Функциональное программирование

Лекции 6-7.

Диалекты языка Lisp

Диалекты

- xLisp
- Common Lisp
- muLisp
- Scheme
- AutoLisp

xLisp

- Практически классический Lisp

Common Lisp

- Самый распространенный диалект
- Поддерживает объектную модель
- Особенности Common Lisp

(в сравнении с другими языками программирования) ->

Богатая и точная арифметика

- Лисп предоставляет богатую иерархию числовых типов, которые хорошо интегрированы с остальными частями языка.
- **Длинные числа** (bignums) создаются автоматически по мере надобности, что снижает риск переполнений и обеспечивает точность. Например, мы можем быстро вычислить значение 10↑↑4:

➢(expt (expt (expt (expt 10 10) 10) 10) 10)
10000000000000000000000000000000000[...]

- **Рациональные числа** представляются в виде дробей, так что никаких ошибок округления при их использовании не происходит. Точная рациональная арифметика интегрирована в язык (то же и в xLisp)

```
> (+ 5/9 3/4) 47/36
```

- **Комплексные числа** также являются встроенным типом данных в лиспе. Они могут быть представлены в виде краткого синтаксиса: #c(10 5) означает 10 + 5i. Арифметические операции также могут работать с комплексными значениями (то же в xLisp):

➢ (* 2 (+ #c(10 5) 4))

#C(28 10)

Обобщённые ссылки

- *Формы* или *позиции* (places) могут использоваться так, как если бы они были отдельными изменяемыми переменными. При помощи SETF и других подобных конструкций можно изменять значения, которые концептуально связаны с заданной позицией (то же в xLisp).
- Например, можно использовать SETF следующим образом:

Пример 1

- (defvar *colours* (list 'red 'green 'blue))

COLOURS

> (setf (first *colours*) 'yellow)

YELLOW

> *colours*

(YELLOW BLUE GREEN)

Пример 2

- (push 'red (rest *colours*))

(RED BLUE GREEN)

- *colours*

(YELLOW RED BLUE GREEN)

Обобщённые ссылки работают не только в применении к спискам, но и ко многим другим видам структур и объектов. Например, в объектно-ориентированных программах один из способов изменить какое-то поле объекта — при помощи SETF

Множественные значения

- Значения могут быть объединены без явного создания структуры, такой как список. Например, (values 'foo 'bar) возвращает два значения — 'foo и 'bar. При помощи этого механизма функции могут возвращать сразу несколько значений, что может упростить программу (то же в xLisp).

  Например, FLOOR — это стандартная функция, которая возвращает два значения:

```
> (floor pi)
3
0.14159265358979312d0
```

- По соглашению функции, которые возвращают несколько значений, по умолчанию используются так, как будто бы возвращалось только одно значение — первое:

➢ (+ (floor pi) 2)

5

- При этом вы можете явно получить и использовать остальные значения. В следующем примере мы разделяем целую и дробную части PI при округлении:

➢ (multiple-value-bind (integral fractional) (floor pi)

(+ integral fractional))

3.14159265358979 3d0

Макросы

- **Макрос в лиспе** — это своего рода функция, которая получает в качестве аргументов лисповские формы или объекты и, как правило, генерирует код, который затем будет скомпилирован и выполнен. Это происходит до выполнения программы, во время фазы, которая называется *развёрткой макросов* (macroexpansion). Макросы могут выполнять какие-то вычисления во время развёртки, используя полные возможности языка.

  Одно из применений макросов — **преобразовывать какой-либо исходный код в представление, корректное в терминах уже существующих определений**. Другими словами, макросы позволяют добавлять новый синтаксис к языку (такой подход известен как *синтаксическая абстракция*).

  Это позволяет с лёгкостью **встраивать в лисп предметно-ориентированные языки (DSL)**, так как специальный синтаксис может быть добавлен в язык перед выполнением программы.

  Основной выигрыш от использования макросов заключается в том, что они расширяют **возможности языка, позволяя программисту выражать свои идеи проще и при помощи меньшего объёма кода**. Можно добавить в язык новые средства так, как будто они являются встроенными. К тому же, если макросы использовать для предварительного вычисления данных или инициализации, они могут помочь в оптимизации производительности.

Макрос LOOP

- Макрос LOOP — это мощное средство для представления циклов. На самом деле это целый небольшой встроенный язык для описания итерационных процессов. LOOP предоставляет все необходимые типы выражений для записи циклов, от простых повторений до итераторов и сложных конечных автоматов

Пример 1: (в xLisp не работает)

- (defvar *list*
  (loop :for x := (random 1000)
    :repeat 5
    :collect x))
  LIST
- *list*
  (324 794 102 579 55)

Пример 2

- (loop :for elt :in *list*
  :when (oddp elt)
  :maximizing elt)

  579

Пример 3

- (loop :for elt :in *list*
  :collect (log elt)
  :into logs
  :finally
  (return
  (loop :for l :in logs
  :if (> l 5.0) :collect l :into ms
  :else :collect l :into ns
  :finally (return (values ms ns)))))

(5.7807436 6.6770835 6.3613024) (4.624973 4.0073333)

Функция FORMAT

- Функция FORMAT поддерживает встроенный язык для описания того, как данные должны быть отформатированы. Помимо простой текстовой подстановки, инструкции FORMAT-а могут в компактном виде выражать различные правила генерации текста, такие как условия, циклы и обработка граничных случаев (то же в xLisp).

  Мы можем отформатировать список имён при помощи такой функции:

- (defun format-names (list)

  (format nil "~{~:(~a~)~#[.~; and ~:;, ~]~}" list))
- (format-names '(doc grumpy happy sleepy bashful sneezy dopey))

  "Doc, Grumpy, Happy, Sleepy, Bashful, Sneezy and Dopey.“

  > (format-names '(fry laurie))

  "Fry and Laurie.“

  > (format-names '(bluebeard))

  "Bluebeard."

FORMAT передаёт свой результат в указанный поток, будь то стандартный вывод на экран, строка или любой другой поток.

Функции высшего порядка

- Функции в лиспе являются настоящими сущностными первого класса. Функциональные объекты могут динамически создаваться, передаваться в качестве параметров или возвращаться в качестве результата. Таким образом, поддерживаются функции **высшего порядка**, то есть такие, аргументы и возвращаемые значения которых сами могут быть функциями (то же в xLisp).

  Здесь вы видите вызов функции SORT, аргументами которой являются список и ещё одна функция (в данном случае это #'<):

➢ (sort (list 4 2 3 1) #'<)

(1 2 3 4)

- **Анонимные функции**, которые также называют лямбда-выражениями, могут использоваться вместо имени передаваемой функции. Они особенно полезны, когда вы хотите создать функцию для однократного использования, не засоряя программу лишним именем. В общем случае их можно использовать для создания лексических замыканий.

  В данном примере мы создаём анонимную функцию, чтобы использовать её в качестве первого аргумента MAPCAR:

➢ (mapcar (lambda (x)
(+ x 10)) '(1 2 3 4 5))
(11 12 13 14 15)

- При создании функции захватывают контекст, что позволяет нам использовать полноценные лексические замыкания:

```
>(let ((counter 10))
 (defun add-counter (x)
 (prog1
    (+ counter x)
    (incf counter))))
```

➢ (mapcar #'add-counter '(1 1 1 1))

(11 12 13 14)

> (add-counter 50)

64

Обработка списков

- (defvar *nums* (list 0 1 2 3 4 5 6 7 8 9 10 11 12))

NUMS

> (list (fourth *nums*) (nth 8 *nums*))

(3 8)

> (list (last *nums*) (butlast *nums*))

((12) (0 1 2 3 4 5 6 7 8 9 10 11))

> (remove-if-not #'evenp *nums*)

(0 2 4 6 8 10 12)

- А так — с ассоциативным списком

➢ (defvar *capital-cities* '((NZ . Wellington)
(AU . Canberra)
(CA . Ottawa)))

CAPITAL-CITIES

> (cdr (assoc 'CA *capital-cities*))

OTTAWA

> (mapcar #'car *capital-cities*)

(NZ AU CA)

Лямбда-списки

- Лямбда-список задаёт параметры функций, макросов, форм связывания и некоторых других конструкций. Лямбда-списки определяют обязательные, опциональные, именованные, хвостовые (rest) и дополнительные параметры, а также значения по умолчанию и тому подобное. Это позволяет определять очень гибкие и выразительные интерфейсы.

  Опциональные параметры не требуют от вызывающего указывать какое-либо значение. Для них может быть определено значение по умолчанию, в противном случае вызываемый код может проверять, было ли предоставлено значение и действовать по ситуации (то же в xLisp).

  Следующая функция принимает опциональный параметр delimiter, значением по умолчанию для которого является пробельный символ:

```
> (defun explode (string &optional (delimiter #\Space))
 (let ((pos (position delimiter string)))
    (if (null pos) (list string)
          (cons (subseq string 0 pos)
                (explode (subseq string (1+ pos))
                         delimiter)))))
```

При вызове функции EXPLODE мы можем либо предоставить опциональный параметр, либо опустить его.

- (explode "foo, bar, baz" #\,)

("foo " " bar " " baz")

- (explode "foo, bar, baz")

("foo," "bar," "baz")

- **Именованные параметры** аналогичны опциональным параметрам, но их можно передавать в произвольном порядке, поскольку они определяются именами. Использование имён улучшает читаемость кода и служит своего рода документацией, когда вы делаете вызов с несколькими параметрами.

  К примеру, сравните эти два вызова функций:

```
// In C:
 xf86InitValuatorAxisStruct(device, 0, 0, -1, 1, 0, 1);
;; In Lisp:
 (xf86-init-valuator-axis-struct :dev device :ax-num 0
          :min-val 0
         :max-val -1
          :min-res 0
          :max-res 1
          :resolution 1)
```

Символы как сущности первого класса

- Символы — это уникальные объекты, полностью определяемые своими именами. Скажем, 'foo — это символ, чьё имя «FOO». Символы могут использоваться в качестве идентификаторов или как некие абстрактные имена. Сравнение символов происходит за фиксированное время.

  Символы, как и функции, являются сущностями первого класса. Их можно динамически создавать, квотировать (quote, unevaluate), хранить, передавать в качестве аргументов, сравнивать, преобразовывать в строки, экспортировать и импортировать, на них можно ссылаться.

  Здесь '*foo* является идентификатором переменной:

➢ (defvar *foo* 5)

FOO

> (symbol-value '*foo*)

5

Пакеты как сущности первого класса

- Пакеты, которые играют роль пространств имён (namespaces), также являются объектами первого класса. Поскольку их можно создавать, хранить, возвращать в качестве результата во время выполнения программы, возможно динамически переключать контекст или преобразовывать пространства имён динамически (то же в xLisp).

  В следующем примере мы используем INTERN для того, чтобы включить символ в некоторый пакет:

➢ (intern "ARBITRARY"
        (make-package :foo :use '(:cl)))
FOO::ARBITRARY
NIL

- В лиспе есть специальная переменная *package*, которая указывает на текущий пакет. Скажем, если текущим пакетом является FOO, то можно выполнить:

➢ (in-package :foo)

#<PACKAGE "FOO">

> (package-name *package*)

"FOO"

Специальные переменые

- Лисп поддерживает динамический контекст переменных в дополнение к лексическому контексту. Динамические переменные в некоторых случаях могут быть полезны, так что их поддержка позволяет добиться максимальной гибкости.
- Например, мы можем перенаправить вывод какого-то кода в нестандартный поток, такой как файл, создав динамическую связь для специальной переменной *standard-output*:

Специальные переменые (2) Пример (в xLisp не работает)

```
(with-open-file (file-stream #p"somefile" :direction
   :output)
(let ((*standard-output* file-stream))
 (print "This prints to the file, not stdout."))
 (print "And this prints to stdout, not the file."))
```

- Помимо *standard-output*, Лисп включает несколько специальных переменных, которые хранят состояние программы, включая ресурсы и параметры, такие как *standard-input*, *package*, *readtable*, *print-readably*, *print-circle* и т.д.

Передача управления

- В Лиспе есть два способа передачи управления в точку, находящуюся выше в иерархии вызовов. При этом может учитываться лексическая или динамическая область, для локальных и нелокальных переходов, соответственно.

- **Именованные блоки** позволяют вложенной форме вернуть управление из любой именованой родительской формы при помощи BLOCK и RETURN-FROM.
- К примеру, здесь вложенный цикл возвращает список из блока early в обход внешнего цикла (в xLisp не работает):

➢ (block early

(loop :repeat 5 :do

(loop :for x :from 1 :to 10 :collect x :into xs

:finally (return-from early xs))))

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

- **Catch/throw** — это что-то вроде нелокального goto. THROW производит переход к последнему встреченному CATCH и передаёт значение, которое было указано в качестве параметра.
- В функции THROW-RANGE, основанной на предыдущем примере, мы можем применить THROW и CATCH, используя при этом динамическое состояние программы.

```
(defun throw-range (a b)
 (loop :for x :from a :to b :collect x :into xs
  :finally (throw :early xs)))
```

-
```
(catch :early
  (loop :repeat 5 :do
    (throw-range 1 10)))
(1 2 3 4 5 6 7 8 9 10)
```

Когда достаточно использовать лексическую область видимости и catch/throw, когда необходимо учитывать динамическое состояние.

Условия, перезапуск

- Система условий (conditions) в Лиспе — это механизм для передачи сигналов между частями программы.
- Одно из возможных применений — вызывать исключения и обрабатывать их, примерно так же, как это делается в Java или Python. Но, в отличие от других языков, во время передачи сигнала в Лиспе стек *не разворачивается*, поэтому все данные сохраняются и обработчик сигнала может перезапустить программу начиная с любой точки в стеке.
- Этот подход к обработке исключительных ситуаций позволяет улучшить разделение задач и добиться таким образом большей структурированности кода. Но такой механизм имеет более широкую область применения, как системы передачи произволных сообщений (а не только ошибок) между частями программы.

Обобщённые функции

- Объектная ситема Common Lisp (Common Lisp Object System, CLOS) не привязывает методы к классам, а позволяет использовать обобщённые функции.
- Обобщённые функции задают сигнатуры, которым могут удовлетворять несколько различных методов. При вызове выбирается метод, который лучше всего соответствует аргументам.
- Здесь мы определяем обобщённую функцию, которая обрабатывает события от клавиатуры:

(defgeneric key-input (key-name))

Затем определяем несколько методов, которые удовлетворяют различным значениям KEY-NAME.

```
(defmethod key-input (key-name)
;; Default case
(format nil "No keybinding for ~a" key-name))
(defmethod key-input ((key-name (eql :escape)))
(format nil "Escape key pressed"))
(defmethod key-input ((key-name (eql :space)))
(format nil "Space key pressed"))
```

- Посмотрим на вызов методов в действии:

➢ (key-input :space) "Space key pressed«

➢ (key-input :return)

"No keybinding for RETURN«

➢ (defmethod key-input ((key-name (eql :return)))

(format nil "Return key pressed"))

➢ (key-input :return)

"Return key pressed"

- Мы обошлись без конструкций а-ля switch и явной работы с таблицей методов. Таким образом, мы можем добавлять обработку новых частных случаев независимо, динамически, по мере надобности и вообще в любой точке программы. Это, в частности, обеспечивает развитие программ на лиспе «снизу-вверх».
- Обобщённые функции определяют некоторые общие характеристики группы методов. Скажем, способы комбинации методов, опции специализации и другие свойства могут задаваться обобщённой функцией.
- Лисп предоставляет многие полезные стандартные обобщённые функции; примером может служить PRINT-OBJECT, которая может быть специализирована для любого класса, чтобы задать его текстовое представление.

Комбинации методов

- Комбинации методов позволяют при вызове какого-либо метода выполнить целую *цепочку* методов, либо в некотором порядке, либо так, чтобы одни функции обрабатывали результаты других.
- Есть встроенные способы комбинации методов, которые выстраивают методы в заданном порядке. Методы, снабжённые ключевыми словами :before, :after или :around помещаются в соответствующее место в цепочке вызовов.
- Например, в предыдущем примере каждый из методов KEY-INPUT повторяет вывод фразы «key pressed». Мы можем улучшить код при помощи комбинации типа :around

```
(defmethod key-input :around (key-name) (format
    nil "~:(~a~) key pressed«
      (call-next-method key-name)))
```

- После этого мы заново определим методы KEY-INPUT, в каждом из них указав лишь одну строку:

```
(defmethod key-input ((key-name (eql :escape)))
    "escape")
```

- При вызове KEY-INPUT происходит следующее:
  - вызывается метод с меткой :around
  - он вызывает следующий метод, то есть одну из специализированных версий KEY-INPUT,
  - которая возвращает строку, и эту строку форматирует метод с :around.

- Надо заметить, что вариант по умолчанию можно обработать по-разному. Мы можем просто использовать пару THROW/CATCH (более продвинутая реализация могла бы использовать условия):

```
(defmethod key-input (key-name)
  (throw :default
    (format nil "No keybinding for ~a" key-name)))
(defmethod key-input :around (key-name)
  (catch :default
    (format nil "~:(~a~) key pressed«
      (call-next-method key-name))))
```

- В результате, встроенный способ комбинации методов позволяет нам обобщить обработку событий от клавиатуры в модульный, расширяемый, легко изменяемый механизм. Эта техника может быть дополнена при помощи определяемых пользователем способов комбинации; скажем, можно добавить способ комбинации, который будет выполнять суммирование или сортировку результатов методов.

Множественное наследование

- Любой класс может иметь много предков, что позволяет создавать более богатые модели и достигать более эффективного повторного использования кода. Поведение дочерних классов определяется в соответствии с порядком следования, который строится по определениям классов-предков.
- При помощи комбинаций методов, метаобъектного протокола и других особенностей CLOS можно обходить традиционные проблемы множественного наследования (такие как fork-join).

Метаобъектный протокол

- Метаобъектный протокол (Meta-object protocol, MOP) — это программный интерфейс к CLOS, который сам реализован при помощи CLOS. MOP даёт программистам возможность исследовать, использовать и модифицировать внутреннее устройство CLOS через сам CLOS.

Классы как сущности первого класса

- Сами классы также являются объектами. При помощи MOP можно изменять определение и поведение классов.
- Пусть класс FOO является потомком класса BAR, тогда мы можем при помощи функции ENSURE-CLASS добавить, скажем, класс BAZ к списку предков FOO:

```
(defclass bar () ())
 (defclass foo (bar) ())
 (defclass baz () ())
```

- (class-direct-superclasses (find-class 'foo))
  (#<STANDARD-CLASS BAR>)
- (ensure-class 'foo :direct-superclasses
  '(bar baz))

  #<STANDARD-CLASS FOO>
- (class-direct-superclasses (find-class 'foo))

  (#<STANDARD-CLASS BAR>
  #<STANDARD-CLASS BAZ>)

- Мы использовали функцию CLASS-DIRECT-SUPERCLASSES, чтобы получить информацию о предках класса; в данном случае она принимает в качестве аргумента класс в виде объекта, полученного от FIND-CLASS.
- Приведённый пример иллюстрирует механизм, при помощи которого классы могут модифицироваться во время выполнения программы, что позволяет, кроме всего прочего, динамически добавлять в классы примеси (mixins).

Динамические переопределения

- Лисп представляет собой очень интерактивную и динамическую среду. Функции, макросы, классы, пакеты, параметры и объекты могут быть переопределены практически в любое время, и при этом результат будет адекватен и предсказуем.
- Так, если вы переопределили класс во время выполнения программы, изменения немедленно будут применены ко всем объектам и подклассам данного класса. Мы можем определить класс BALL со свойством radius и его подкласс TENNIS-BALL:

- (defclass ball ()
  ((%radius :initform 10 :accessor radius)))

#<STANDARD-CLASS BALL>

> (defclass tennis-ball (ball) ())

#<STANDARD-CLASS TENNIS-BALL>

Вот объект класса TENNIS-BALL, у него есть слот для свойства radius:

> (defvar *my-ball* (make-instance 'tennis-ball))

MY-BALL

> (radius *my-ball*)

10

- А теперь мы можем переопределить класс BALL, добавив в него ещё один слот volume:

```
> (defclass ball ()
 ((%radius :initform 10 :accessor radius)
 (%volume :initform (* 4/3 pi 1e3)
       :accessor volume)))
#<STANDARD-CLASS BALL>
```

И *MY-BALL* автоматически обновился, получив новый слот, который был определён в классе-предке.

```
> (volume *my-ball*)
4188.790204786391d0
```

Доступ к компилятору во время выполнения программы

- Благодаря функциям COMPILE и COMPILE-FILE компилятор Лиспа можно напрямую использовать из выполняемой программы. Таким образом, функции, которые создаются или изменяются во время работы программы, также могут скомпилированы.
- Выходит, программы можно компилировать поэтапно, что делает разработку интерактивной, динамической и быстрой. Запускаемые программы могут изменяться, отлаживаться и расти постепенно.

Макросы компиляции

- Макросы компиляции определяют альтернативные стратегии для компиляции функции или макроса. В отличие от обычных макросов, макрос компиляции не расширяет синтаксис языка и может быть применён только во время компиляции. Поэтому они в основном используются для того, чтобы определить способы оптимизации кода отдельно от самого кода.

Определения типов

- Хотя лисп и является динамически типизированным языком — что довольно удобно при быстром прототипировании — программист может явно указать типы переменных. Это, а также другие директивы, позволяют компилятору оптимизировать код, как будто бы язык был статически типизированным.
- Например, мы можем определить типы параметров в нашей функции EXPLODE, вот так:

```
(defun explode (string &optional (delimiter #\Space))
 (declare (type character delimiter)
    (type string string)) ...)
```

Программируемый парсер

- Парсер лиспа позволяет легко разбирать входные данные. Он получает текст из входного потока и создаёт лисповские объекты, которые обычно называют S-выражениями. Это очень сильно упрощает разбор входных данных.
- Парсер можно использовать посредством нескольких функций, таких как READ, READ-CHAR, READ-LINE, READ-FROM-STRING и т.д. Входной поток может быть файлом, вводом с клавиатуры и так далее, но, кроме того, мы можем читать данные из строк или последовательностей символов при помощи соответствующих функций.
- Вот простейший пример чтения при помощи READ-FROM-STRING, который создаёт объект (400 500 600), то есть список, из строки "(400 500 600)".

➢ (read-from-string "(400 500 600)")

(400 500 600)

13

> (type-of (read-from-string "t"))

BOOLEAN

- **Макросы чтения** (reader macros) позволяют определить специальную семантику для заданного синтаксиса. Это возможно потому, что парсер лиспа является программируемым. Макросы чтения — это ещё один способ расширить синтаксис языка (они обычно используются, чтобы добавить синтаксический сахар).
- Некоторые стандартные макросы чтения:
  - #'foo — функции,
  - #\\ — символы (characters),
  - #c(4 3) — комплексные числа,
  - #p"/path/" — пути к файлам.
- Парсер может сгенерировать любой объект, для которого определены правила чтения; в частности, эти правила можно задать при помощи макросов чтения. На самом деле парсер, о котором идёт речь, используется и для интерактивных интерпретаторов (read-eval-print loop, REPL).
- Вот так мы можем прочитать число в шестнадцатеричной записи при помощи стандартного макроса чтения:

➢ (read-from-string "#xBB")

Программируемая печать

- Система текстового вывода в лиспе предоставляет возможности для печати структур, объектов или каких-либо ещё данных в разном виде.

  PRINT-OBJECT — это встроенная обобщённая функция, которая принимает в качестве аргументов объект и поток, и соответствующий метод выводит в поток текстовое представление данного объекта. В любом случае, когда нужно текстовое представление объекта, используется эта функция, в том числе в FORMAT, PRINT и в REPL.

  Рассмотрим класс JOURNEY:

```
(defclass journey ()
 ((%from :initarg :from :accessor from)
 (%to :initarg :to :accessor to)
 (%period :initarg :period :accessor period)
 (%mode :initarg :mode :accessor mode)))
```

- Если мы попытаемся распечатать объект класса JOURNEY, мы увидим нечто подобное:

```
> (defvar *journey*
  (make-instance 'journey
  :from "Christchurch" :to "Dunedin«
  :period 20 :mode "bicycle"))
*JOURNEY*
> (format nil "~a" *journey*)
"#<JOURNEY {10044DCCA1}>"
```

- Можно определить метод PRINT-OBJECT для класса JOURNEY, и с его помощью задать какое-то текстовое представление объекта:

```
(defmethod print-object ((j journey) (s stream))
 (format s "~A to ~A (~A hours) by ~A.«
 (from j) (to j) (period j) (mode j)))
```

Наш объект теперь будет использовать новое текстовое представление:

```
> (format nil "~a" *journey*)
"Christchurch to Dunedin (20 hours) by bicycle."
```

Пример: вычисление факториала

- На классическом Lisp

```
(defun factorial (n)
 (if (<= n 1) 1
 (* n (factorial (- n 1)))))
```

- На Common Lisp

```
(defun factorial (n)
 (loop for i from 1 to n
    for fac = 1 then
 (* fac i)
 finally (return fac)))
```

Scheme

- Более близок к классическому Lisp, чем Common Lisp

Пример: вычисление факториала

- На классическом Lisp

```
(defun factorial (n)
 (if (<= n 1) 1
 (* n (factorial (- n 1)))))
```

- На Scheme

```
(define (factorial n)
 (if (= n 0) 1
 (* n (factorial (- n 1)))))
```