Техники функционального программирования

Лектор: Артамонов Юрий Николаевич

Университет "Дубна"
филиал Котельники

Понятие рекурсивных процессов

В большинстве случаев рекурсивное представление алгоритмов обладает большей ясностью и наглядностью, код получается более кратким, представленным в виде независимых модулей. Однако из-за возникающих рекурсивных процессов за это приходится иногда платить так называемой вычислительной сложностью. Возьмем пример вычисления факториала.

```
(factorial 6)
  (* 6 (factorial 5))
  (* 6 (* 5 (factorial 4)))
  (* 6 (* 5 (* 4 (factorial 3))))
  (* 6 (* 5 (* 4 (* 3 (factorial 2)))))
  (* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
  (* 6 (* 5 (* 4 (* 3 (* 2 1)))))
  (* 6 (* 5 (* 4 (* 3 2))))
  (* 6 (* 5 (* 4 6)))
  (* 6 (* 5 24))
  (* 6 120)
```

Линейный рекурсивный процесс

Подстановочная модель показывает сначала серию расширений, а затем сжатие. Расширение происходит по мере того, как процесс строит цепочку отложенных операций, в данном случае цепочку умножений. Сжатие происходит тогда, когда выполняются эти отложенные операции. Такой тип процесса, который характеризуется цепочкой отложенных операций, и называется **рекурсивным процессом**. Выполнение этого процесса требует, чтобы интерпретатор запоминал, какие операции ему нужно выполнить впоследствии. При вычислении n! длина цепочки отложенных умножений, а следовательно, и объем информации, который требуется, чтобы ее сохранить, растет линейно с ростом n (пропорционален n), как и число шагов. Такой процесс называется **линейно рекурсивным процессом**.

Накапливающий параметр, итеративный процесс

Рассмотрим следующую модификацию алгоритма вычисления факториала:

```
(defun factorial (n)
  (fact-iter 1 1 n))
(defun fact-iter (product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count)))
```

Подстановочная модель при накапливающем параметре

Получаем следующую подстановочную модель:

```
(factorial 6)
(fact-iter 1 1 6)
(fact-iter 1 2 6)
(fact iter 2 3 6)
(fact-iter 6 4 6)
(fact-iter 24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720
```

Этот процесс не растет и не сжимается. На каждом шаге при любом значении n необходимо помнить лишь текущие значения переменных product, counter и max-count. Такой процесс называется итеративным. Здесь переменная product называется накапливающим параметром.

Блочная структура в Common Lisp

Чтобы не плодить в коде вспомогательные одноразовые функции можно использовать блочную структуру — в Лиспе внутри определения функции можно определять любое количество функций, которые будут недоступны внешний функциям (при этом функции должны быть определены раньше до того, как они будут использоваться)

```
(defun factorial (n)
  (defun iter (product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

Итеративный процесс - определение

В общем случае, **итеративный процесс** — это такой процесс, состояние которого можно описать конечным числом переменных состояния плюс заранее заданное правило, определяющее, как эти переменные состояния изменяются от шага к шагу, и плюс (возможно) тест на завершение, который определяет условия, при которых процесс должен закончить работу. При вычислении n! число шагов линейно растет с ростом n. Такой процесс называется линейно итеративным процессом.

Различие между итеративным и рекурсивным процессами

Можно посмотреть на различие итеративного, рекурсивного процессов и с другой точки зрения. В итеративном случае в каждый момент переменные программы дают полное описание состояния процесса. Если мы остановим процесс между шагами, для продолжения вычислений нам будет достаточно дать интерпретатору значения трех переменных программы. С рекурсивным процессом это не так. В этом случае имеется дополнительная «спрятанная» информация, которую хранит интерпретатор и которая не содержится в переменных программы. Она указывает, «где находится» процесс в терминах цепочки отложенных операций. Чем длиннее цепочка, тем больше информации нужно хранить .

Рекурсивная процедура, итеративный процесс

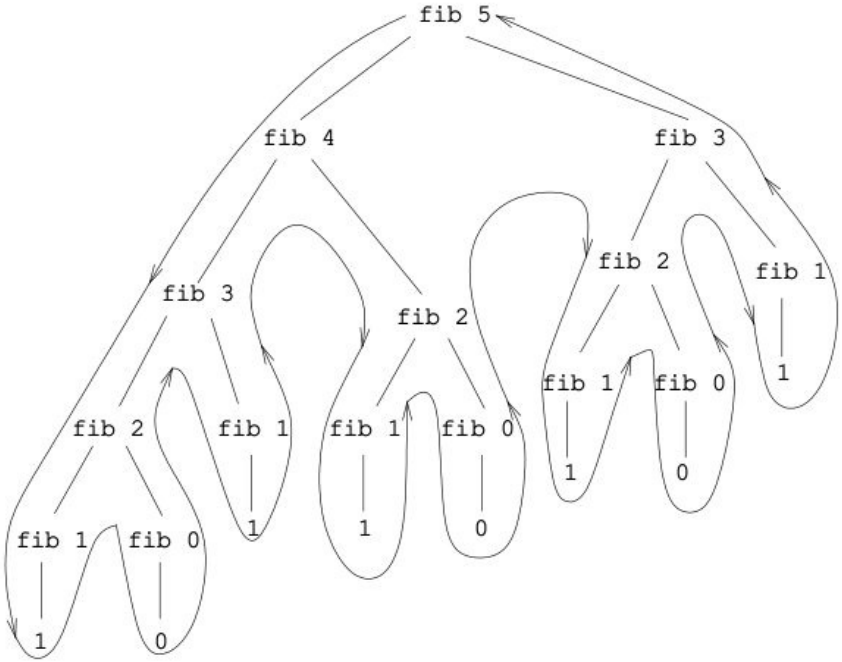
Противопоставляя итерацию и рекурсию, нужно вести себя осторожно и не смешивать понятие рекурсивного процесса с понятием рекурсивной процедуры. Когда мы говорим, что «процедура рекурсивна», мы имеем в виду факт синтаксиса: определение процедуры ссылается (прямо или косвенно) на саму эту процедуру. Когда же мы говорим о процессе, что он следует, скажем, линейно рекурсивной схеме, мы говорим о развитии процесса, а не о синтаксисе, с помощью которого написана процедура. Может показаться странным, например, высказывание «рекурсивная процедура fact-iter описывает итеративный процесс». Однако процесс действительно является итеративным: его состояние полностью описывается тремя переменными состояния, и чтобы выполнить этот процесс, интерпретатор должен хранить значение только трех переменных.

Хвостовая рекурсия

Различие между процессами и процедурами может запутывать отчасти потому, что большинство реализаций обычных языков (например, Паскаль и Си) построены так, что интерпретация любой рекурсивной процедуры поглощает объем памяти, линейно растущий пропорционально количеству вызовов процедуры, даже если описываемый ею процесс в принципе итеративен. Как следствие, эти языки способны описывать итеративные процессы только с помощью специальных «циклических конструкций» вроде do, repeat, until, for и while. Или приходится самому использовать технику накапливающего параметра. Большинство реализаций Лисп свободны от этого недостатка. Они сами определяют ситуацию, когда рекурсивная процедура может порождать итерационный процесс. Такое свойство реализации языка называется **поддержкой хвостовой рекурсии** (реализация Лисп - Scheme поддерживает хвостовую рекурсию на уровне стандарта, в стандарте Common Lisp такого нет, SBCL ее поддерживает, CLISP - нет). Если реализация языка поддерживает хвостовую рекурсию, то итерацию можно выразить с помощью обыкновенного

Древовидно-рекурсивный процесс

Кроме линейно-рекурсивного процесса, существует еще древовидно-рекурсивный процесс, который возникает, например, для процедур, использующих параллельную рекурсию. Для примера рассмотрим подстановочную модель, возникающую при вычислении чисел Фибоначчи, которую мы рассматривали ранее:



Анализ древовидно-рекурсивного процесса

Таким образом, число шагов нашего процесса растет экспоненциально при увеличении аргумента. С другой стороны, требования к памяти растут при увеличении аргумента всего лишь линейно, поскольку в каждой точке вычисления нам требуется запоминать только те вершины, которые находятся выше нас по дереву. В общем случае число шагов, требуемых древовидно-рекурсивным процессом, будет пропорционально числу вершин дерева, а требуемый объем памяти будет пропорционален максимальной глубине дерева. Попробуем и этот рекурсивный процесс преобразовать в итеративный с помощью техники накапливающего параметра. Идея состоит в том, чтобы использовать пару целых a и b, которым в начале даются значения $\mathrm{Fib}(1) = 1$ и $\mathrm{Fib}(0) = 0$, и на каждом шаге применять одновременную трансформацию

$a \leftarrow a+b$

$b \leftarrow a$

Нетрудно показать, что после того, как мы проделаем эту трансформацию n раз, a и b будут соответственно равны $\mathrm{Fib}(n + 1)$ и $\mathrm{Fib}(n)$.

Итеративное вычисление чисел Фибоначчи

Таким образом, мы можем итеративно вычислять числа Фибоначчи при помощи процедуры:

```
(defun fib (n)
  (fib-iter 1 0 n))
(defun (fib-iter a b count)
  (if (= count 0)
      b
      (fib-iter (+ a b) a (- count 1))))
```

Этот метод вычисления чисел Фибоначчи представляет собой линейную итерацию. Разница в числе шагов, требуемых двум этим методам, — один пропорционален n, другой растет так же быстро, как и само Fib(n), — огромна, даже для небольших значений аргумента.

Нужны ли древовидно-рекурсивные процессы?

Не нужно из этого делать вывод, что древовидно-рекурсивные процессы бесполезны. Если мы будем рассматривать процессы, работающие не с числами, а с иерархически структурированными данными, мы увидим, что древовидная рекурсия является естественным и мощным инструментом. Но даже при работе с числами древовидно-рекурсивные процессы могут быть полезны — они помогают нам понимать и проектировать программы. Например, хотя рекурсивная процедура fib без накапливающего параметра и намного менее эффективна, чем с ним, зато она проще, поскольку это немногим более, чем перевод определения последовательности чисел Фибоначчи на Лисп. Чтобы сформулировать итеративный алгоритм, нам пришлось заметить, что вычисление можно перестроить в виде итерации с тремя переменными состояния.

Пример пользы древовидно-рекурсивных процессов

Чтобы сочинить итеративный алгоритм для чисел Фибоначчи, нужно совсем немного смекалки. Теперь для контраста рассмотрим следующую задачу: сколькими способами можно разменять сумму в 1 доллар, если имеются монеты по 50, 25, 10, 5 и 1 цент?
В более общем случае, можно ли написать процедуру подсчета способов размена для произвольной суммы денег?
У этой задачи есть простое решение в виде рекурсивной процедуры.

Задача размена суммы

Предположим, мы как-то упорядочили типы монет, которые у нас есть. В таком случае будет верно следующее соотношение.
Число способов разменять сумму a с помощью n типов монет равняется:

- числу способов разменять сумму a с помощью всех типов монет, кроме первого, плюс
- число способов разменять сумму a − d с использованием всех n типов монет, где d — достоинство монет первого типа.

Задача размена суммы

Чтобы увидеть, что это именно так, заметим, что способы размена могут быть поделены на две группы: те, которые не используют первый тип монеты, и те, которые его используют. Следовательно, общее число способов размена какой-либо суммы равно числу способов разменять эту сумму без привлечения монет первого типа плюс число способов размена в предположении, что мы этот тип используем. Но последнее число равно числу способов размена для суммы, которая остается после того, как мы один раз употребили первый тип монеты. Таким образом, мы можем рекурсивно свести задачу размена данной суммы к задаче размена меньших сумм с помощью меньшего количества типов монет.

База рекурсии в задаче размена суммы

Внимательно рассмотрите это правило редукции и убедите себя, что мы можем использовать его для описания алгоритма, если укажем следующие вырожденные случаи (базы рекурсии):

- если a в точности равно 0, мы считаем, что имеем 1 способ размена.
- если a меньше 0, мы считаем, что имеем 0 способов размена.
- если n равно 0, мы считаем, что имеем 0 способов размена.

Это описание легко перевести в рекурсивную процедуру.

Код в задаче размена суммы

```
(defun count-change (amount)
  (cc amount 5))
(defun cc (amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (t (+ (cc amount  (- kinds-of-coins 1))
              (cc (- amount
                     (first-denomination
                       kinds-of-coins))
                  kinds-of-coins)))))
(defun first-denomination (kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

Тестирование кода в задаче размена суммы

Процедура first-denomination принимает в качестве входа номер типа монет и возвращает их достоинство. Здесь мы упорядочили монеты от самой крупной к более мелким, но годился бы и любой другой порядок. Теперь мы можем ответить на исходный вопрос о размене доллара:

```
>(count-change 100)
>292
```

Count-change порождает древовидно-рекурсивный процесс с избыточностью, похожей на ту, которая возникает в нашей первой реализации fib. (На то, чтобы получить ответ 292, уйдет заметное время.) С другой стороны, неочевидно, как построить более эффективный алгоритм для получения этого результата.

Исследование хвостовой рекурсии

Для сравнения будем использовать встроенную фукцию Лисп time:
(time <exp>)
которая возвращает время и объем памяти, затраченный на вычисление выражения <exp>. Все оценки ресурсных затрат рассматриваются в дистрибутивах Лисп - SBCL 1.2.4, GNU CLISP 2.49 и могут не совпадать для других дистрибутивов.

Пример 1

Реализовать функцию expt1, позволяющую возвести число A в заданную натуральную степень N:

```
(defun expt1 (A N)
        (cond
                ((= N 0) 1)
                (t (* A (expt1 A (- N 1))))))
* (time (expt1 5 100))
```

Оценки затрат для примера 1 в разных дистрибутивах

SBCL - все хорошо!

```
Evaluation took:
  0.000 seconds of real time
  0.000000 seconds of total run time
  (0.000000 user, 0.000000 system)
  100.00% CPU
  20,369 processor cycles
  0 bytes consed
```

CLISP - появляются временные затраты.

```
Real time: 4.43E-4 sec.
Run time: 0.0 sec.
Space: 2688 Bytes
```

Улучшение примера 1 в блочной конструкции

Если рассмотреть подстановочную модель, легко увидеть рекурсивный процесс — вначале происходит расширение, потом сжатие. Преобразуем теперь вычисления в итерационный процесс, при этом будем использовать блочную структуру

```
(defun expt2 (A N)
   (defun iter (result count)
                (cond  ((> count N) result)
                       (t (iter (* A result)
                       (+ count 1)))))
        (iter 1 1))
```

Внутри функции expt2 мы определяем вспомогательную функцию iter с двумя аргументами result — накапливающий параметр, аккумулирует результат, count — счетчик, как только он превысит показатель степени N, result выдается в качестве результата. Само тело функции expt2 состоит из вызова вспомогательной функции iter от аргументов result=1, count=1. Вся эта конструкция похожа на цикл императивных языков программирования.

Сравнение улучшений для примера 1

SBCL - опять все хорошо

```
(time (expt2 5 100))
(Evaluation took:
  0.000 seconds of real time
  0.000000 seconds of total run time
  (0.000000 user, 0.000000 system)
  100.00% CPU
  78,136 processor cycles
  0 bytes consed
```

CLISP - уже лучше, но с более высоким расходом памяти

```
Real time: 1.72E-4 sec.
Run time: 0.0 sec.
Space: 3416 Bytes
```

Улучшение примера 1 - реализация без блочной структуры

Попробуем убрать блочную структуру:

```
(defun iter1 (result count A N)
       (cond
             ((> count N) result)
             (t (iter1 (* A result)
             (+ count 1 ) A N))))
(defun expt3 (A N)
       (iter1 1 1 A N))
```

Оценка затрат в безблочной конструкции

SBCL - лучшая реализация по количеству циклов процессора (правда - это некоторая случайная величина)

```
Evaluation took:
  0.000 seconds of real time
  0.000000 seconds of total run time
  (0.000000 user, 0.000000 system)
  100.00% CPU
  11,175 processor cycles
  0 bytes consed
```

CLISP - наши попытки тщетны (памяти чуть меньше, времени больше).

```
(time (expt3 5 100))
Real time: 3.04E-4 sec.
Run time: 0.0 sec.
Space: 2688 Bytes
```

Оценка затрат для встроенной функции

SBCL - эта самая лучшая!

```
Evaluation took:
  0.000 seconds of real time
  0.000000 seconds of total run time
  (0.000000 user, 0.000000 system)
  100.00% CPU
  1,471 processor cycles
  0 bytes consed
```

CLISP - лучше и по времени и по памяти.

```
Real time: 2.4E-5 sec.
Run time: 0.0 sec.
Space: 128 Bytes
```

Использование встроенной функции expt эффективнее как по времени, так и объему памяти, что объясняется скомпилированным и оптимизированным кодом внутреннего представления этой функции.

Разница - наличие, отсутствие хвостовой рекурсии

Причина отличий SBCL, CLISP объясняется просто — интерпретатор Лиспа SBCL умеет оптимизировать код, самостоятельно распознавая хвостовую рекурсию, преобразуя ее в итерационный процесс, где это возможно. Но распознавать хвостовую рекурсию интерпретатор может только в простых конструкциях и надеяться на него не стоит.

Еще одно улучшение

В качестве замечания отметим, что можно вычислять степени за меньшее число шагов, если использовать последовательное возведение в квадрат. Например, вместо того, чтобы вычислять a^8 в виде

$$a \cdot (a \cdot (a \cdot (a \cdot (a \cdot (a \cdot a)))))$$

мы можем вычислить его за три умножения:

$$a^2 = a \cdot a$$
$$a^4 = a^2 \cdot a^2$$
$$a^8 = a^4 \cdot a^4$$

Этот метод хорошо работает для степеней, которые сами являются степенями двойки. В общем случае при вычислении степеней мы можем получить преимущество от последовательного возведения в квадрат, если воспользуемся правилом:

- $a^n = (a^{n/2})^2$, если n четно
- $a^n = a \cdot a^{n-1}$, если n нечетно

Еще одно улучшение - реализация

Этот метод можно выразить в виде процедуры

```
(defun fast-expt (A N)
  (cond ((= N 0) 1)
  ((= (mod N 2) 0)
  (expt (fast-expt A (/ N 2)) 2))
        (t (* A (fast-expt A (- N 1))))))
```

Несмотря на то, что получаемый от этой процедуры процесс будет рекурсивным - экономия вычислительных ресурсов присутствует. При желании можно было бы преобразовать процедуру fast-expt так, чтобы получался итерационный процесс.

Еще одно улучшение - оценки

```
SBCL
  (time (fast-expt 5 100))
(0.000 seconds of real time
  0.000000 seconds of total run time
  (0.000000 user, 0.000000 system)
  100.00% CPU
  28,881 processor cycles
  0 bytes consed
CLISP
Real time: 4.6E-5 sec.
Run time: 0.0 sec.
Space: 128 Bytes
```

Пример 2

Определим две функции H и V над дробью $\frac{p}{q}$ следующим образом:

$$H(\frac{p}{q}) = 1 + \frac{p}{q} = \frac{(p+q)}{q};$$

$$V(\frac{p}{q}) = \frac{\frac{p}{q}}{\frac{p}{q}+1} = \frac{p}{p+q}.$$

Необходимо реализовать процедуду, которая например находит дробь для следующей последовательности:

$$V(V(V(H(V(V(H(H(H(V(V(V(V(H(V(V(H(H(H(V(1)))))))))))))))))))$$

Для простоты будем опускать скобки и единицу и писать:

$$VVVHVVHHHVVVVHVVHHHV$$

Пример 2 - демонстрация

Например, нужно найти дробь, соответствующую последовательности VVH. Имеем VVH=VVH(1)=VV(2/1)=V(2/3)=2/5.

Пример 2 - реализация 1

Данный алгоритм легко реализовать в виде процедуры, будем в списке L передавать последовательность H-V

```
(defun HV_To_Number (L)
  (cond
     ((and (Null (cdr L)) (eq (car L) 'H)) 2)
     ((and (Null (cdr L)) (eq (car L) 'V)) (/ 1 2))
     ((eq (car L) 'H) (+ 1 (HV_To_Number(cdr L))))
     ((eq (car L) 'V) (/ (HV_To_Number (cdr L))
                         (+ 1 (HV_To_Number(cdr L)))))))
```

Заметим, что у нас нет необходимости, в отличие от многих других языков программирования, заботиться о том, чтобы результат представлялся в виде дроби. Лисп делает эти преобразования автоматически.

Пример 2 - оценка

Легко увидеть, что получается рекурсивный процесс. Оценим затраты вычислительных ресурсов:

SBCL

```
  (time (HV_To_Number
  '(V V V H V V H H H V V V V H V V H H H V)))
Evaluation took: 0.001 seconds of real time
  0.000000 seconds of total run time
  (0.000000 user, 0.000000 system)
  0.00% CPU
  1,529,047 processor cycles
  360,432 bytes consed
```

CLISP

```
Real time: 0.022102 sec.
Run time: 0.02 sec.
Space: 478368 Bytes
1149/4249
```

Пример 2 - оценка другой последовательности

Удвоим первоначальную последовательность и оценим затраты вычислительных ресурсов. Если у вас хватит терпения, то получится следующий результат: (V V V H V V H H H V V V V H V V H H H V V V V V H V V H H H V V V V V H V V H H H V)))

SBCL

```
Evaluation took:
2.686 seconds of real time
2.644000 seconds of total run time
(2.592000 user, 0.052000 system)
[ Run times consist of 0.040 seconds GC time,
  and 2.604 seconds non-GC time. ]
98.44% CPU
6,434,637,989 processor cycles
1 page fault
1,423,113,824 bytes consed
2870201/10614001
```

Пример 2 - оценка другой последовательности

CLISP

```
Real time: 93.94243 sec.
Run time: 93.712 sec.
Space: 1960004768 Bytes
GC: 2235, GC time: 5.652 sec.
2870201/10614001
```

Это возмутительно много (особенно для CLISP)! Попробуйте преобразовать это в итеративный процесс!!!