Теоретические основы функционального программирования - лямбда-исчисление

Лектор: Артамонов Юрий Николаевич

Университет "Дубна"
филиал Котельники

Знакомство с синтаксисом

Лямбда-исчисление (λ-исчисление, ламбда-исчисление) — формальная система разработанная американским математиком Алонзо Чёрчем для формализации и анализа понятия вычислимости.
λ-исчисление может рассматриваться как семейство языков программирования. Их основная особенность состоит в том, что они являются языками *высших порядков*. Тем самым обеспечивается систематический подход к исследованию операторов, *аргументами* которых могут быть другие операторы, а значением также может быть оператор.
Первоначально λ-исчисление было использовано как средство для описания (чисто синтаксического) свойств математических функций и эффективной обработки их в качестве правил. Как мы уже отмечали, функциональные программы построены из «чистых» функций, т. е. функций в математическом смысле, и поэтому нотация λ-исчисления особенно удобна для формального описания манипулирования функциями и даже в качестве промежуточного кода, в который можно транслировать исходную программу.

Знакомство с синтаксисом (продолжение)

В действительности функциональные языки программирования являются «синтаксическим сахаром», и часто представляют собой «улучшенные» версии нотации λ-исчисления в том смысле, что все функциональные программы можно преобразовать в эквивалентные λ-выражения.

Попросту говоря, λ-исчисление — это исчисление анонимных (безымянных) функций. Оно дает, во-первых, метод представления функций и, во-вторых, множество правил вывода для синтаксического преобразования функций.

Начнем с рассмотрения следующей простой исходной функции, которая удваивает значение своего аргумента:

$$\mathrm{double(x)} = 2 * \mathrm{x} ;$$

Чтобы представить эту функцию в нотации λ-исчисления, мы опускаем имя функции "double", делая ее анонимной. Она представляется в виде λ-выражения следующим образом:

$$\lambda\ \mathrm{x}.(*\ 2\ \mathrm{x}).$$

Знакомство с синтаксисом (продолжение)

В языке Лисп подобного можно добиться, используя специальное служебное слово Lambda:

$$(\text{lambda } (\text{x}) \ (* \ 2 \ \text{x})).$$

(поскольку это выражение не вычисляется, интерпретатор выдает ошибку)

Мы читаем символ λ как «функция от» и точку (.) как «которая возвращает». λ-выражения такой формы называются λ-абстракциями.

Символ x после λ называется связанной переменной абстракции и соответствует понятию формального параметра в традиционной процедуре или функции. Выражение справа от точки называется телом абстракции, и, подобно коду традиционной процедуры или функции, оно описывает, что нужно сделать с параметром, поступившим на вход функции.

Тело абстракции может быть любым допустимым λ-выражением, и поэтому оно также может содержать другую абстракцию, например:

$$\lambda \ \text{x}.\lambda \ \text{y}.(* \ (\ + \ \text{x} \ \text{y}) \ 2)$$

Это выражение читается как «функция от x, которая возвращает функцию от y, которая возвращает (x+y)*2».

Знакомство с синтаксисом (продолжение)

К функциям в λ-нотации применим карринг, поэтому все они имеют только один аргумент. Все абстракции включают только один символ λ и единственную связанную переменную. Описанный процесс превращения функций многих переменных в функцию одной переменной и называется *карринг*, в честь американского математика Хаскелла Карри.

Поговорим теперь о правилах вывода λ-исчисления. Простейший тип λ-выражения— это константа. Константы являются самоопределенными, т. е. их невозможно преобразовать в более простые выражения. Вычисление целой константы 5, например, дает ту же самую константу 5; то же самое относится и к остальным константам. Применение любой функции записывается в виде выражения для этой функции, за которым следуют выражения для ее аргументов. Например, выражение + 1 3 означает применение константы (встроенной арифметической функции +) к двум числам 1 и 3. Это выражение может быть преобразовано в число 4 с помощью соответствующего δ-правила для функции +, которое мы записываем следующим образом (такой процесс упрощения выражений называется редукцией):

$$+\ \mathbf{1}\ \mathbf{3} \rightarrow_{\delta} 4$$

Знакомство с синтаксисом (продолжение)

Вот еще пример:

$$(* \ (+ \ 1 \ 2) \ (- \ 4 \ 1)) \rightarrow_\delta (* \ (+ \ 1 \ 2) \ 3) \rightarrow_\delta (* \ 3 \ 3) \rightarrow_\delta 9$$

Более интересное правило редукции описывает, как применять λ-абстракции. Рассмотрим следующее применение:

$$(\lambda \ x. \ (* \ x \ x) \ ) \ 2.$$

Ситуация аналогична вызову процедуры или функции языка высокого уровня с фактическим параметром, замещающим ее формальный параметр. В λ-исчислении такая замена является чисто текстовой, и мы, таким образом, физически заменяем все вхождения связанной переменной в теле применяемой λ-абстракции на выражение аргумента, 2, получая в результате:

$$(\lambda \ x. \ (* \ x \ x)) \ 2 \rightarrow_\beta (* \ 2 \ 2).$$

Это так называемая β-редукция.

Знакомство с синтаксисом (продолжение)

В языке Лисп это выглядит так:

$$((\text{lambda } (\text{x}) \ (* \ \text{x} \ \text{x})) \ 2)$$

Вот еще пример:

$$((\lambda \ \text{x}.\lambda \ \text{y}. \ (+ \ \text{x} \ \text{y}) \ 7) \ 8) \rightarrow_\beta \ (\ \lambda \ \text{y}. \ (+ \ 7 \ \text{y}) \ 8) \rightarrow_\beta (+ \ 7 \ 8) \rightarrow_\delta 15.$$

Порядок редукций

Говорят, что λ-выражение находится в нормальной форме, если к нему нельзя применить никакое правило редукции. Другими словами, λ-выражение — в нормальной форме, если оно не содержит редексов. Нормальная форма, таким образом, соответствует понятию конца вычислений в традиционном программировании. Отсюда немедленно вытекает наивная схема вычислений:

while существует хотя бы один редекс
do преобразовать один из редексов
end.

Проблема, связанная с такой схемой, заключается в том, что в выражении может быть несколько редексов и непонятно, какой и них выбрать для преобразования. Чтобы увидеть, сколь важным может быть выбор, рассмотрим следующее выражение:

$$(\lambda\ x.\lambda\ y.y)\ ((\lambda\ z.z\ z)\ (\lambda\ z.z\ z))$$

Здесь два редекса:

$$(\lambda\ z.z\ z)(\lambda\ z.z\ z) \text{ и } (\ \lambda\ x.\ \lambda\ y.\ y)\ ((\ \lambda\ z.\ z\ z)(\lambda\ z.z\ z))$$

Порядок редукций (продолжение)

Выбрав первый из них, получим следующую цепочку редукций:

$$(\lambda\ \mathrm{z.z}\ \mathrm{z})(\lambda\ \mathrm{z.z}\ \mathrm{z}) \text{--}> {}_{\beta}\ (\lambda\ \mathrm{z.z}\ \mathrm{z})(\lambda\ \mathrm{z.z}\ \mathrm{z}) \text{--}> {}_{\beta}(\ \lambda\mathrm{,z.}\ \mathrm{z}\ \mathrm{z})(\lambda\ \mathrm{z.z}\ \mathrm{z})\ldots$$

которая никогда не закончится.

Выбрав второй, получим редукцию, которая заканчивается за один шаг:

$$(\lambda\ \mathrm{x}.\lambda\ \mathrm{y.y})\ ((\lambda\ \mathrm{z.z}\ \mathrm{z})\ (\lambda\ \mathrm{z.z}\ \mathrm{z})) \text{--}> {}_{\beta}\ \lambda\ \mathrm{y.y}$$

Эти рассуждения приводят нас к рассмотрению порядка редукций, определяющего в случае нескольких редексов, какой из них выбрать для преобразования.

В контексте функциональных языков и λ-исчисления существуют два важных порядка редукций:

- Аппликативный порядок редукций, который предписывает вначале преобразовывать самый левый из самых внутренних редексов.
- Нормальный порядок редукций, который предписывает вначале преобразовывать самый левый из самых внешних редексов.

Порядок редукций (продолжение)

Возвращаясь к предыдущему примеру, видим, что самым левым из самых внутренних редексов является $(\lambda\ z.z\ z)(\lambda\ z.z\ z)$, а самый левый из самых внешних $(\lambda\ x.\lambda\ y.y)\ ((\lambda\ z.z\ z)\ (\lambda\ z.z\ z))$.

В некотором смысле нормальный порядок редукций соответствует ленивым вычислениям, а аппликативный — энергичным.

В принципе можно предположить, что если вычисление закончится, то мы получим идентичные результаты при любом порядке редукций. Это весьма вольное допущение, но оно оказывается правильным.

Более того, возможно также доказать, что нормальный порядок редукций всегда приводит выражение к нормальной форме, если нормальная форма существует. Эти два результата представлены в виде двух теорем:

- **Теорема Черча—Россера.** Если выражение E может быть приведено двумя различными способами к двум нормальным формам, то эти нормальные формы являются алфавитно-эквивалентными (т.е. эквивалентными с точностью до переобозначения переменных).
- **Теорема стандартизации.** Если выражение E имеет нормальную форму, то редукция самого левого из самых внешних редексов на каждом этапе вычисления E гарантирует достижение этой нормальной формы (с точностью до алфавитной эквивалентности).

Порядок редукций (продолжение)

Рассмотрим еще пример β-редукции. Дано:

$$(\lambda\ \mathrm{f}.\lambda\ \mathrm{x}.\mathrm{f}\ 4\ \mathrm{x})(\lambda\ \mathrm{y}.\lambda\ \mathrm{x}.\ +\ \mathrm{x}\ \mathrm{y})\ 3$$

Шаг 1. Преобразуем единственный редекс, подставляя аргумент (λ y.λ x. + x y) вместо f в тело абстракции (λ f.λ x.f 4 x):

$$\text{-->}_{\beta}\ (\ \lambda\ \mathrm{x}.\ (\ \lambda\ \mathrm{y}.\ \lambda\ \mathrm{x}.\ +\mathrm{x}\ \mathrm{y})\ 4\ \mathrm{x})\ 3$$

Шаг 2. Произвольно выбрав редекс (один из двух возможных), подставляем аргумент 3 в тело абстракции λ x. (λ y. λ x. +x y) 4 x, но при этом оставляем без изменений внутреннюю абстракцию (потому, что там другой x)

$$\text{-->}_{\beta}\ (\ \lambda\ \mathrm{y}.\ \lambda\ \mathrm{x}.\ +\mathrm{x}\ \mathrm{y})\ 4\ 3$$

Порядок редукций (продолжение)

Шаг 3. Преобразуем единственный редекс, т. е.

$$\text{--}>_{\beta}\ (\ \lambda\ \text{x.}\ +\text{x}\ 4)\ 3$$

Шаг 4. Преобразуем единственный редекс

$$\text{--}>_{\beta}\ +3\ 4$$

Шаг 5. Преобразуем полученный редекс с помощью δ-правила:

$$+\ 3\ 4\text{-}>_{\delta} 7.$$

Важно заметить, что на шаге 2 у нас был выбор между двумя редексами и мы для преобразования произвольно взяли внешний редекс. Однако можно было бы применить β-редукцию и к внутреннему редексу, что привело бы к следующей цепочке преобразований:

$$(\lambda\ \text{f.}\lambda\ \text{x.f}\ 4\ \text{x})(\lambda\ \text{y.}\lambda\ \text{x.}\ +\ \text{x}\ \text{y})\ 3\text{--}>_{\beta}\ (\ \lambda\ \text{x.}\ (\ \lambda\ \text{y.}\ \lambda\ \text{x.}\ +\text{x}\ \text{y})\ 4\ \text{x})\ 3$$

$$\text{--}>_{\beta}\ (\ \lambda\ \text{x.}\ (\ \lambda\ \text{x.}\ +\text{x}\ 4)\ \text{x})\ 3$$

$$\text{--}>_{\beta\ \text{(опять произвольно выбираем редекс)}}\ (\ \lambda\ \text{x.}\ +\text{x}\ 4)\ 3$$

$$\text{--}>_{\beta}\ +\ 3\ 4\text{-}>_{\delta} 7$$

Этот путь дает такой же результат и поэтому кажется вполне разумным, но, как мы показали, в общем случае можно получить совершенно разное поведение для двух различных порядков выполнения редукций.

Выражение основных примитивов Лиспа в -исчислении

Фактически любое из рассмотренных нами ранее выражений Лиспа может быть выражено (хотя и в несколько необычной форме) средствами чистого λ-исчисления.

Когда мы записываем выражение вида

$$\text{if P then Q else R (или if P Q R)}$$

мы обычно представляем условный оператор как встроенную функцию, которая выбирает Q или R в зависимости от значения предиката P. Другими словами, предикат условного оператора можно рассматривать как функцию, которая сама выбирает одно из выражений Q или R. Выражение if PQR в этом случае транслируется в PQR с правилами преобразования

$$\text{TRUE x y-->x}$$
$$\text{FALSE x y-->y}$$

Этот альтернативный подход дает основу для представления булевых констант и операций средствами чистого λ-исчисления.

Выражение основных примитивов Лиспа в -исчислении (продолжение)

Условный оператор if выражается следующим образом:

$$\text{if} = \lambda\ \text{p}.\lambda\ \text{q}.\lambda\ \text{r}.\text{p}\ \text{q}\ \text{r}$$

и, как теперь можно ожидать, TRUE и FALSE представляются в виде

$$\text{TRUE} = \lambda\text{x}.\ \lambda\text{y}.\text{x}$$

$$\text{FALSE} = \lambda x.\ \lambda y.y$$

Так, например:

if TRUE A B=(λp. λq. λr.p q r)(λx. λy.x) A B-> $_\beta$(λq.λr.(λx.λy.x) q r) A B-> $_\beta$
(λr.(λx.λy.x) A r) B-> $_\beta$(λx.λy.x) A B-> $_\beta$ (λy.A) B-> $_\beta$ A
как и ожидалось.

Выражение основных примитивов Лиспа в -исчислении (продолжение)

Имея такое представление TRUE и FALSE, можно теперь записать выражения для вычисления A OR B, A AND B, NOT A и т. д. Например, функции AND и OR выражаются следующим образом:

$$\text{AND} = \lambda x.\lambda y.x\ y\ \text{FALSE}$$
$$\text{OR} = \lambda x.\lambda y.(x\ \text{TRUE})\ y$$

Правильность такого представления может быть доказана получением соответствующих таблиц истинности. Например:

$$\text{TRUE AND FALSE} = (\lambda x.\ \lambda y.\ x\ y\ (\lambda x.\ \lambda y.\ y))\ (\ \lambda x.\ \lambda y.\ x)\ (\lambda x.\ \lambda y.\ y)$$
$$-->_\beta (\lambda y.(\lambda x.\lambda y.x) y (\lambda x.\lambda y.y))(\lambda x.\lambda y.y)$$
$$-->_\beta (\lambda x.\lambda y.x)(\lambda x.\lambda y.y)(\lambda x.\lambda y.y)$$
$$-->_\beta (\lambda y.(\lambda x.\lambda y.y))(\lambda x.\lambda y.y)$$
$$-->_\beta \{\lambda x.\lambda y.y)$$
$$= \text{FALSE}.$$

Выражение основных примитивов Лиспа в -исчислении (продолжение)

Теперь, имея представления для булевых констант TRUE и FALSE, можем использовать их для определения списков.
Обычно списки строятся с помощью двух функций-конструкторов, одна из которых обозначает пустой список (NIL), а другая строит новый список из элемента и старого списка (CONS). Чтобы представить CONS в чистом λ-исчислении, следует рассматривать выражение

$$\mathrm{CONS\ h\ t}$$

как функцию, берущую в качестве одного из своих аргументов функцию-селектор и применяющую ее к выражениям для головы (h) и хвоста (t) списка. Следовательно,

$$\mathrm{CONS} = \lambda h.\lambda t.\lambda s.s\ h\ t$$

где переменные h, t, s соответствуют голове списка, хвосту списка и функции-селектору.

Выражение основных примитивов Лиспа в -исчислении (продолжение)

Селекторы — это такие функции, которые возвращают либо первый, либо второй аргумент функции-конструктора. Поэтому они могут быть представлены в виде следующих λ-выражений:

$$\lambda\ \mathrm{h}.\lambda\ \mathrm{t}.\mathrm{h} \text{ и } \lambda\ \mathrm{h}.\lambda.\mathrm{t}\ \mathrm{t}$$

что в точности соответствует определениям TRUE и FALSE, приведенным выше. Это означает, что функции для выделения головного и хвостового элементов списка L (т. е. CAR и CDR соответственно) могут быть определены следующим образом:

$$\mathrm{CAR}{=}\ \lambda\mathrm{L}.\mathrm{L}\ \mathrm{TRUE}$$
$$\mathrm{CDR}{=}\lambda\mathrm{L}.\mathrm{L}\ \mathrm{FALSE}$$

Применение функций CAR или CDR к выражению CONS a b дает в результате a или b соответственно. В этом легко убедиться:

$$\mathrm{CAR}\ (\mathrm{CONS}\ \mathrm{a}\ \mathrm{b}){=}\ (\lambda\mathrm{c}.\mathrm{c}\ \mathrm{TRUE})((\lambda\mathrm{h}.\lambda\mathrm{t}.\lambda\mathrm{s}.\mathrm{s}\ \mathrm{h}\ \mathrm{t})\mathrm{a}\ \mathrm{b}){-}{-}{>}_{\beta}$$
$$((\lambda\mathrm{h}.\lambda\mathrm{t}.\lambda\mathrm{s}.\mathrm{s}\ \mathrm{h}\ \mathrm{t})\mathrm{a}\ \mathrm{b})\mathrm{TRUE}\ {-}{-}{>}_{\beta}(\lambda\mathrm{s}.\mathrm{s}\ \mathrm{a}\ \mathrm{b})\mathrm{TRUE}{-}{-}{>}_{\beta}\ \mathrm{TRUE}\ \mathrm{a}\ \mathrm{b}{=}$$
$$(\lambda\mathrm{h}.\lambda\mathrm{t}.\mathrm{h})\mathrm{a}\ \mathrm{b}{-}{-}{>}_{\beta}\ (\lambda t.a)b{-}{-}{>}a$$

Выражение основных примитивов Лиспа в -исчислении (продолжение)

Последней функцией обработки списков, которую мы определим в терминах чистого λ-исчисления, будет функция NULL, которая при применении к пустому списку (т. е. NIL) возвращает TRUE. Определение этой функции приводит нас к удобному представлению пустого списка. Начнем с рассмотрения выражения

NULL (CONS a b) = NULL ((λ h.λ t.λ s.s h t)a b)-->NULL (λ s.s a b)

Так как мы ожидаем, что это выражение дает FALSE для любых значений a и b, ясно, что это выражение для функции NULL должно иметь вид

$$\text{NULL} = \lambda\ \text{c.}\ \text{c}(\ \lambda\ \text{h.}\ \lambda\ \text{t.}\ \text{FALSE}\ )$$

Отсюда естественным образом мы приходим к выражению для пустого списка, определяемому таким образом, чтобы NULL x давало TRUE, если x — NIL, и FALSE в противном случае:

$$\text{NIL} = \lambda\ \text{x.TRUE}$$

Выражение основных примитивов Лиспа в -исчислении (продолжение)

В качестве примера приведем последовательность редукций для выражения NULL (CDR (CONS 1 NIL)):

$$\text{NULL (CDR (CONS 1 NIL))}=$$
$$(\lambda\ \text{c.c}(\lambda\ \text{h.}\lambda\text{t.FALSE}))(\text{CDR}(\text{CONS 1 NIL}))$$
$$\text{->}(\text{CDR}(\text{CONS 1 NIL}))(\lambda\ \text{h.}\lambda\text{t. FALSE})$$
$$=((\lambda\ \text{c.c FALSE})(\text{CONS 1 NIL}))(\lambda\ \text{h.}\lambda\ \text{t.FALSE})$$
$$\text{-->}((\text{CONS 1 NIL})\ \text{FALSE})\ (\lambda\ \text{h.}\lambda\ \text{t. FALSE})$$
$$=(((\lambda\ \text{h.}\lambda\ \text{t.}\lambda\ \text{s.s h t})\ 1\ \text{NIL})\ \text{FALSE})(\lambda\ \text{h.}\lambda\ \text{t.FALSE})$$
$$\text{-->}(((\lambda\ \text{t.}\lambda\ \text{s.s 1 t})\ \text{NIL})\ \text{FALSE})\ (\lambda\ \text{h.}\lambda\ \text{t. FALSE})$$
$$\text{-->}((\lambda\ \text{s.s 1 NIL})\ \text{FALSE})\ (\lambda\ \text{h.}\lambda\ \text{t. FALSE})$$
$$\text{-->}(\text{FALSE 1 NIL})\ (\lambda\ \text{h.}\lambda\ \text{t. FALSE})$$
$$=((\lambda\ \text{x.}\lambda\ \text{y.y})\ 1\ \text{NIL})\ (\lambda\ \text{h.}\lambda\ \text{t. FALSE})$$
$$\text{-->}\ ((\lambda\ \text{y .y})\ \text{NIL})(\lambda\ \text{h.}\lambda\ \text{t .FALSE})$$
$$\text{-->NIL}\ (\lambda\ \text{h.}\lambda\ \text{t. FALSE})$$
$$\text{-->}(\lambda\ \text{x.TRUE})(\lambda\ \text{h.}\lambda\text{t.FALSE})\ \text{->TRUE}$$

Выражение рекурсии в λ-исчислении

До сих пор мы рассматривали представление только нерекурсивных функций в λ-исчислении. В языках программирования высокого уровня, однако, необходимо иметь возможность записывать определение рекурсивных функций. Мы можем выразить рекурсию в языке высокого уровня, так как имеем возможность дать имя каждой функции, используемой в программе.

На такие имена можно ссылаться где угодно в программе (при определенных ограничениях, обусловленных языком), даже в теле функции, названной тем именем, на которое ссылаемся.

Теперь рассмотрим проблему рекурсивных функций в λ-исчислении, где функции не имеют имени. Поэтому для представления рекурсии необходимо придумать метод, позволяющий функциям вызывать себя не по имени, а каким-то другим образом. Другой (гораздо менее очевидный) взгляд на рекурсию состоит в том, чтобы представить рекурсивную функцию как функцию, имеющую саму себя в качестве аргумента. В этом случае функция может оказаться связанной с одной из своих собственных переменных и будет, таким образом, содержать в своем теле ссылки на саму себя.

Выражение рекурсии в λ-исчислении (продолжение)

Рассмотрим, например, рекурсивную функцию sum, определяемую на языке Лисп следующим образом:
(defun sum (n) (if (= n 0) 0 (+ n (sum (- n 1)))))
Это выражение может быть представлено в виде λ-абстракции, имеющей дополнительный параметр, который при применении этой абстракции связывается с самой функцией. Мы запишем эту промежуточную версию функции SUM:
SUM = λ s.λ n. if(= n 0) 0 (+n (s (- n 1)))
Таким образом, вы ввели некоторый функционал s в тело λ-абстракции.
Все, что нам осталось сделать теперь, — это связать s со значением функции sum, которую мы пытаемся определить. Для того, чтобы понять как это сделать рассмотрим как математики иногда поступают при решении уравнений.

Выражение рекурсии в λ-исчислении (продолжение)

Допустим, нужно решить уравнение: $x^2 + 3x - 1 = 0$. Выражаем

$$x(x+3) = 1$$

$$x = 1/(x+3)$$

Далее до заданной точности: $x = 1/(3 + 1/(3 + x))$

$$x = 1/(3 + 1/(3 + 1/(3 + x)))$$

$$x = 1/(3 + 1/(3 + 1/(3 + 1/(...))))$$

Т.е. мы пытаемся найти представление решения в форме $x = f(x)$. Далее начинается подстановка функции в функцию $x = f(f(f(...f(x)..)))$. В зависимости от вида функции f мы можем сходиться к некоторому x_0 — неподвижной точке отображения (теорема о сжимающем отображении). Доказано, что любой λтерм имеет неподвижную точку. Значит интерпретатор может быть реализован при помощи некоторой функции Y, которая для функционала f находит его неподвижную точку (соответственно определяя искомую функцию) — $f = Y f$.

Выражение рекурсии в λ-исчислении (продолжение)

Эту специальную функцию Y называют Y - комбинатором, которая удовлетворяет следующему уравнению:

$$Y f = f(Y f)$$

Посмотрим теперь, что получится при применении Y к функции SUM, приведенной выше:

Y SUM=Y(λ s. λ n. if(=n 0) 0 (+n (s(- n 1))))$\Rightarrow$

(λ s. λ n. if (=n 0) 0(+n(s(- n l)))) (Y SUM)

$\Rightarrow$ λ n. if (= n 0) 0 (+n ((Y SUM)(- n 1)))

Это оказалось тем, что нам нужно. Чтобы убедиться в этом, распишем внутреннее выражение (Y SUM) подобным образом:

$\Rightarrow$(λ n. if (= n 0) 0 (+n ((λ n. if (=n 0) 0 (+n ((Y SUM)(- n 1)))) (- n 1))))

Мы видим, что данное выражение ведет себя точно так же, как исходное рекурсивное определение sum. Чтобы завершить рассмотрение рекурсии, мы должны дать определение Y. Очевидный путь сделать это состоит в том, чтобы включить Y в множество констант, т. е. сделать эту функцию встроенной подобно + или if. Однако существуют и другие пути.

Выражение рекурсии в λ-исчислении (продолжение)

Y может быть записан в виде λ-выражения, что очень важно для чистого λ- исчисления, не имеющего встроенных функций. Рассмотрим следующее λ-выражение, которое имеет довольно необычный вид:
λ h.(λ x.h(x x))(λ x.h(x x))
Именно это выражение и представляет собой функцию Y. Чтобы убедиться в этом, применим данное выражение к функции f:
Yf = λ h.(λ x.h(x x))(λ x.h(x x)) f
$\Rightarrow$ (λ x.f (x x))(λ x.f(x x))$\Rightarrow$
f(λ x.f(x x))(λ x.f(x x)) = f(Y f)
Y редко реализуется таким путем из соображений эффективности. Существует много реализаций расширенного λ-исчисления, где имеется возможность давать имена λ-выражениям.
Литература: Филд, Харрисон Функциональное программирование (см. на github)