3 Элементы императивного программирования

3.1 Побочный эффект

Несмотря на то, что LISP считается языком функционального программирования в нем заложены элементы императивного стиля.

Императивная парадигма программирования рассматривает компьютер как устройство с множеством состояний, а каждая команда программы изменяет это состояние. Чисто функциональная программа предполагает получение результата без каких либо воздействий на окружающую среду (в состав которой входит и компьютер). Такие воздействия называются «побочными эффектами».

«Побочный эффект» — изменение состояния окружающей среды в следствие выполнения функции.

Естественно, что никакая программа, в которой ведется диалог с пользователем, не обходится без побочного эффекта, т. к. такая программа как минимум изменяет состояние видеопамяти. При чтении данного из потока или при записи в поток изменяется состояние потока.

Побочный эффект функции оправдан, если результат побочного эффекта повторно используется в дальнейшем, т. е. при реализации последовательных действий.

Другое оправданное использование побочного эффекта возможно в целях экономии памяти. Так как результат выполнения любой функции без побочного эффекта есть новый лисповский объект, то в некоторых случаях (например в случае больших объемов памяти отводимой под новый объект) целесообразнее получить его путем деструктивного видоизменения некоторого старого объекта (возможно аргумента функции), если дальнейшее использование последнего не предполагается.

3.2 Реализация базовых конструкций императивного программирования

Основными отличительными чертами императивного программирования является наличие:

1. оператора присваивания;
2. возможности последовательного выполнения команд;
3. операторов ветвления и циклов;
4. операторов безусловного перехода.

В LISP присутствует реализация всех этих пунктов. Мы рассмотрим только некоторые из них.

3.2.1 Функции присваивания

Есть набор функций с побочным эффектом, которые можно использовать в качестве операторов присваивания.

Функция `set` принимает два аргумента, причем значением первого аргумента должно быть символьное имя (элемент типа SYMBOL). Функция `set` вычисляет оба своих аргумента и результату вычисления первого аргумента «присваивает» результат вычисления второго. Т. е. действие (и побочный эффект) функции `set` выходит за рамки представления об операторе присваивания традиционного языка императивного программирования, где «адресат» присваиваемого значения не вычисляется а задается непосредственно. Своим результатом функция `set` возвращает значение второго аргумента. В следующем вызове с именем `a` связывается значение 25.

```
* (set (car '(a b c)) 26)

26
* a

26
```

Если необходимо с помощью set связать символьное имя с некоторым значением, то вычисление этого имени необходимо заблокировать с помощью функции `quote`:

```
* (set 'a 28)

28
* a

28
```

Функция `setq` не вычисляет первый аргумент, который должен быть символьным именем, и связывает с ним значение второго аргумента, т. е. работает так же как традиционный оператор присваивания.

```
* (setq a 28)

28
* a

28
```

Макрос `setf` расширяет возможности функции `setq`. Значение указателя на результат вычисления первого аргумента этот макрос меняет на ссылку на значение выражения второго аргумента.

Например, после вычисления

```
* (setq L '(1 2 3 4 5))

(1 2 3 4 5)
```

значением вычисления `L` становится указанный список. А вызов

```
* (setf (car L) 8)

8
```

приводит к тому, что результатом вычисления `(car L)` становится значение `8`. Побочным эффектом такого присваивания является изменение значения всего списка `L`:

```
* L

(8 2 3 4 5)
```

Для варианта присваиваний, при которых значение переменной увеличивается/уменьшается на определенную величину имеются функции инкремента/декремента `incf` и `decf`. Вместо использования

```
(setf a (+ a b))
```

можно использовать

```
(incf a b)
```

(аналогично для `decf` и вычитания).

Если второй аргумент у `incf`/`decf` опущен, то он предполагается равным единице.

```
* (setq L '(1 2 3 4))

(1 2 3 4)
* (incf (cadr L))

3
* L

(1 3 3 4)
```

3.2.2 Организация последовательных вычислений

Основной механизм последовательных вычислений в LISP предоставляется конструкцией `progn`. Конструкция принимает любое количество параметров, последовательно вычисляет их значения и выдает значение последнего вычисленного параметра в качестве результата.

```
* (progn (setq a 25)
         (setq b (+ a 35))
         (setq c (* a b))
         (print c)
         (- c b))


1500
1440
```

Механизм функции `progn` неявно присутствует во многих лисповских конструкциях. Так, среди уже рассмотренных конструкций, в функции `lambda`, в конструкциях `defun`, `let`, `let*`, `labels`: тело может состоять из нескольких выражений, составляющих неявный `progn`. В конструкции `cond` в каждой условной паре после условия может присутствовать несколько выражений, обрабатываемых неявной конструкцией `progn`.

4 Символьные вычисления

Функции, описанные в следующих двух пунктах, позволяют реализовать вычисления, при которых программа на LISP генерирует и выполняет другую программу (фрагмент программы) на LISP.

4.1 Функция `eval`

В дополнение к функции `quote`, блокирующей вычисление S-выражения, LISP также предоставляет в распоряжение пользователя функцию `eval`, позволяющую оценить S-выражение.

Она зависит от одного аргумента, представляющего собой S-выражение. Этот аргумент оценивается как обычный аргумент функции. Однако результат снова оценивается, и в качестве значения выражения `eval` возвращается окончательный результат.

Например, можно построить S-выражение

```
* ((list 'cons '(quote +) '(quote (1 2 3 4)))

(CONS (quote +) (quote (1 2 3 4)))
```

Применяя функцию `eval` можно оценить значение этого выражения.

```
* (eval (list 'cons '(quote +) '(quote (1 2 3 4))))

(+ 1 2 3 4)
```

а если применить функцию `eval` к результату, то можно оценить и его значение:

```
* (eval (eval (list 'cons '(quote +) '(quote (1 2 3 4)))))

10
```

Можно сказать, что функция `eval` является обратной функцией к функции `quote`, хотя лучше сказать, что функция `eval` отменяет результат выполнения функции `quote`.

```
* (eval (quote (+ 2 3)))

5
```

4.2 `apply` и `funcall`

Функции `apply` и `funcall` применяют функцию, заданную первым аргументом, к заданному набору параметров. В функции `apply` набор параметров передается в виде списка, а в функции `funcall` — как параметры самой функции `funcall`.

Например, для применения функции `+` к числам 25, 17, 39, 40 можно следующими способами:

```
* (apply '+ '(25 17 39 40))

121
* (funcall '+ 25 17 39 40)

121
```

Обычно, нет смысла применять `apply` или `funcall` если вызываемая функция заранее определена. Надобность в этих функциях возникает, когда необходимо применить функцию, определяемую «на лету» (например, неименованную функцию) и/или сохраненной в какой-то переменной.

```
* (let ((f-name '+))
    (apply f-name '(25 17 39 40)))

121
* (let ((func (lambda (x y) (+ (* x x) (* y y)))))
    (funcall func 3 5))

34
```

5 Макросы

Макросы можно считать средством определения в LISP новых синтаксических конструкций.

Макросы на определяются с помощью конструкции `defmacro`:

```
(defmacro my-if (cond &optional then else)
  (list 'cond (list cond then) (list T else)))
```

Как и при определении функции при определении макроса указывается имя, список параметров и тело, состоящее из S-выражений. Но отличия макроса от функции становятся очевидными при его запуске. Если бы мы определили функцию

```
(defun my-if (cond &optional then else)
  (list 'cond (list cond then) (list T else)))
```

то ее работа заключалась бы в конструировании списка, на первом месте которого стоит атом `cond` за которым следуют два подсписка со значениями аргументов.

```
* (my-if (< 25 60) (cons 'a '(b c d)) (list 1 2 3))

(COND (T (A B C D)) (T (1 2 3)))
```

В случае определения макроса значение выражения существенно меняется:

```
* (my-if (< 25 60) (cons 'a '(b c d)) (list 1 2 3))

(A B C D)
```

На самом деле, работа макроса не ограничивается вычислением выражений, составляющих «тело» макроса. Рассмотрим работу макроса по-порядку.

Прежде всего отметим, что при запуске макроса его аргументы не вычисляются, т. е. когда тело макроса в нашем примере начинает оцениваться, значениями параметров `cond`, `then` и `else` являются не значения `T`, `(A B C D)` и `(1 2 3)` (как при начале оценивания тела функции), а списки `(< 25 60)`, `(cons 'a '(b c d))` и `(list 1 2 3)`. Таким образом, вычисление тела макроса приведет к получению S-выражения

```
(COND ((< 25 60) (CONS 'A '(B C D))) (T (LIST 1 2 3)))
```

Это выражение заменяет собой вызов макроса и его оценка приводит к получению результата вызова.

```
* (COND ((< 25 60) (CONS 'A '(B C D))) (T (LIST 1 2 3)))

(A B C D)
```

Вследствие того, что аргументы макроса не вычисляются при вызове, некоторые из них могут не вычисляться совсем. Так, в примере, значение третьего аргумента не вычисляется (т. к. вычисление второго аргумента приводит к получению значения `cond`).

Тело макроса конструирует S-выражение — списочную структуру. Для более лаконичного описания таких структур в LISPe предлагается дополнительные средства. В частности, можно использовать обратный апостроф «‘» для блокировки вычислений списка:

```
* ‘(1 2 3 4 5)

(1 2 3 4 5)
```

Такое применение напоминает применение функции `quote` (обычный апостроф). Но есть существенная разница. При использовании обратного апострофа не все вычисления блокируются. Если внутри такого «блокированного» вычисления перед S-выражением стоит символ «`,`» (запятая), то это выражение вычисляется и результат вычислений встает на место S-выражения:

```
* ‘(1 2 ,(cons 2 ’(1 2 3)) 4 5)

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

Если внутри «блокированного» вычисления перед S-выражением стоит «`,@`» (запятая с собакой), то это выражение вычисляется, результат вычислений должен быть списком, и элементы этого списка последовательно занимают место исходного S-выражения:

```
* ‘(1 2 ,@(cons 2 ’(1 2 3)) 4 5)

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

Так, описанный выше пример можно переписать в следующем виде

```
(defmacro my-if (cond &optional then else)
  `(cond (,cond ,then)
        (T ,else)))
```

Такое описание может казаться более ясным по сравнению с исходным описанием этого-же макроса.

Описание параметров макроса несколько отличается от описания параметров функции. Во-первых, к тем ключевым, которые можно использовать при описании функции, добавляется ключевое слово `&body`, которое по сути является аналогом `&rest`.

Во-вторых, в описание параметров макроса можно вносить дополнительную структуру, в которой аргументы макроса могут объединяться в дополнительные списки. Такое объединение указывает, что при вызове макроса соответствующие параметры должны представлять такую-же структуру.

Например, определим немного странный макрос:

```
(defmacro if-satisfy ((a &key (comp #'=))
                      (b &body body1)
                      &body body2)
   `(if (funcall ,comp ,a ,b)
        (progn ,@body1)
        (progn ,@body2)))
```

Данный макрос имеет четыре обязательных параметра (`a`, `b`, `body1`, `body2`) и один ключевой (`comp`). В результате работы макроса происходит сравнение параметра `a` и `b` с помощью функции, имя которой задается в `comp` (по умолчанию — функция сравнения чисел на равенство), и если результат не равен `NIL`, то выполняется последовательность выражений `body1`, а иначе выполняется последовательность выражений `body2`. При этом, аргументы `a` и `comp` объединяются с помощью дополнительных скобок. То же самое происходит с аргументами `b` и `body1`. Так, например, возможен вызов

```
* (if-satisfy (17)
    (17 (print "Line 1")
        (print "Line 2"))
    (print "Line 3")
    (print "Line 4"))

"Line 1"
"Line 2"
"Line 2"
```

Здесь в качестве `a` передается `17`, в качестве `b` — `17`. Значение `comp` берется по умолчанию, поэтому в первых скобках только значение `a`. Во вторых скобках на первом месте стоит значение `b` (как при описании параметров макроса), за которым до конца этого списка следуют элементы `body1`. Последние два параметра в вызове `if-satisfy` — элементы списка body2. Т. к. значения `a` и `b` равны, оцениваются все составляющие `body1`, и результат последней оценки (`"Line 2"`) выдается как результат макроса.

Подобный вызов, но в котором указывается значение `comp`:

```
* (if-satisfy ('(a b c) :comp 'equal)
    ((cons 'a '(b c)) (print "Line 1")
                      (print "Line 2"))
    (print "Line 3")
    (print "Line 4"))

"Line 1"
"Line 2"
"Line 2"
```

Функция `macroexpand-1` получает в качестве аргумента S-выражение, представляющее собой вызов макроса. Функция показывает результат «раскрытия» указанного вызова макроса. Т. е. `macroexpand-1` производит выполнение «тела» макроса без оценивания результата. Например, для вызова макроса `my-if`:

```
* (macroexpand-1 '(my-if (< 25 60) (cons 'a '(b c d)) (list 1 2 3)))

(COND ((< 25 60) (CONS 'A '(B C D))) (T (LIST 1 2 3)))
T
```

или для последнего вызова макроса `if-satisfy`

```
* (macroexpand-1 '(if-satisfy ('(a b c) :comp 'equal)
                    ((cons 'a '(b c)) (print "Line 1")
                                      (print "Line 2"))
                    (print "Line 3")
                    (print "Line 4")))

(IF (FUNCALL 'EQUAL '(A B C) (CONS 'A '(B C)))
    (PROGN (PRINT "Line 1") (PRINT "Line 2"))
    (PROGN (PRINT "Line 3") (PRINT "Line 4")))
T
```