## 4.3 Деревья и генетическое программирование

**Генетическое программирование** (**ГП**) (***Genetic Programming, GP***) является скорее названием для исследовательского сообщества, нежели метода. Данное сообщество изучает применение стохастических методов для поиска и оптимизации небольших *компьютерных программ* или других вычислительных устройств. Отметим, что говоря об *оптимизации* компьютерных программ, мы должны также определить и *субоптимальные* программы, вместо рассмотрения программ, которые работают только правильно, либо неправильно [1]. Поэтому в ГП как правило рассматриваются пространства поиска, в которых определено *множество* возможных программ (обычно небольших), но при этом неясно какие из программ лучше остальных и насколько. Примером является поиск поведения для команды роботов-футболистов, либо аппроксимация множества данных произвольными математическими выражениями, либо отыскание конечного автомата, соответствующего данному языку.

Поскольку компьютерные программы могут иметь различный размер, то в рассматриваемом сообществе используются динамические структуры данных для их представления, в основном **списки** и **деревья**. В ГП такие списки и деревья как правило сформированы с использованием базисных функций или операций ЦП (например, + или *if* или *пнуть-в-сторону-цели*). Часть из этих операций не может быть выполнена совместно друг с другом. К примеру, 4+*пнуть-в-сторону-цели()* не имеет смысла, если только *пнуть-в-сторону-цели* не возвращает число. По схожим причинам некоторые узлы не могут иметь ограничение на *количество* узлов-потомков: например, если узел реализует операцию *перемножение матриц*, то вероятно он ожидает на входе две матрицы для перемножения. По этой причине операторы инициализации и TWEAKING делают с поддержкой целостности для обеспечения *корректности* их результатов.

Один из вопросов, остроумно решаемых при оптимизации компьютерных программ, является способ вычисления приспособленности: нужно просто запустить программу и посмотреть, как она работает! Это означает, что *данные*, используемые для хранения информации о генотипе особей, могут быть преобразованы к виду, соответствующему запускаемому фенотипу-*коду*. Неудивительно, что ранние реализации ГП использовали язык Lisp, в котором программный код и описывающие его данные выглядят похоже.

Наиболее часто в ГП в качестве представления используются **деревья**, впервые предложенные для этого Найклом Крамером *(Nichael Cramer)*[2], однако большая часть из того, что описывается далее, была придумана Джоном Козой, внесшего значительный вклад[3].

Рассмотрим дерево справа, содержащее математическое выражение $\sin(\cos(x-\sin x)+x\sqrt{x})$, представляющее *дерево грамматического разбора* (*parse tree*) для простой программы, которая вычисляет это выражение. В таком дереве узлом может быть функция, условная конструкция и т.д., а потомками — аргументы этой функции. Если использовать только функции, без операторов, то С-подобная запись для данного выражения выглядела бы как sin(add(cos(subtract(x, sin(x))), multiply(x, sqrt(x)))).

Подобная запись используется в языках семейства Лисп (*Lisp*). Имена функций записываются *внутри* скобок и запятые удаляются, поэтому функция foo(bar, baz(quux)) представляется в виде (foo bar (baz quux)). В языке Лисп объекты вида ( ...) являются односвязаными списками, поэтому Лисп обрабатывает код программы так, как будто бы это обычные данные, что идеально подходит для ГП. Дерево справа на Лиспе можно записать как (sin (+ (cos (x (sin x))) (+ x (sqrt x)))).

---

[0] Перевод раздела из книги Luke S. Essentials of Metaheuristics. A Set of Undergraduate Lecture Notes. Zeroth Edition. Online Version 0.9. July, 2010 (`http://cs.gmu.edu/~sean/book/metaheuristics/`). Перевел – Юрий Цой, 2010 г. Любые замечания, касающиеся перевода, просьба присылать по адресу yurytsoy@gmail.com

Данный текст доступен по адресу: `http://qai.narod.ru/GA/meta-heuristics_4_3.pdf`

[1] Джон Коза предложил именно такое понимание в своей книге *Генетическое программирование*: «...можно было предположить, что я говорил о написании *корректных* компьютерных программ...В действительности же в данной книге в основном рассматриваются *некорректные* программы. Я, в частности, хочу определить такое понятие, при котором допускаются градации качества исполнения компьютерных программ. Какие-то программы будут очень плохими, другие лучше, чем некоторые, третьи приблизительно корректными, а одна какая-нибудь программа может оказаться на 100 % корректной.» (с. 130 из John R. Koza, 1992, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press.)

[2] В одной и той же статье Крамер предложил как ГП с кодированием деревьями, так и ГП с кодированием списками, схожее с описываемым в Разделе 4.4. Он назвал это версией языка JB, основанной на списках, и версией языка TB, основанной на деревьях. Nichael Lynn Cramer, 1985, A representation for the adaptive generation of simple sequential programs, in John J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187.

[3] Весь материал в Разделе 4.3, если не оговорено иное, разработан Козой. Для ссылки на работу см. сноску 1, на этой странице

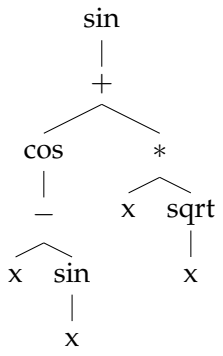


Рис. .18: Дерево для задачи символьной регрессии

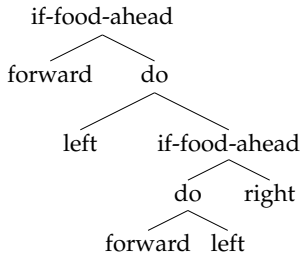


Рис. .19: Дерево для задачи искусственного муравья

Каким образом правильно оценить приспособленность особи? Возможно выражение должно как можно лучше аппроксимировать некоторые данные, представляющие, допустим, двадцать пар вида $\langle x_i, f(x_i)\rangle$. Для тестирования дерева $i$-й парой возвращаемое значение оператора x принимается равным $x_i$, затем исполняется вычисление по дереву с записью результата в переменную $v_i$, после чего подсчитывается квадратичная ошибка относительно $f(x_i)$, т.е. как $\delta_i = (v_i - f(x_i))^2$. Приспособленность особи можно вычислить как квадратный корень из суммарной ошибки: $\sqrt{\delta_1 + \delta_2 + \ldots + \delta_n}$. Задачи в ГП, имеющие подобную постановку, в которых необходимо аппроксимировать произвольную кривую сложной формы по наборам измерений, называются задачами **символьной регрессии** (***symbolic regression***).

Программы не обязательно должны представлять уравнения, они могут и *делать* что-нибудь, а не просто *возвращать некоторое значение*. Примером является дерево, показанное на рис. 19, которая представляет программу, управляющую движением муравья на поле, по которому разбросана пища. Оператор if-food-ahead зависит от двух аргументов, один из них соответствует оценке есть ли впереди еда, а другой — ее отсутствию. Оператор do также имеет два аргумента и оценивает сначала левый, а затем правый. Операторы left и right поворачивают муравья на 90°влево или вправо, forward перемещает муравья на одну клетку вперед, при этом муравей съедает еду перед ним. Считая что пища разбросана внутри некоторой сетки, целью является поиск программы, которая, будучи исполненной (возможно, многократном), приводит к поеданию как можно большего количества пищи, которое и является значением приспособленности. Это достаточно популярная тестовая задача известная как **задача искусственного муравья** (artificial ant).

Программа управления искусственным муравьем на языке Лисп (в несколько некорректном виде) будет выглядеть так: (if-food-ahead forward (do left (if-food-ahead (do forward left) right))). На C аналогичная программа может быть записана как if (foodAhead) forward(); else left(); if (foodAhead) forward(); left(); else right(); . Здесь предполагается, что do это обычный блок {}[4].

Для ГП с кодированием деревьями, конечно же, можно использовать любой алгоритм оптимизации. Однако существует и свой традиционный алгоритм, описанный ранее в Разделе 3.3.3 и являющийся таковым без особенных на то причин.

[4] Эх, код без выравнивания плохо читается

### 4.3.1 Инициализация

В ГП новые деревья создаются путем итеративного выбора элементов из **множества функций** (***function set***) (набор элементов, которые могут выступать в роли узлов дерева) и их связывания. В примере для задачи искусственного муравья множество функций могло состоять из if-food-ahead, do, forward, left, и right. Для примера задачи символьной регрессии это множество могло включать +, -, *, sin, cos, sqrt, x, и других различных математических операторов. Отметим, что для каждой функции во множестве определена ее **арность**, обозначающая заранее определенное количество узлов-потомков. У sin один потомок. У do и + — два, а x и forward вообще не имеют потомков. Узлы с нулевой арностью (без потомков) называются *листьями*, узлы с арностью $\geq 1$ — *не листьями*. Алгоритмы, которые связывают узлы между собой в общем случае должны учитывать эти обстоятельства, чтобы построить правильное дерево.

Одним из распространенных алгоритмов является **Grow**, который строит случайные деревья в глубину до заданного значения.

**Алгоритм 54** Алгоритм **Grow**

```
max ← максимально возможная глубина
FunctionSet ← множество функций

return DoGrow(1, max, FunctionSet)

Procedure DoGrow (depth, max, FunctionSet)
if depth > max then
   return Copy(случайный узел-лист из FunctionSet)
else
   b ← Copy(случайный узел из FunctionSet)
   l ← количество потомков для n
   for i от 1 до l do
      i-й потомок узла n ← DoGrow(depth + 1, max, FunctionSet)
   end for
   return n
end if
```

Алгоритм **Full** является небольшой модификацией предыдущего алгоритма, в которой принудительно строится дерево максимально возможной глубины.

**Алгоритм 55** Алгоритм **Full**

```
max ← максимально возможная глубина
FunctionSet ← множество функций

return DoFull(1, max, FunctionSet)

Procedure DoFull (depth, max, FunctionSet)
if depth > max then
   return Copy(случайный узел-лист из FunctionSet)
else
   b ← Copy(случайный узел не лист из FunctionSet) {Единственное отличие!}
   l ← количество потомков для n
   for i от 1 до l do
      i-й потомок узла n ← DoFull(depth + 1, max, FunctionSet)
   end for
   return n
end if
```

Традиционно в ГП при построении нового дерева случайно выбирается один из этих алгоритмов, а максимальная глубина выбирается от 2 до 6. Данная процедура получила название **Ramped Half-and-Half**.

Сложность использования этих алгоритмов в том, что они не предоставляют способа контролировать размер деревьев, и результатом их работы часто является довольно странное «распределение»

**Алгоритм 56** Алгоритм Ramped Half-and-Half

1: $minMax \leftarrow$ минимальная разрешенная максимальная глубина
2: $maxMax \leftarrow$ максимальная разрешенная максимальная глубина {...если это вообще имеет какой-нибудь смысл...}
3: $FunctionSet \leftarrow$ множество функций

4: $d \leftarrow$ случайная величина, равномерно распределенная в интервале $[minMax, maxMax]$
5: **if** $0.5 <$ случайная величина, равномерно распределенная в интервале $[0, 1]$ **then**
6:  **return** DoGrow(1, $d$, $FunctionSet$)
7: **else**
8:  **return** DoFull(1, $d$, $FunctionSet$)
9: **end if**

деревьев. В настоящее время разработано весьма большое количество алгоритмов, позволяющих получать более предсказуемые результаты[5]. Вот мой алгоритм, **PTC2**[6], производящий дерево с глубиной не более требуемой и ограничением на максимальное количество потомков у каждого нелистового узла. Этот алгоритм легко описать. Мы случайно расширяем дерево до тех пор, пока количество нелистовых узлов в сумме с незаполненными (неинициализированными) узлов не станет равно или больше желаемого размера. После этого на место незаполненных узлов подставляются узловые:

**Алгоритм 57** Алгоритм PTC2

1: $s \leftarrow$ требуемый размер дерева
2: $FunctionSet \leftarrow$ множество функций

3: **if** $s = 1$ **then**
4:  **return** Copy(Случайный листовой узел из $FunctionSet$)
5: **else**
6:  $Q \leftarrow \{\}$
7:  $r \leftarrow$ Copy(Случайный нелистовой узел из $FunctionSet$)
8:  $c \leftarrow 1$
9:  **for** каждый неинициализированный узел-потомок $b$ из $r$ **do**
10:  $Q \leftarrow Q \cup \{b\}$
11:  **end for**
12:  **while** $c + \|Q\| < s$ **do**
13:  $a \leftarrow$ случайный неинициализированный узел, изъятый без возрата из $Q$.
14:  $m \leftarrow$ Copy(Случайный нелистовой узел из $FunctionSet$)
15:  $c \leftarrow c + 1$
16:  Вместо $a$ подставляется $m$
17:  **for** каждый неинициализированный узел-потомок $b$ из $m$ **do**
18:  $Q \leftarrow Q \cup \{b\}$
19:  **end for**
20:  **end while**
21:  **for** каждый узел-потомок $q \in Q$ **do**
22:  $m \leftarrow$ Copy(Случайный листовой узел из $FunctionSet$)
23:  Вместо $q$ подставляется $m$
24:  **end for**
25:  **return** $r$
26: **end if**

[5] Ливиу Панаит (*Liviu Panait*) и я делали исследование на эту тему, представленное в Sean Luke and Liviu Panait, 2001, A survey and comparison of tree generation algorithms, in Lee Spector, et al., editors, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), pages 81–88, Morgan Kaufmann, San Francisco, California, USA.

[6] PTC2 предложен в Sean Luke, 2000, Two fast tree-creation algorithms for genetic programming, IEEE Transactions on Evolutionary Computation, 4(3), 274–283. Это достаточно очевидный алгоритм, который без сомнения был использован и ранее в различных других контекстах.

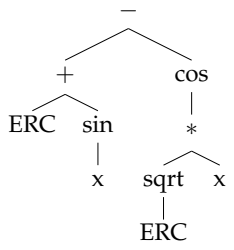


Рис. .20: Дерево с ERC узлами. См. также рис. 21

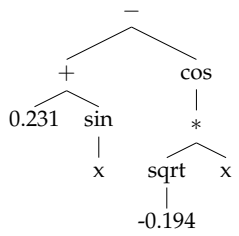


Рис. .21: Дерево на рис. 20 с ERC узлами, замененными на константы

**Эфемерные случайные константы** Часто бывает полезным включить во множество функций потенциально бесконечное число констант (таких как 0.2462, $\langle 0.9, 2.34, 3.14\rangle$, 2924056792 или "s%&e:m"), которые вставляются в дерево при построении. Например, в задаче символьной регрессии было бы полезным уметь включить в уравнения константы, допустим $-2.3129$. Как это можно сделать? Для начала, при осторожном проектировании, множество функций не обязательно должно быть фиксированного размера, можно включить туда специальный узел (часто листовой), который называют **эфемерная случайная константа** (***ephemeral random constant, ERC***). Каждый раз, когда ERC выбирается для вставки в дерево, она автоматически преобразуется в псевдослучайную константу, либо в заданное вами значение. В дальнейшем это значение уже не меняется (кроме как специальным оператором мутации). На рис. 20 показаны ERC, вставленные в дерево, а на рис. 21 — результат их преобразования в константы.

### 4.3.2 Рекомбинация

В ГП рекомбинация обычно производится с использованием **кроссинговера для поддеревьев** (***subtree crossover***). Его идея очевидна: у каждой скрещиваемой особи выбирается случайное поддерево (которое может начинаться и от корня), а затем производится обмен этими поддеревьями. Часто, хотя и совершенно не обязательно, для случайного выбора поддерева в 10 % случаев выбирают листовые узлы, а в 90 % — листовые. В алгоритме 57 показано, как выбрать поддерево заданного типа:

### 4.3.3 Мутация

В ГП мутация производится нечасто, поскольку оператор кроссинговера **негомологичен**[7] и вносит большие изменения. Тем не менее, присутствует множество возможностей для мутации, например:

- **Мутации поддерева** (***Subtree mutation***): Выбираем случайное поддерево и заменяем его на случайно-сгенерированное дерево с использованием вышеприведенных алгоритмов. Обычно Grow используется с максимальной глубиной равной 5, и, как уже упоминалось, листовые узлы выбираются в 10% случаев, а нелистовые — в 90%.
- Замена случайного нелистового узла одним из его поддеревьев.
- Для случайного нелистового узла производится обмен поддеревьев местами.

[7] Вспомним, что в случае гомологичного кроссинговера скрещивание особи с собой дает в результате копии этой особи

**Алгоритм 58** Выбор поддерева

```
r ← корневой узел дерева
f(node) ← функция, возвращающая true, если узел принадлежит к искомому типу

глобальная c ← 0
CountNodes(r, f)
if c = 0 then
   return □ {"null", "failure"или что-нибудь в этом духе}
else
   a ← случайная целочисленная величина из интервала [1; c]
   c ← 0
   return PickNode(r, a, f)
end if

Procedure CountNodes(r, f) {Обычный поиск в глубину}
if f(r) == true then
   c ← c + 1
end if
for Каждый узел-потомок i для r do
   CountNodes(i, f)
end for

Procedure PickNode(r, a, f) {И снова поиск в глубину!}
if f(r) == true then
   c ← c + 1
   if c ≥ a then
      return r
   end if
end if
for Каждый узел-потомок i для r do
   v ← PickNode(r, a, f)
   if v ≠ □ then
      return v
   end if
end for
return □ {До этого места программа не должна дойти}
```

- Если узлы дерева являются эфемерными случайными константами, то они изменяются с некоторым шумом.
- Выбираются два поддерева особи таким образом, что ни одно из них не содержит другое, и меняются местами.

Можно использовать алгоритм 57 для выбора поддеревьев для данных способов. Алгоритм 57 называется **выбор поддерева**, но с таким же успехом он мог бы быть назван и выбор узла. Для начала подсчитываются все узлы дерева заданного типа, например, нужно выбрать листовой узел. Затем генерируется случайное число $a$, меньшее количества посчитанных узлов. Посл этого проходим по дереву в глубину, одновременно подсчитывая количество встреченных узлов данного типа, до тех пор, пока не дойдем до $a$-го, который и будет искомым.

### 4.3.4 Леса и автоматически определенные функции

В ГП нет ограничения на использование только одного дерева: совершенно разумным бывает использование генотипа в форме вектора деревьев (известного как **лес**). Например, однажды я разрабатывал простые программы для команды роботов-футболистов, и одна особь представляла целую команду роботов. Каждая программа для робота включала два дерева: одно определяло игру робота без мяча (возвращало вектор направления движения), а второе — игру робота, когда тот был достаточно близко к мячу, чтобы нанести удар (возвращало вектор направления удара). Особь состояла из $n$ таких пар деревьев, допустим по одной на каждого робота, или по одной на каждый класс роботов (голкиперы, форварды и т.д.), или всего одна пара на всех роботов сразу (команда близнецов). Таким образом, особь для футбола могла иметь от 2 до 22 деревьев!

Деревья также могут быть использованы, чтобы задавать **автоматически определяемые функции** (**АОФ**) (***automatically defined functions***, ***ADFs***)[8], которые могут вызываться главным деревом. Здесь используется такой прием как **модульность**, которая позволяет вести поиск в очень больших пространствах в том случае, если известно, что решения содержат повторяющиеся элементы. Вместо того, чтобы требовать кодировать в генотипе особи все такие повторы (не нарушая состава и порядка), что очень маловероятно, можно поступить проще, а именно: разбить особь на модули, и в генотипе задавать связи и отношения между этими модулями.

В случае использования АОФ, когда замечено, что в хороших решениях часто присутствуют большие по размеру деревья с повторяющимися поддеревьями, предпочтительней, чтобы особь включала одну или две *подфункции*, которые многократно вызываются главным деревом. Этого можно достичь, добавив к особи второе дерево (АОФ) и включив во множество функций родительского дерева особые узлы[9], которые просто вызывают это второе дерево. При необходимости можно добавлять новые АОФ.

В качестве иллюстративного примера, рассмотрим задачу для ГП, в которой хорошее решение с большой вероятностью будет иметь подфункции от двух аргументов. Заранее неизвестно, какие это функции, однако предполагается, что это условие верно. Этого можно достичь, если представление особи в ГП будет состоять из двух деревьев: главного дерева и еще одного АОФ-дерева от двух аргументов, которое назовем, допустим, ADF1.

Добавим ко множеству функций главного дерева новую нелистовую: ADF1(*child1*, *child2*). Дерево ADF1 может иметь любое множество функций, которое допустимо для построения этого дерева. Кроме этого понадобятся еще две листовые функции, также добавляемые во множество функций главного дерева. Назовем их ARG1 и ARG2.

На рис. 22 показан пример особи. Вот как все работает. Сначала исполняем первое дерево. При вызове узла ADF1, сначала производим вызов его узлов-потомков и сохраняем их результат (скажем, *result1* и *result2*). После этого вызываем дерево ADF1. При вызове его функции ARG1, автоматически возвращается *result1*, и аналогично ARG2 возвращает *result2*. Когда дерево ADF1 завершает исполнение, его возвращаемый результат сохраняется (допустим, что в переменной *final*). Затем возвращаемся в главное дерево — узел ADF1 возращает результат *final*, а исполнение главного дерева продолжается далее.

Заметим, что можно задавать больше одного АОФ-дерева. При этом из одного АОФ-дерева можно вызывать другие! Ведь нет никаких причин запрещать вызовы функций из других функций, верно? Теоретически можно организовать и *рекурсивный* вызов функций, когда АОФ-деревья вызывают

[8] Автоматически определяемые функции также разработаны Джоном Козой, хотя представлены в его второй книге, John R. Koza, 1994, *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press.

[9] У каждого дерева может быть свое, возможно, уникальное множество функций.

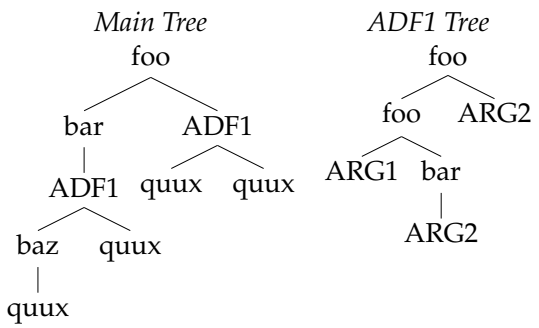


Рис. .22: Пример АОФ-дерева

друг друга. Однако особи не настолько умны, чтобы с этим справиться, поэтому оберегайте вашу систему от вечных рекурсивных циклов, ограничивайте максимальную глубину вызовов.

Напоследок еще один вариант: **автоматически определяемые макросы** (**АОМ**) (***automatically defined macros***, ***ADMs***), придуманные Ли Спектром (*Lee Spector*)[10]. Здесь при вызове узла ADF1, программа немедленно переходит в соответствующее дерево, не дожидаясь сначала вызова потомков этого узла. Вместо этого каждый раз при вызове ARG1, происходит возврат в исходное дерево, вызов первого узла-потомка, получение его результата, перехода в дерево ADF1 и затем вместо ARG1 подставляется вычисленный результат. Это происходит каждый раз при вычислении ARG1. Аналогично для ARG2. Идея заключается в ограниченной возможности селективно, либо повторно вызывать потомков наподобие конструкции if-then, циклов while и т.д. (В Лиспе это называется *макросы*, отсюда и название).

### 4.3.5 Сильно типизированное генетическое программирование

Сильно типизированное генетическое программирование является вариантом генетического программирования, предложенным Дэвидом Монтаной (*David Montana*)[11]. Вспомним, что в примерах, приведенных ранее каждый узел возвращает значения одного и того же типа (к примеру, в задаче символьной регрессии все узлы возвращают значения с плавающей запятой). Однако в больших программах это скорее исключение, чем правило. Что если в задаче символьной регрессии необходимо добавить особый оператор if, который принимает три аргумента: булевскую *проверку* (*test*), *то-значение* (*then-value*), возвращаемое в случае, если проверка истинна, и *иначе-значение* (*else-value*), если проверка ложна. Узел if возвращает вещественное значение, как и все другие узлы, однако требует наличие узла с булевским результатом. Это означает, что необходимо добавить некоторые узлы, которые возвращают только булевские значения, включая как листовые, так и нелистовые, например, операторы And и Not.

Проблема заключается в том, что для сохранения целостности мы больше не можем как раньше создавать деревья, скрещивать их или проводить мутации, не обращая внимания на то, какие узлы могут быть потомками данного узла и где они должны располагаться. Что случится, если попытаться умножить, например, $sin(x)$ на "ложь"? Чтобы избежать подобного необходимо каждому узлу присвоить **ограничения типа** (***type constraints***), чтобы регулировать соединения узлов и их порядок.

Существует множество подходов к реализации типизации. В простейшем случае, **атомарная типизация** (***atomic typing***) каждый тип представлен одним символом или целым числом. Типы возвращаемых значений каждого узла, ожидаемые типы узлов-потомков для узлов и ожидаемый тип значения, возвращаемый деревом в целом, задаются с помощью разрешенных типов. Узел может быть присоединен в качестве потомка, или являться корнем узла только если совпадают типы. В **типизации набором** (***set typing***) типы представлены не обычными символами, а *наборами* символов. Два типа считаются совпавшими, если пересечение их наборов не пусто. Типизация набором может использоваться для обеспечения достаточного объема информации о различных свойствах типа, включая подклассы.

Но и это еще не все. Атомарная типизация и типизация набором предусматривает конечный набор типов. Но как быть, если узлы работают матрицами? Например, рассмотрим узел *умножение-*

[10] Lee Spector, 1996, Simultaneous evolution of programs and their control structures, in Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 7, pages 137–154, MIT Press.

[11] David Montana, 1995, Strongly typed genetic programming, *Evolutionary Computation*, 3(2), 199–230.

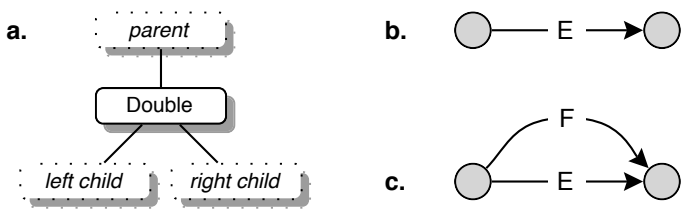


Рис. .23: Оператор double для реберного кодирования

*матриц*, который имеет двух потомков (возвращающих матрицы) и перемножает их результат, возвращая новую матрицу. Размерность этой матрицы является функцией от размерностей матриц узлов-потомков. Допустим, мы поменяли одного из потомков на поддерево, которое возвращает новую матрицу, с новым размером. С этим можно справиться, если изменить возвращаемый тип родительского узла, что в свою очередь может привести к каскадному изменению типов значений других узлов и их потомков в ходе перенастройки дерева. Такая типизация называется **полиморфной** (***polymorphic typing***). Она опирается на алгоритмы согласования типов, подобные используемым в языках программирования с полиморфной типизацией, таких как Haskell или ML. Она сложная.

### 4.3.6 Клеточное кодирование

Деревья также можно использовать в качестве коротких программ, что указать интерпретатору, как построить *вторичную* структуру данных (обычно граф). Эта вторичная структура затем и используется как фенотип. Такой подход известен под названием **Клеточное кодирование** (***Cellular Encoding***) (разработано Фредериком Груо)[12]. Общая идея заключается в выборе *начального графа* (*seed*) (например, графа, состоящего из одного узла или ребра) и передаче его корню дерева. Оператор корня изменяет и расширяет граф, а затем передает некоторые его элементы своим узлам-потомкам, которые также могут расширить граф, и передать модифицированную версию уже своим потомкам и т.д., до тех пор, пока дерево в дереве не закончатся узлы. После этого расширенный граф используется в качестве фенотипа.

Этот подход в изначальной формулировке Груо, использованный в основном для нейронных сетей, работал над *вершинами* графа, что требовало достаточно сложных механизмов работы. Альтернативой являются операции над *ребрами*, которые хотя и не могут привести к появлению произвольных графов, но весьма полезны для разреженных или планарных графов, которые часто можно найти в электрических цепях или конечных автоматах. Такой подход был назван Ли Спектором и мной **Реберным кодированием** (***Edge Encoding***)[13]. Поскольку оно проще в описании, я его и продемонстрирую.

Исполнение дерева в клеточном и реберном кодировании отличается от деревьев, используемых для той же задачи символьной регрессии: в них потомки получают объекты от родительских узлов, обрабатывают эти данные и передают результаты потомкам. В качестве примера на рис. 23 показан оператор реберного кодирования, названный double (удвоение). Он берет ребро, получаемое от родительского узла (ребро $E$ на рис. 23b) и создает его копию, соединяющую те же вершины, что и ребро-оригинал (ребро $F$ на рис. 23c). Затем оператор передает по одному ребру своим двум потомкам.

На рис. 24 показано дерево для реберного кодирования, которое конструирует конечный автомат. Помимо double, используются операторы: reverse, который разворачивает ребро, loop, который создает петлю для вершины, на которой заканчивается ребро, передаваемое в loop, bud, который

[12] Frr̈ edr̈ eric Gruau, 1992, Genetic synthesis of boolean neural networks with a cell rewriting developmental process, in J. D. Schaffer and D.Whitley, editors, *Proceedings of the Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN92)*, pages 55–74, IEEE Computer Society Press.

[13] Ли Спектор и написали раннюю статью на эту тему, назвав подход реберным кодированием: Sean Luke and Lee Spector, 1996, Evolving graphs and networks with edge encoding: Preliminary report, in John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1996 Conference*, pages 117–124, Stanford Bookstore. Однако я сомневаюсь, что мы первыми его изобрели: к тому времени, когда наша статья появилась, Джон Коза, Форрест Беннетт (*Forrest Bennett*), Дэвид Андре (*David Andre*) и Мартин Киэн (*Martin Keane*) уже использовали подобное представление для эволюции электрических цепей. См. See John R. Koza, Forrest H Bennett III, David Andre, and Martin A. Keane, 1996, Automated WYWIWYG design of both the topology and component values of electrical circuits using genetic programming, in John R. Koza, et al., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 123–131, MIT Press.

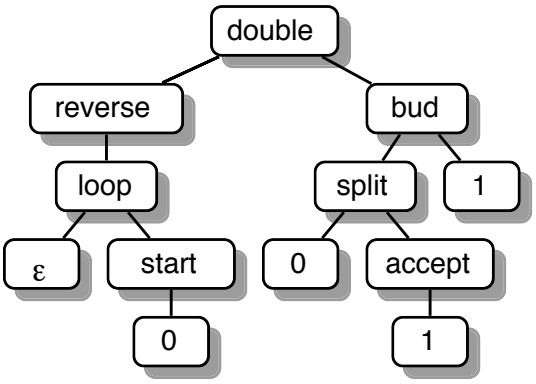


Рис. .24: Реберное кодирование

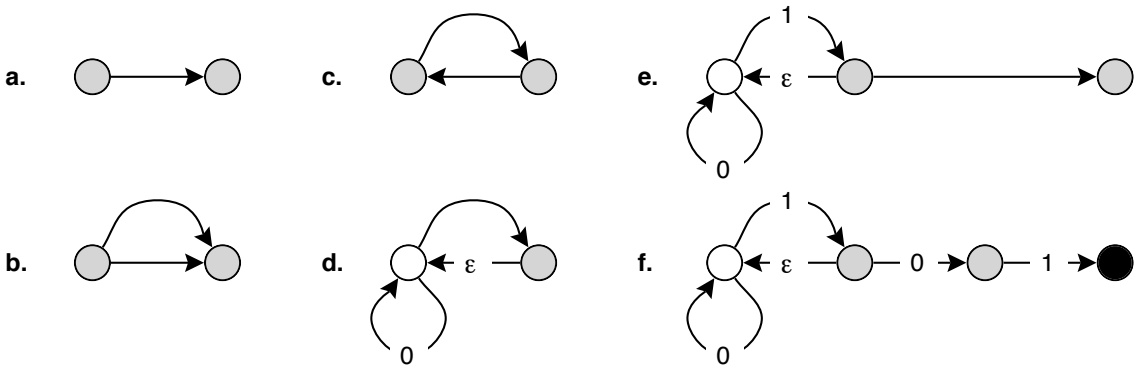


Рис. .25: Расширение конечного автомата с использованием дерева реберного кодирования на рис. 24. (a) Начальное ребро. (b) После оператора double. (c) После оператора reverse. (d) После операторов loop, $\epsilon$, start и 0. Белый кружок обозначает начальное состояние. (e) После операторов bud и 1. (b) После операторов split, 0, accept и 1. Черный кружок обозначает финальное состояние.

создает новую вершину, и ребро к ней, идущее от начала ребра, передаваемого в bud, split, который разделяет свое ребро на два, первое из которых идет начала исходного ребра в сторону нового узла, а второе от нового узла в конец исходного. Другие операторы, используемые в задаче конструирования конечного автомата, либо помечают свое ребро $(\epsilon, 1, 0)$, либо помечают конечную вершину ребра (start, accept).

Что, запутались? Я бы запутался! Но, возможно, так будет понятнее: на рис. 25 показаны результаты расширения графа, индуцируемые деревом на рис. 24. Работа начинается с графа с одним ребром, которые постепенно растет до конечного автомата для интерпретации регулярного языка (1|0)*01.

Клеточное и реберное кодирование являются примерами **косвенного** (***indirect***) или **развивающегося** (***developmental***) кодирования: способа представления, содержащего набор правил, определяющих развитие (эволюцию) вторичной структуры данных, используемой в качестве фенотипа. Косвенные способы кодирования популярны по двум причинам. Во-первых, в силу биологической привлекательности: ДНК является косвенной кодировкой, т.к. используется для формирования РНК и белков, которые затем принимают участие в формировании живых организмов. Во-вторых, существуют понятия **компактности** (***compactness***) и **модульности** (***modularity***), которые уже обсуждались ранее: многие правила в косвенном кодировании совершают многократные вызовы некоторых подправил. В клеточном и реберном кодировании модульности нет, но она тривиально вводится посредством включения автоматически определяемых функций. Аналогично, если только не используются АОФ, компактность также невелика: деревья реберного кодирования будут иметь как минимум столько же вершин, сколько ребер у искомого графа!

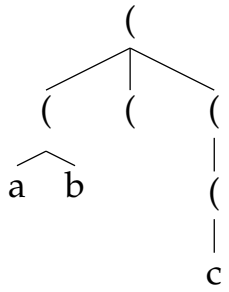


Рис. .26: Запись выражения ((a b) ( ) ((c))) с использованием круглых скобок

### 4.3.7 Стековые языки

Альтернативой Лиспу являются **стековые языки** (***stack languages***), в которых программный код представлен потоком инструкций обычно с постфиксной нотацией. Среди существующих языков программирования стековыми являются Forth и PostScript. Эти языки предполагают наличие стека для записи, хранения и извлечения временных переменных и, в ряде случаев, цепочек кода. Выражение $5 \times (4 + 3)$ на стековом языке будет записано как $534 + \times$. В стек записываются 5, 3 и 4; затем два числа извлекаются и складываются, а результат помещается в стек; после этого из стека извлекаются оставшиеся два числа и перемножаются, и то, что получается (35) записывается в стек.

В стековых языках часто используются подпрограммы путем записи в стек участков кода и их последующего многократного исполнения. Например, необходимо выделить вышеприведенную процедуру — $a \times (b+c)$ — в подпрограмму. Для этого операторы обрамляются скобками и подставляются в специальный оператор записи в стек: push $(+\times)$. Используя еще один оператор do, в котором $n$ раз производится считывание подпрограммы из стека, ее выполнение и запись обратно в стек, можно исполнять следующий код 5 7 9 2 4 3 6 5 9 push $+\times$ 4 do, вычисляющий $5\times(7+9)\times(2+4)\times(3+6)\times(5+9)$.

Стековые языки используются в ГП уже продолжительное время. Среди наиболее известных упомянем стековый язык для ГП **Push**[14], разработанный Ли Спектором. Push поддерживает множественные стеки, по одному для каждого типа данных, что позволяет коду аккуратно обрабатывать различные типы данных. Кроме этого в Push имеются особые стеки для хранения, изменения и исполнения кода, что позволяет программам на Push изменять свой код *во время исполнения*. Это позволяет реализовать, к примеру, автоматическое создание **самоадаптивных** (***self-adaptive***) операторов размножения.

Для использования стекового языка для оптимизации необходимо принять решение об используемом способе кодирования. Если язык просто формирует потоки символов и не использует ограничения, можно применять списочные способы представления (см. следующий раздел 4.4). Однако большинство стековых языков требуют, хотя бы чтобы скобки, используемые для разделения кода, присутствовали парами. Существует множество способов выполнить это требование. В некоторых языках за открывающей скобкой должен обязательно следовать символ не-скобка. Этого легко достичь, в точности также, как и во встреченных выражениях на Лиспе (см. рис. 18 и 19). Если же, наоборот, язык позволяет использовать скобку сразу же после открывающей скобки, например, ((a b) ( ) ((c))), можно просто считать открывающую скобку корневой вершиной для поддерева, а содержимое скобок — потомками этой вершины, как показано на рис. 26. Оба этих подхода требуют, чтобы вершины дерева обладали произвольной арностью. Либо, как в случае языка Push, можно использовать внутренний формат языка Лисп: вложенные связанные списки. Каждое скобочное выражение (такое как (a b)) формирует один связанный список, а элементы этого выражения могут являться другими связанными списками. Вершины, которым соответствуют элементы-связанные списки, называются **cons ячейками** (***cons cells***), представленные на рис. 27 символом $\triangleright$. Левый потомок cons ячейки содержит элемент списка, а правый потомок указывает на следующую cons ячейку, либо содержит символ $\square$, обозначающий конец списка.

[14] Основы изложены в статье Lee Spector and Alan Robinson, 2002, Genetic programming and autoconstructive evolution with the Push programming language, *Genetic Programming and Evolvable Machines*, 3(1), 7–40. Последняя версия языка описана в статье Lee Spector, Jon Klein, and Martin Keijzer, 2005, The Push3 execution stack and the evolution of control, in *Proceedings of the Genetic and Evolutionary Conference (GECCO 2005)*, pages 1689–1696, Springer.

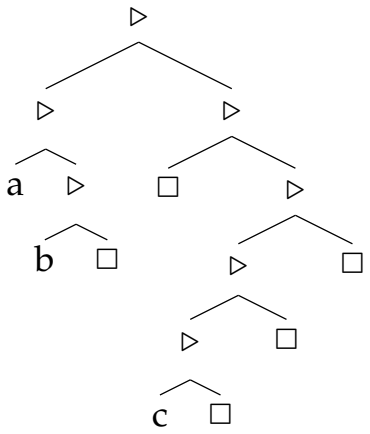


Рис. .27: Запись выражения ((a b) ( ) ((c))) с использованием cons ячеек