;;Recursively print the elements of a list
(defun print-list (elements)
    (cond
        ((null elements) '()) ;; Base case: There are no elements that have yet to be printed.
Don't do anything and return a null list.
        (t
            ;; Recursive case
            ;; Print the next element.
            (write-line (write-to-string (car elements)))
            ;; Recurse on the rest of the list.
            (print-list (cdr elements))
        )
    )
)
```

Чтобы проверить это, запустите:

```
(setq test-list '(1 2 3 4))
(print-list test-list)
```

Результатом будет:

```
1
2
3
4
```

Вычислить факториал целого числа

Один простой алгоритм для реализации в качестве рекурсивной функции является факториалом.

```
;;Compute the factorial for any n >= 0. Precondition: n >= 0, n is an integer.
(defun factorial (n)
    (cond
        ((= n 0) 1) ;; Special case, 0! = 1
        ((= n 1) 1) ;; Base case, 1! = 1
        (t
            ;; Recursive case
            ;; Multiply n by the factorial of n - 1.
            (* n (factorial (- n 1)))
        )
    )
)
```

Прочитайте Рекурсия онлайн: https://riptutorial.com/ru/common-lisp/topic/3190/рекурсия

глава 25: Согласование образцов

Examples

обзор

Две основные библиотеки, обеспечивающие соответствие шаблонов в Common Lisp, - это Optima и Trivia . Оба обеспечивают аналогичный API-интерфейс и синтаксис. Однако пустяки предоставляют унифицированный интерфейс для расширения соответствия, `defpattern` .

Отправка запросов Clack

Поскольку запрос clack представлен как plist, мы можем использовать сопоставление шаблонов в качестве точки входа в приложение clack как способ маршрутизации запроса к соответствующим контроллерам

```
(defvar *app*
  (lambda (env)
    (match env
      ((plist :request-method :get
              :request-uri uri)
       (match uri
         ("/" (top-level))
         ((ppcre "/tag/(\\w+)/$" name) (tag-page name)))))))
```

Примечание. Чтобы запустить `*app*` мы передаем его в clackup. ej `(clack:clackup *app*)`

DEFUN матча

Используя сопоставление с образцом, можно переплетать определение функции и сопоставление шаблонов, аналогично SML.

```
(trivia:defun-match fib (index)
  "Return the corresponding term for INDEX."
  (0 1)
  (1 1)
  (index (+ (fib (1- index)) (fib (- index 2)))))

(fib 5)
;; => 8
```

Конструкторские узоры

Консоли, структуры, векторы, списки и т. Д. Могут быть сопоставлены с шаблонами конструктора.

```
(loop for i from 1 to 30
      do (format t "~5<~a~;~>"
                 (match (cons (mod i 3)
                              (mod i 5))
                   ((cons 0 0) "Fizzbuzz")
                   ((cons 0 _) "Fizz")
                   ((cons _ 0) "Buzz")
                   (_ i)))
      when (zerop (mod i 5)) do (terpri))
; 1    2    Fizz 4    Buzz
; Fizz 7    8    Fizz Buzz
; 11   Fizz 13   14   Fizzbuzz
; 16   17   Fizz 19   Buzz
; Fizz 22   23   Fizz Buzz
; 26   Fizz 28   29   Fizzbuzz
```

Guard-паттерн

Шаблоны Guard могут использоваться для проверки того, что значение удовлетворяет произвольной тестовой форме.

```
(dotimes (i 5)
  (format t "~d: ~a~%"
          i (match i
              ((guard x (oddp x)) "Odd!")
              (_ "Even!"))))
; 0: Even!
; 1: Odd!
; 2: Even!
; 3: Odd!
; 4: Even!
```

Прочитайте Согласование образцов онлайн: https://riptutorial.com/ru/common-lisp/topic/2933/согласование-образцов

глава 26: Создание бинарников

Examples

Строительство Buildapp

Автономные Common Lisp-файлы могут быть созданы с помощью `buildapp` . Прежде чем мы сможем использовать его для создания двоичных файлов, нам нужно его установить и построить.

Самый простой способ я знаю, как использовать `quicklisp` и Common Lisp (в этом примере используется [`sbcl`], но не имеет значения, какой из них у вас есть).

```
$ sbcl

This is SBCL 1.3.5.nixos, 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.

* (ql:quickload :buildapp)
To load "buildapp":
  Load 1 ASDF system:
    buildapp
; Loading "buildapp"

(:BUILDAPP)

* (buildapp:build-buildapp)
;; loading system "buildapp"
[undoing binding stack and other enclosing state... done]
[saving current Lisp image into /home/inaimathi/buildapp:
writing 4800 bytes from the read-only space at 0x20000000
writing 3216 bytes from the static space at 0x20100000
writing 47349760 bytes from the dynamic space at 0x1000000000
done]
NIL

* (quit)

$ ls -lh buildapp
-rwxr-xr-x 1 inaimathi inaimathi 46M Aug 13 20:12 buildapp
$
```

Когда вы создадите этот двоичный файл, вы можете использовать его для создания двоичных файлов ваших общих программ Lisp. Если вы намереваетесь сделать это много, вы также можете поместить его где-нибудь в свой `PATH` чтобы вы могли просто запустить его с помощью `buildapp` из любого каталога.

Buildapp Hello World

Простейший возможный двоичный код, который вы могли бы построить

1. Не имеет зависимостей
2. Не принимает аргументы командной строки
3. Просто пишет «Привет, мир!». `stdout`

После того, как вы построили `buildapp` , вы можете просто ...

```
$ buildapp --eval '(defun main (argv) (declare (ignore argv)) (write-line "Hello, world!"))' --entry main --output hello-world
[undoing binding stack and other enclosing state... done]
[saving current Lisp image into hello-world:
writing 4800 bytes from the read-only space at 0x20000000
writing 3216 bytes from the static space at 0x20100000
writing 43220992 bytes from the dynamic space at 0x1000000000
done]

$ ./hello-world
Hello, world!

$
```

Buildapp Hello Web World

Более реалистичный пример включает проект, который вы создаете с несколькими файлами на диске (а не с опцией `--eval` переданной `buildapp`), и некоторыми зависимостями, чтобы втянуть.

Поскольку во время поиска и загрузки систем `asdf` (включая загрузку других, потенциально несвязанных систем) может случиться что-либо, недостаточно просто проверить файлы `asd` проектов, в которых вы находитесь, чтобы узнать, что вам нужно загрузить , Общий подход - использовать `quicklisp` для загрузки целевой системы, а затем вызвать `ql:write-asdf-manifest-file` чтобы выписать полный манифест всего загруженного.

Вот игрушечная система, построенная с помощью `hunchentoot` чтобы проиллюстрировать, как это может произойти на практике:

```
;;;; buildapp-hello-web-world.asd

(asdf:defsystem #:buildapp-hello-web-world
  :description "An example application to use when getting familiar with buildapp"
  :author "inaimathi <leo.zovic@gmail.com>"
  :license "Expat"
  :depends-on (#:hunchentoot)
  :serial t
  :components ((:file "package")
               (:file "buildapp-hello-web-world"))
```

```
;;;; package.lisp

(defpackage #:buildapp-hello-web-world
  (:use #:cl #:hunchentoot))
```

```
;;;; buildapp-hello-web-world.lisp

(in-package #:buildapp-hello-web-world)

(define-easy-handler (hello :uri "/") ()
  (setf (hunchentoot:content-type*) "text/plain")
  "Hello Web World!")

(defun main (argv)
  (declare (ignore argv))
  (start (make-instance 'easy-acceptor :port 4242))
  (format t "Press any key to exit...~%")
  (read-char))
```

```
;;;; build.lisp
(ql:quickload :buildapp-hello-web-world)
(ql:write-asdf-manifest-file "/tmp/build-hello-web-world.manifest")
(with-open-file (s "/tmp/build-hello-web-world.manifest" :direction :output :if-exists
:append)
  (format s "~a~%" (merge-pathnames
            "buildapp-hello-web-world.asd"
            (asdf/system:system-source-directory
             :buildapp-hello-web-world))))
```

```
#### build.sh
sbcl --load "build.lisp" --quit

buildapp --manifest-file /tmp/build-hello-web-world.manifest --load-system hunchentoot --load-
system buildapp-hello-web-world --output hello-web-world --entry buildapp-hello-web-world:main
```

После того, как вы сохраните эти файлы в каталоге с именем `buildapp-hello-web-world` , вы можете сделать

```
$ cd buildapp-hello-web-world/

$ sh build.sh
This is SBCL 1.3.7.nixos, 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.
To load "cffi":
  Load 1 ASDF system:
    cffi
; Loading "cffi"
........
To load "buildapp-hello-web-world":
  Load 1 ASDF system:
    buildapp-hello-web-world
```

```
; Loading "buildapp-hello-web-world"
....
;; loading system "cffi"
;; loading system "hunchentoot"
;; loading system "buildapp-hello-web-world"
[undoing binding stack and other enclosing state... done]
[saving current Lisp image into hello-web-world:
writing 4800 bytes from the read-only space at 0x20000000
writing 4624 bytes from the static space at 0x20100000
writing 66027520 bytes from the dynamic space at 0x1000000000
done]

$ ls -lh hello-web-world
-rwxr-xr-x 1 inaimathi inaimathi 64M Aug 13 21:17 hello-web-world
```

Это создает двоичный файл, который делает именно то, что, по вашему мнению, должен, учитывая вышеизложенное.

```
$ ./hello-web-world
Press any key to exit...
```

Затем вы можете запустить еще одну оболочку, сделать `curl localhost:4242` и увидеть ответ в стиле « `Hello Web World!` распечатайте.

Прочитайте Создание бинарников онлайн: https://riptutorial.com/ru/common-lisp/topic/5457/создание-бинарников

глава 27: Типы списков

Examples

Обычные списки

Обычные списки - это самый простой тип списка в Common Lisp. Они представляют собой упорядоченную последовательность элементов. Они поддерживают основные операции, такие как получение первого элемента списка и остальной части списка за постоянное время, поддержка произвольного доступа в линейном времени.

```
(list 1 2 3)
;=> (1 2 3)

(first (list 1 2 3))
;=> 1

(rest (list 1 2 3))
;=> (2 3)
```

Существует множество функций, которые работают в «простых» списках, поскольку они только заботятся об элементах списка. К ним относятся **find** , **mapcar** и многие другие. (Многие из этих функций также будут работать над 17.1 концепциями последовательности для некоторых из этих функций.

Списки ассоциаций

Обычные списки полезны для представления последовательности элементов, но иногда более полезно представлять тип ключа для сопоставления значений. Common Lisp предоставляет несколько способов сделать это, включая настоящие хеш-таблицы (см. 18.1 «Хэш-таблицы»). Существует два основных способа или представление ключей для сопоставлений значений в Common Lisp: списки свойств и списки ассоциаций . В этом примере описаны списки ассоциаций.

Список ассоциаций, или *alist,* является «простым» списком, элементами которого являются пунктирные пары, в которых *автомобиль* каждой пары является ключом, а *cdr* каждой пары является ассоциированным значением. Например,

```
(defparameter *ages* (list (cons 'john 34) (cons 'mary 23) (cons 'tim 72)))
```

может рассматриваться как список ассоциаций, который отображает символы, указывающие личное имя, с целым числом, указывающим возраст. Можно реализовать некоторые функции поиска, используя простые функции списка, например, **член** . Например, чтобы получить возраст **Джона** , можно было написать

```
(cdr (first (member 'mary *age* :key 'car)))
;=> 23
```

Функция- **член** возвращает хвост списка, начиная с ячейки cons, чей *автомобиль* является **mary** , то есть **((mary 23) (tim. 72))** , **сначала** возвращает первый элемент этого списка, который является **(mary. 23)** , а **cdr** возвращает правую часть этой пары, которая равна **23** . Хотя это один из способов доступа к значениям в списке ассоциаций, целью такого соглашения, как списки ассоциаций, является абстрагирование от основного представления (списка) и предоставление функций более высокого уровня для работы с структурой данных.

Для списков ассоциации, функция поиска является **ассоциативной** , который принимает ключ, список ассоциаций и дополнительные ключевые слова тестирования (ключ, тест, тест-нет), и возвращает пару для соответствующего ключа:

```
(assoc 'tim *ages*)
;=> (tim . 72)
```

Поскольку результат всегда будет ячейкой cons, если элемент присутствует, если **ассоциатор** возвращает **nil** , то элемент не был в списке:

```
(assoc 'bob *ages*)
;=> nil
```

Для обновления значений в списке ассоциаций **setf** может использоваться вместе с **cdr** . Например, когда придет день рождения **Джона** и возраст возрастает, можно выполнить одно из следующих действий:

```
(setf (cdr (assoc 'john *ages*) 35)

(incf (cdr (assoc 'john *ages*)))
```

incf работает в этом случае, потому что он основан на **setf** .

Списки ассоциаций также могут использоваться как тип двунаправленной карты, так как сопоставление ключей к значениям извлекается на основе значения с помощью функции **обратного связывания rassoc** .

В этом примере список ассоциаций был создан с использованием **списков** и **минусов** явно, но списки ассоциаций также могут быть созданы с помощью **пары** , которая берет список ключей и данных и создает на их основе список ассоциаций:

```
(pairlis '(john mary tim) '(23 67 82))
;=> ((john . 23) (mary . 67) (tim . 82))
```

В список ассоциаций можно добавить одну пару ключей и значений, используя **acons** :

```
(acons 'john 23 '((mary . 67) (tim . 82)))
;=> ((john . 23) (mary . 67) (tim . 82))
```

Функция- **посредник** выполняет поиск по списку слева направо, что означает, что возможно «замаскировать» значения в списке ассоциаций, не удаляя их из списка или не обновляя структуру структуры списка, просто добавляя новые элементы в список начало списка. Для этого **предусмотрена** функция **acons** :

```
(defvar *ages* (pairlis '(john mary tim) '(34 23 72)))

(defvar *new-ages* (acons 'mary 29 *ages*))

*new-ages*
;=> ((mary . 29) (john . 34) (mary . 23) (tim . 72))
```

И теперь поиск для **mary** вернет первую запись:

```
(assoc 'mary *new-ages*)
;=> 29
```

Списки свойств

Обычные списки полезны для представления последовательности элементов, но иногда более полезно представлять тип ключа для сопоставления значений. Common Lisp предоставляет несколько способов сделать это, включая настоящие хеш-таблицы (см. 18.1 «Хэш-таблицы»). Существует два основных способа или представление ключей для сопоставлений значений в Common Lisp: списки свойств и списки ассоциаций . В этом примере описаны списки свойств.

Список свойств или *plist* - это «простой» список, в котором переменные значения интерпретируются как ключи и их связанные значения. Например:

```
(defparameter *ages* (list 'john 34 'mary 23 'tim 72))
```

может рассматриваться как список свойств, который отображает символы, указывающие личное имя, с целым числом, указывающим возраст. Можно реализовать некоторые функции поиска, используя простые функции списка, например, **член** . Например, чтобы получить возраст **Джона** , можно было написать

```
(second (member 'mary *age*))
;=> 23
```

Функция- **член** возвращает хвост списка, начинающийся с **mary** , то есть **(mary 23 tim 72)** , а **второй** возвращает второй элемент этого списка, то есть **23** . Хотя это один из способов доступа к значениям в списке свойств, назначение такого списка, как списки свойств, состоит в том, чтобы абстрагироваться от основного представления (списка) и

предоставлять функции более высокого уровня для работы с структурой данных.

Для списков свойств функция поиска - **getf** , которая принимает список свойств, ключ (обычно называемый *индикатором*) и необязательное значение по умолчанию для возврата, если список свойств не содержит значения для ключа.

```
(getf *ages* 'tim)
;=> 72

(getf *ages* 'bob -1)
;=> -1
```

Для обновления значений в списке свойств может использоваться **setf** . Например, когда придет день рождения **Джона** и возраст возрастает, можно выполнить одно из следующих действий:

```
(setf (getf *ages* 'john) 35)

(incf (getf *ages* 'john))
```

incf работает в этом случае, потому что он основан на **setf** .

Чтобы просмотреть несколько свойств в списке свойств как один раз, используйте **get-properties** .

Функция **getf** выполняет поиск по списку слева направо, что означает, что возможно «замаскировать» значения в списке свойств, не удаляя их из списка или не обновляя ни одну структуру структуры списка. Например, используя **список *** :

```
(defvar *ages* '(john 34 mary 23 tim 72))

(defvar *new-ages* (list* 'mary 29 *ages*))

*new-ages*
;=> (mary 29 john 34 mary 23 tim 72)
```

И теперь поиск для **mary** вернет первую запись:

```
(getf *new-ages* 'mary)
;=> 29
```

Прочитайте Типы списков онлайн: https://riptutorial.com/ru/common-lisp/topic/3744/типы-списков

глава 28: формат

параметры

Лямбда-List	`(format DESTINATION CONTROL-STRING &REST FORMAT-ARGUMENTS)`
`DESTINATION`	вещь, на которую нужно писать. Это может быть выходной поток, `t` (сокращенно для `*standard-output*`) или `nil` (который создает строку для записи)
`CONTROL-STRING`	строка шаблона. Это может быть примитивная строка, или она может содержать директивы командной строки с тильд-префиксами, которые определяют и каким-то образом преобразуют дополнительные аргументы.
`FORMAT-ARGUMENTS`	потенциальные дополнительные аргументы, требуемые данным `CONTROL-STRING` .

замечания

Документацию CLHS для директив `FORMAT` можно найти в Разделе 22.3 . С помощью SLIME вы можете ввести `Cc Cd ~` для поиска документации CLHS для определенной директивы формата.

Examples

Основное использование и простые директивы

Первые два аргумента для форматирования - это выходной поток и строка управления. Основное использование не требует дополнительных аргументов. Передача `t` когда поток записывается в `*standard-output*` .

```
> (format t "Basic Message")
Basic Message
nil
```

Это выражение будет записывать `Basic Message` в стандартный вывод и возвращать `nil` .

Передача `nil` когда поток создает новую строку и возвращает ее.

```
> (format nil "Basic Message")
```

```
"Basic Message"
```

Большинство директив строки управления требуют дополнительных аргументов. `~a` директива («эстетическое») напечатает любой аргумент , как будто по `princ` процедуры. Это печатает форму без каких-либо escape-символов (ключевые слова печатаются без ведущего двоеточия, строки без их окружения и т. Д.).

```
> (format nil "A Test: ~a" 42)
"A Test: 42"
> (format nil "Multiples: ~a ~a ~a ~a" 1 (list 2 3) "four five" :six)
"Multiples: 1 (2 3) four five SIX"
> (format nil "A Test: ~a" :test)
"A Test: TEST"
> (format nil "A Test: ~a" "Example")
"A Test: Example"
```

`~a` опциональный ввод правого или левого вкладок на основе дополнительных входов.

```
> (format nil "A Test: ~10a" "Example")
"A Test: Example   "
> (format nil "A Test: ~10@a" "Example")
"A Test:    Example"
```

Директива `~s` похожа на `~a` , но печатает escape-символы.

```
> (format nil "A Test: ~s" 42)
"A Test: 42"
> (format nil "Multiples: ~s ~s ~s ~s" 1 (list 2 3) "four five" :six)
"Multiples: 1 (2 3) \"four five\" :SIX"
> (format nil "A Test: ~s" :test)
"A Test: :TEST"
> (format nil "A Test: ~s" "Example")
"A Test: \"Example\""
```

Итерирование по списку

Можно перебирать список с помощью директив `~{` и `~}` .

```
CL-USER> (format t "~{~a, ~}~%" '(1 2 3 4 5))
1, 2, 3, 4, 5,
```

`~^` можно использовать для выхода, если осталось больше элементов.

```
CL-USER> (format t "~{~a~^, ~}~%" '(1 2 3 4 5))
1, 2, 3, 4, 5
```

Числовой аргумент может быть задан `~{` чтобы ограничить, сколько итераций можно сделать:

```
CL-USER> (format t "~3{~a~^, ~}~%" '(1 2 3 4 5))
```

```
1, 2, 3,
```

`~@{` будет перебирать оставшиеся аргументы вместо списка:

```
CL-USER> (format t "~a: ~@{~a~^, ~}~%" :foo 1 2 3 4 5)
FOO: 1, 2, 3, 4, 5
```

Подсвечники могут быть повторены с помощью `~:{` :

```
CL-USER> (format t "~:{(~a, ~a) ~}~%" '((1 2) (3 4) (5 6)))
(1, 2) (3, 4) (5, 6)
```

Условные выражения

Условные выражения могут быть выполнены с помощью `~[` и `~]` . Выражения выражения разделяются с помощью `~;` ,

По умолчанию `~[` принимает целое число из списка аргументов и выбирает соответствующее предложение. Оговорки начинаются с нуля.

```
(format t "~@{~[First clause~;Second clause~;Third clause~;Fourth clause~]~%~}"
        0 1 2 3)
; First clause
; Second clause
; Third clause
; Fourth clause
```

Последнее предложение может быть разделено `~:;` вместо этого сделать это else-clause.

```
(format t "~@{~[First clause~;Second clause~;Third clause~:;Too high!~]~%~}"
        0 1 2 3 4 5)
; First clause
; Second clause
; Third clause
; Too high!
; Too high!
; Too high!
```

Если условное выражение начинается с `~:[` , он будет ожидать обобщенного булева вместо целого числа. Он может иметь только два предложения; первая печатается, если логическое значение было `NIL` , а второе - если оно было правдой.

```
(format t "~@{~:[False!~;True!~]~%~}"
        t nil 10 "Foo" '())
; True!
; False!
; True!
; True!
; False!
```

Если условное выражение начинается с `~@[` , должно быть только одно предложение, которое печатается, если ввод, обобщенный логический, является правдивым. Булевы не будут потребляться, если они правдивы.

```
(format t "~@{~@[~s is truthy!~%~]~}"
        t nil 10 "Foo" '())
; T is truthy!
; 10 is truthy!
; "Foo" is truthy!
```

Прочитайте формат онлайн: https://riptutorial.com/ru/common-lisp/topic/687/формат

глава 29: функции

замечания

Анонимные функции могут быть созданы с использованием `LAMBDA` . Локальные функции могут быть определены с помощью `LABELS` или `FLET` . Их параметры определены так же, как и в глобальных названных функциях.

Examples

Обязательные параметры

```
(defun foobar (x y)
  (format t "X: ~s~@
             Y: ~s~%"
          x y))

(foobar 10 20)
; X: 10
; Y: 20
;=> NIL
```

Дополнительные параметры

Необязательные параметры могут быть указаны после необходимых параметров, используя ключевое слово `&OPTIONAL` . После него может быть несколько необязательных параметров.

```
(defun foobar (x y &optional z)
  (format t "X (~s) and Y (~s) are required.~@
             Z (~s) is optional.~%"
          x y z))

(foobar 10 20)
; X (10) and Y (20) are required.
; Z (NIL) is optional.
;=> NIL
(foobar 10 20 30)
; X (10) and Y (20) are required.
; Z (30) is optional.
;=> NIL
```

По умолчанию

Значение по умолчанию может быть задано для необязательных параметров, указав параметр со списком; второе значение является значением по умолчанию. Форма значения

по умолчанию будет оцениваться только в том случае, если аргумент был дан, поэтому его можно использовать для побочных эффектов, например, для сообщения об ошибке.

```
(defun foobar (x y &optional (z "Default"))
  (format t "X (~s) and Y (~s) are required.~@
             Z (~s) is optional.~%"
          x y z))

(foobar 10 20)
; X (10) and Y (20) are required.
; Z ("Default") is optional.
;=> NIL
(foobar 10 20 30)
; X (10) and Y (20) are required.
; Z (30) is optional.
;=> NIL
```

Проверьте, был ли предоставлен дополнительный аргумент

Третий член может быть добавлен в список после значения по умолчанию; имя переменной, которое является истинным, если аргумент был дан, или `NIL` если он не был указан (и используется значение по умолчанию).

```
(defun foobar (x y &optional (z "Default" zp))
  (format t "X (~s) and Y (~s) are required.~@
             Z (~s) is optional. It ~:[wasn't~;was~] given.~%"
          x y z zp))

(foobar 10 20)
; X (10) and Y (20) are required.
; Z ("Default") is optional. It wasn't given.
;=> NIL
(foobar 10 20 30)
; X (10) and Y (20) are required.
; Z (30) is optional. It was given.
;=> NIL
```

Функция без параметров

Глобальные названные функции определяются с помощью `DEFUN` .

```
(defun foobar ()
  "Optional documentation string. Can contain line breaks.

Must be at the beginning of the function body. Some will format the
docstring so that lines are indented to match the first line, although
the built-in DESCRIBE-function will print it badly indented that way.

Ensure no line starts with an opening parenthesis by escaping them
\(like this), otherwise your editor may have problems identifying
```

```
toplevel forms."
  (format t "No parameters.~%"))

(foobar)
; No parameters.
;=> NIL

(describe #'foobar) ; The output is implementation dependant.
; #<FUNCTION FOOBAR>
;   [compiled function]
;
; Lambda-list: ()
; Derived type: (FUNCTION NIL (VALUES NULL &OPTIONAL))
; Documentation:
;   Optional documentation string. Can contain line breaks.
;
;   Must be at the beginning of the function body. Some will format the
;   docstring so that lines are indented to match the first line, although
;   the built-in DESCRIBE-function will print it badly indented that way.
; Source file: /tmp/fileInaZ1P
;=> No values
```

Тело функции может содержать любое количество форм. Значения из последней формы будут возвращены из функции.

Параметр останова

Один параметр rest-параметра может быть задан с ключевым словом `&REST` после необходимых аргументов. Если такой параметр существует, функция может принимать несколько аргументов, которые будут сгруппированы в список в параметре rest. Обратите внимание, что переменная `CALL-ARGUMENTS-LIMIT` определяет максимальное количество аргументов, которые могут использоваться в вызове функции, поэтому количество аргументов ограничено значением специфической реализации не менее 50 или более аргументов.

```
(defun foobar (x y &rest rest)
  (format t "X (~s) and Y (~s) are required.~@
             The function was also given following arguments: ~s~%"
          x y rest))

(foobar 10 20)
; X (10) and Y (20) are required.
; The function was also given following arguments: NIL
;=> NIL
(foobar 10 20 30 40 50 60 70 80)
; X (10) and Y (20) are required.
; The function was also given following arguments: (30 40 50 60 70 80)
;=> NIL
```

Параметры останова и ключевого слова вместе

Параметр rest может быть до параметров ключевого слова. В этом случае он будет содержать список свойств, заданный пользователем. Значения ключевых слов по-прежнему будут привязаны к соответствующему параметру ключевого слова.

```
(defun foobar (x y &rest rest &key (z 10 zp))
  (format t "X (~s) and Y (~s) are required.~@
             Z (~s) is a keyword argument. It ~:[wasn't~;was~] given.~@
             The function was also given following arguments: ~s~%"
          x y z zp rest))

(foobar 10 20)
; X (10) and Y (20) are required.
; Z (10) is a keyword argument. It wasn't given.
; The function was also given following arguments: NIL
;=> NIL
(foobar 10 20 :z 30)
; X (10) and Y (20) are required.
; Z (30) is a keyword argument. It was given.
; The function was also given following arguments: (:Z 30)
;=> NIL
```

Ключевое слово `&ALLOW-OTHER-KEYS` могут быть добавлены в конце лямбда-списка, чтобы позволить пользователю давать аргументы ключевого слова, не определенные как параметры. Они отправятся в список отдыха.

```
(defun foobar (x y &rest rest &key (z 10 zp) &allow-other-keys)
  (format t "X (~s) and Y (~s) are required.~@
             Z (~s) is a keyword argument. It ~:[wasn't~;was~] given.~@
             The function was also given following arguments: ~s~%"
          x y z zp rest))

(foobar 10 20 :z 30 :q 40)
; X (10) and Y (20) are required.
; Z (30) is a keyword argument. It was given.
; The function was also given following arguments: (:Z 30 :Q 40)
;=> NIL
```

Вспомогательные переменные

Ключевое слово `&AUX` можно использовать для определения локальных переменных для функции. Они не являются параметрами; пользователь не может их предоставить.

`&AUX` используются редко. Вы всегда можете использовать `LET` или какой-либо другой способ определения локальных переменных в теле функции.

`&AUX` имеют преимущества, что локальные переменные всего тела функции перемещаются в верхнюю часть, и это делает ненужным один уровень отступов (например, введенный LET).

```
(defun foobar (x y &aux (z (+ x y)))
  (format t "X (~d) and Y (~d) are required.~@
             Their sum is ~d."
          x y z))
```

```
(foobar 10 20)
; X (10) and Y (20) are required.
; Their sum is 30.
;=> NIL
```

Одним из типичных применений может быть разрешение параметров «указатель». Опять же, вам не нужно так поступать; использование `let` так же идиоматично.

```
(defun foo (a b &aux (as (string a)))
  "Combines A and B in a funny way.  A is a string designator, B a string."
  (concatenate 'string as " is funnier than " b))
```

RETURN-FROM, выход из блока или функции

Функции всегда устанавливают блок вокруг тела. Этот блок имеет то же имя, что и имя функции. Это означает, что вы можете использовать `RETURN-FROM` с этим именем блока, чтобы вернуться из функции и вернуть значения.

Вы должны избегать раннего начала, когда это возможно.

```
(defun foobar (x y)
  (when (oddp x)
    (format t "X (~d) is odd. Returning immediately.~%" x)
    (return-from foobar "return value"))
  (format t "X: ~s~@
             Y: ~s~%"
          x y))

(foobar 10 20)
; X: 10
; Y: 20
;=> NIL
(foobar 9 20)
; X (9) is odd. Returning immediately.
;=> "return value"
```

Параметры ключевого слова

Параметры ключевого слова можно определить с помощью ключевого слова `&KEY` . Они всегда являются необязательными (см. Пример дополнительных параметров для подробностей определения). Могут быть несколько параметров ключевых слов.

```
(defun foobar (x y &key (z "Default" zp))
  (format t "X (~s) and Y (~s) are required.~@
             Z (~s) is a keyword argument. It ~:[wasn't~;was~] given.~%"
          x y z zp))

(foobar 10 20)
; X (10) and Y (20) are required.
; Z ("Default") is a keyword argument. It wasn't given.
;=> NIL
(foobar 10 20 :z 30)
```

```
; X (10) and Y (20) are required.
; Z (30) is a keyword argument. It was given.
;=> NIL
```

Прочитайте функции онлайн: https://riptutorial.com/ru/common-lisp/topic/2126/функции

глава 30: Функции как значения первого класса

Синтаксис

- (имя функции); извлекает объект функции этого имени
- #'название ; синтаксический сахар для (имя функции)
- (символ-функция); возвращает функцию, связанную с символом
- (функция funcall args ...); функция вызова с помощью args
- (применить функцию arglist); функция вызова с аргументами, указанными в списке
- (примените функцию arg1 arg2 ... argn arglist); вызов функции с аргументами, заданными arg1, arg2, ..., argn, а остальные в списке arglist

параметры

параметр	подробности
название	некоторый (неоцененный) символ, который называет функцию
условное обозначение	символ
функция	функцию, которая должна называться
арг ...	ноль или более аргументов (*не* список аргументов)
список аргументов	список, содержащий аргументы, передаваемые функции
arg1, arg2, ..., argn	каждый из них представляет собой один аргумент, который должен быть передан функции

замечания

Говоря о Lisp-подобных языках, существует общее различие между тем, что известно как Lisp-1 и Lisp-2. В Lisp-1 символы имеют только значение, и если символ относится к функции, то значением этого символа будет эта функция. В Lisp-2 символы могут иметь отдельные связанные значения и функции, поэтому требуется специальная форма для ссылки на функцию, хранящуюся в символе, а не на значение.

Common Lisp - это в основном Lisp-2, но на самом деле существует более двух пространств имен (к которым могут относиться символы) - символы могут ссылаться, например, на

значения, функции, типы и теги.

Examples

Определение анонимных функций

Функции в Common Lisp являются *значениями первого класса* . Анонимную функцию можно создать с помощью `lambda` . Например, здесь есть функция из 3 аргументов, которые мы затем вызываем с помощью `funcall`

```
CL-USER> (lambda (a b c) (+ a (* b c)))
#<FUNCTION (LAMBDA (A B C)) {10034F484B}>
CL-USER> (defvar *foo* (lambda (a b c) (+ a (* b c))))
*FOO*
CL-USER> (funcall *foo* 1 2 3)
7
```

Анонимные функции также могут использоваться напрямую. Common Lisp предоставляет синтаксис для него.

```
((lambda (a b c) (+ a (* b c)))    ; the lambda expression as the first
                                   ; element in a form
  1 2 3)                           ; followed by the arguments
```

Анонимные функции также могут храниться как глобальные функции:

```
(let ((a-function (lambda (a b c) (+ a (* b c)))))      ; our anonymous function
  (setf (symbol-function 'some-function) a-function))   ; storing it

(some-function 1 2 3)                                   ; calling it with the name
```

Цитированные лямбда-выражения не являются функциями

Обратите внимание, что цитируемые лямбда-выражения не являются функциями в Common Lisp. Это **не** работает:

```
(funcall '(lambda (x) x)
         42)
```

Чтобы преобразовать цитированное выражение лямбда в функцию, используйте `coerce` , `eval` или `funcall` :

```
CL-USER > (coerce '(lambda (x) x) 'function)
#<anonymous interpreted function 4060000A7C>

CL-USER > (eval '(lambda (x) x))
#<anonymous interpreted function 4060000B9C>

CL-USER > (compile nil '(lambda (x) x))
```

```
#<Function 17 4060000CCC>
```

Ссылаясь на существующие функции

Любой символ в Common Lisp имеет слот для переменной, подлежащей привязке, и отдельный слот для связанной функции.

Обратите внимание, что именование в этом примере только для иллюстрации. Глобальные переменные не должны называться `foo` , но `*foo*` . Последняя нотация - это соглашение, дающее понять, что переменная является *специальной* переменной, использующей *динамическое связывание* .

```
CL-USER> (boundp 'foo) ;is FOO defined as a variable?
NIL
CL-USER> (defvar foo 7)
FOO
CL-USER> (boundp 'foo)
T
CL-USER> foo
7
CL-USER> (symbol-value 'foo)
7
CL-USER> (fboundp 'foo) ;is FOO defined as a function?
NIL
CL-USER> (defun foo (x y) (+ (* x x) (* y y)))
FOO
CL-USER> (fboundp 'foo)
T
CL-USER> foo
7
CL-USER> (symbol-function 'foo)
#<FUNCTION FOO>
CL-USER> (function foo)
#<FUNCTION FOO>
CL-USER> (equalp (quote #'foo) (quote (function foo)))
T
CL-USER> (eq (symbol-function 'foo) #'foo)
T
CL-USER> (foo 4 3)
25
CL-USER> (funcall foo 4 3)
;get an error: 7 is not a function
CL-USER> (funcall #'foo 4 3)
25
CL-USER> (defvar bar #'foo)
BAR
CL-USER> bar
#<FUNCTION FOO>
CL-USER> (funcall bar 4 3)
25
CL-USER> #'+
#<FUNCTION +>
CL-USER> (funcall #'+ 2 3)
5
```

Функции более высокого порядка

Common Lisp содержит много функций более высокого порядка, которые передают функции для аргументов и называют их. Возможно, наиболее фундаментальными являются `funcall` и `apply` :

```
CL-USER> (list 1 2 3)
(1 2 3)
CL-USER> (funcall #'list 1 2 3)
(1 2 3)
CL-USER> (funcall #'list 1 2 3 4 5)
(1 2 3 4 5)
CL-USER> (apply #'list '(1 2 3))
(1 2 3)
CL-USER> (apply #'list 1 2 '(4 5))
(1 2 3 4 5)
CL-USER> (apply #'+ 1 (list 2 3))
6
CL-USER> (defun my-funcall (function &rest args)
           (apply function args))
MY-FUNCALL
CL-USER> (my-funcall #'list 1 2 3)
(1 2 3)
```

Существует много других функций более высокого порядка, которые, например, многократно применяют функцию к элементам списка.

```
CL-USER> (map 'list #'/ '(1 2 3 4))
(1 1/2 1/3 1/4)
CL-USER> (map 'vector #'+ '(1 2 3 4 5) #(5 4 3 2 10))
#(6 6 6 6 15)
CL-USER> (reduce #'+ '(1 2 3 4 5))
15
CL-USER> (remove-if #'evenp '(1 2 3 4 5))
(1 3 5)
```

Подведение итогов

Функция **сокращения** может использоваться для суммирования элементов в списке.

```
(reduce '+ '(1 2 3 4))
;;=> 10
```

По умолчанию **сокращение** выполняет *лево-ассоциативное* сокращение, что означает, что сумма 10 вычисляется как

```
(+ (+ (+ 1 2) 3) 4)
```

Первые два элемента сначала суммируются, а затем этот результат (3) добавляется к следующему элементу (3) для получения 6, который, в свою очередь, добавляется к 4, чтобы получить конечный результат.

Это безопаснее, чем использование **apply** (например, in **(apply '+' (1 2 3 4)),** потому что длина списка аргументов, которую можно передать для **применения** , ограничена (см. **Call-arguments-limit**), и **сокращение** будет работать с функциями, которые принимают только два аргумента.

Указав аргумент ключевого слова **from-end** , **reduce** будет обрабатывать список в другом направлении, что означает, что сумма вычисляется в обратном порядке. То есть

```
(reduce '+ (1 2 3 4) :from-end t)
;;=> 10
```

вычисляет

```
(+ 1 (+ 2 (+ 3 4)))
```

Внедрение реверса и аннулирования

Common Lisp уже имеет **обратную** функцию, но если это не так, то его можно легко реализовать с помощью **сокращения** . Учитывая список, подобный

```
(1 2 3) === (cons 1 (cons 2 (cons 3 '())))
```

обратный список

```
(cons 3 (cons 2 (cons 1 '()))) === (3 2 1)
```

Это может быть не очевидное использование **сокращения** , но если у нас есть функция «обратного cons», скажем **xcons** , такая, что

```
(xcons 1 2) === (2 . 1)
```

затем

```
(xcons (xcons (xcons () 1) 2) 3)
```

что является уменьшением.

```
(reduce (lambda (x y)
          (cons y x))
        '(1 2 3 4)
        :initial-value '())
;=> (4 3 2 1)
```

Common Lisp имеет другую полезную функцию, **revappend** , которая представляет собой комбинацию **reverse** и **append** . Концептуально, он меняет список и добавляет его в какой-то хвост:

```
(revappend '(3 2 1) '(4 5 6))
;;=> (1 2 3 4 5 6)
```

Это также может быть реализовано с **уменьшением** . Это факт, это то же самое, что и реализация **обратного** выше, за исключением того, что начальное значение должно быть **(4 5 6)** вместо пустого списка.

```
(reduce (lambda (x y)
          (cons y x))
        '(3 2 1)
        :initial-value '(4 5 6))
;=> (1 2 3 4 5 6)
```

Затворы

Функции запоминают лексическую область действия, в которой они определены. Из-за этого мы можем заключить лямбду в задании закрытия.

```
(defvar *counter* (let ((count 0))
                    (lambda () (incf count))))

(funcall *counter*) ;; => 1
(funcall *counter*) ;; = 2
```

В приведенном выше примере переменная счетчика доступна только для анонимной функции. Это более отчетливо видно в следующем примере

```
(defvar *counter-1* (make-counter))
(defvar *counter-2* (make-counter))

(funcall *counter-1*) ;; => 1
(funcall *counter-1*) ;; => 2
(funcall *counter-2*) ;; => 1
(funcall *counter-1*) ;; => 3
```

Определение функций, выполняющих функции и возвращаемые функции

Простой пример:

```
CL-USER> (defun make-apply-twice (fun)
           "return a new function that applies twice the function`fun' to its argument"
           (lambda (x)
             (funcall fun (funcall fun x))))
MAKE-APPLY-TWICE
CL-USER> (funcall (make-apply-twice #'1+) 3)
5
CL-USER> (let ((pow4 (make-apply-twice (lambda (x) (* x x)))))
           (funcall pow4 3))
81
```

Классический пример функционального состава : $(f \circ g \circ h) (x) = f (g (h (x))$:

```
CL-USER> (defun compose (&rest funs)
           "return a new function obtained by the functional compositions of the parameters"
           (if (null funs)
               #'identity
               (let ((rest-funs (apply #'compose (rest funs))))
                 (lambda (x) (funcall (first funs) (funcall rest-funs x))))))
COMPOSE
CL-USER> (defun square (x) (* x x))
SQUARE
CL-USER> (funcall (compose #'square #'1+ #'square) 3)
100  ;; => equivalent to (square (1+ (square 3)))
```

Прочитайте Функции как значения первого класса онлайн: https://riptutorial.com/ru/common-lisp/topic/1259/функции-как-значения-первого-класса

глава 31: Хэш-таблицы

Examples

Создание хеш-таблицы

Хэш-таблицы создаются с помощью `make-hash-table` :

```
(defvar *my-table* (make-hash-table))
```

Функция может принимать параметры ключевых слов для дальнейшего определения поведения полученной хеш-таблицы:

- `test` : выбор функции, используемой для сравнения ключей для равенства. Может быть, обозначение для одной из функций `eq` , `eql` , `equal` или `equalp` . По умолчанию используется `eq` .
- `size` : подсказка к реализации о пространстве, которое изначально может потребоваться.
- `rehash-size` : Если целое число (> = 1), то при выполнении переименования хэш-таблица увеличит свою емкость на указанное число. Если в противном случае float (> 1.0), то хеш-таблица увеличит его емкость до продукта `rehash-size` и предыдущей емкости.
- `rehash-threshold` : Указывает, как должна быть заполнена хэш-таблица, чтобы вызвать повторный перехват.

Итерация по элементам хэш-таблицы с помощью maphash

```
(defun print-entry (key value)
  (format t "~A => ~A~%" key value))

(maphash #'print-entry *my-table*) ;; => NIL
```

Использование `maphash` позволяет перебирать записи хэш-таблицы. Порядок итераций не указан. Первый аргумент - это функция, принимающая два параметра: ключ и значение текущей записи.

`maphash` всегда возвращает `NIL` .

Итерация по элементам хэш-таблицы с циклом

Макрос **цикла** поддерживает итерацию по ключам, значениям или ключам и значениям хэш-таблицы. Следующие примеры показывают возможности, но полный синтаксис **цикла** позволяет больше комбинаций и вариантов.

По ключам и значениям

```
(let ((ht (make-hash-table)))
  (setf (gethash 'a ht) 1
        (gethash 'b ht) 2)
  (loop for k being each hash-key of ht
     using (hash-value v)
     collect (cons k v)))
;;=> ((A . 1) (B . 2))
```

```
(let ((ht (make-hash-table)))
  (setf (gethash 'a ht) 1
        (gethash 'b ht) 2)
  (loop for v being each hash-value of ht
     using (hash-key k)
     collect (cons k v)))
;;=> ((A . 1) (B . 2))
```

По клавишам

```
(let ((ht (make-hash-table)))
  (setf (gethash 'a ht) 1
        (gethash 'b ht) 2)
  (loop for k being each hash-key of ht
       collect k))
;;=> (A B)
```

За значения

```
(let ((ht (make-hash-table)))
  (setf (gethash 'a ht) 1
        (gethash 'b ht) 2)
  (loop for v being each hash-value of ht
       collect v))
;;=> (1 2)
```

Итерация по элементам хэш-таблицы с помощью итератора хеш-таблицы

Ключи и значения хеш-таблицы можно повторить с помощью макроса **с-хэш-таблицей-итератором** . Это может быть немного сложнее, чем **maphash** или **loop** , но его можно использовать для реализации итерационных конструкций, используемых в этих методах. **with-hash-table-iterator** принимает имя и хэш-таблицу и связывает имя внутри тела, так что последовательные вызовы имени приводят к нескольким значениям: (i) логическое значение, указывающее, присутствует ли значение; (ii) ключ входа; и (iii) значение записи.

```
(let ((ht (make-hash-table)))
  (setf (gethash 'a ht) 1
        (gethash 'b ht) 2)
  (with-hash-table-iterator (iterator ht)
    (print (multiple-value-list (iterator)))
```

```
    (print (multiple-value-list (iterator)))
    (print (multiple-value-list (iterator)))))

;; (T A 1)
;; (T B 2)
;; (NIL)
```

Прочитайте Хэш-таблицы онлайн: https://riptutorial.com/ru/common-lisp/topic/4482/хэш-таблицы

кредиты

S. No	Главы	Contributors
1	Начало работы с общими	blambert, Community, CPHPython, Dan Robertson, Ehvince, Gustav Bertram, Inaimathi, JAL, Rainer Joswig, Renzo, Robert Columbia, WarFox
2	ANSI Common Lisp, языковой стандарт и его документация	Rainer Joswig, sds
3	ASDF - другой механизм определения системы	Inaimathi, jkiiski, Joao Tavora, PuercoPop, Rainer Joswig, Sim, Svante
4	CLOS - общая система объектов Lisp	Joshua Taylor, PuercoPop, Rainer Joswig, Sim
5	LOOP, общий макрос Lisp для итерации	Dan Robertson, Elias Mårtenson, Inaimathi, PuercoPop, Rainer Joswig, RamenChef, Renzo, Throwaway Account 3 Million
6	Streams	jkiiski, Rainer Joswig, Renzo, Svante
7	Группировка форм	Joshua Taylor, Rainer Joswig
8	Единичное тестирование	Ehvince, PuercoPop, Rainer Joswig, sadfaf
9	Консоли и списки	eyqs, Joshua Taylor, Rainer Joswig, Renzo
10	Контрольные структуры	eyqs, Rainer Joswig, Robert Columbia, Soupy, Svante, Throwaway Account 3 Million
11	котировка	MatthewRock, Rainer Joswig, Svante
12	Лексические и специальные переменные	Rainer Joswig, Terje D.
13	Логические и	Rainer Joswig, Renzo, Terje D.

	обобщенные булевы	
14	макрос	JAL, jkiiski, Joshua Taylor, Mark Green, PuercoPop, Rainer Joswig
15	настройка	Daniel Kochmański, Rainer Joswig
16	Основные петли	Joshua Taylor, MatthewRock, Rainer Joswig, sadfaf, Svante
17	Отображение функций по спискам	Aaron, Rainer Joswig, Renzo
18	последовательность - как разбить последовательность	PuercoPop, Rainer Joswig, sadfaf
19	Протокол Meta-Object CLOS	PuercoPop, Rainer Joswig
20	Работа с SLIME	Ehvince, mobiuseng, tsikov
21	Работа с базами данных	Renzo
22	Равенство и другие предикаты сравнения	Renzo
23	Регулярные выражения	jkiiski, PuercoPop
24	Рекурсия	4444, Rainer Joswig, Robert Columbia, sadfaf
25	Согласование образцов	jkiiski, PuercoPop
26	Создание бинарников	Inaimathi
27	Типы списков	jkiiski, Joshua Taylor
28	формат	Dan Robertson, Inaimathi, jkiiski, otyn, Renzo
29	функции	jkiiski, Rainer Joswig, Svante
30	Функции как значения первого класса	Dan Robertson, Joshua Taylor, PuercoPop, Rainer Joswig, Renzo

31	Хэш-таблицы	Daniel Jour, Joshua Taylor