ПРОГРАММИРОВАНИЕ*, БЛОГ КОМПАНИИ NIX SOLUTIONS

Модели памяти, лежащие в основе языков программирования

ПЕРЕВОД

NIX_Solutions 15 марта 2017 в 11:29 23,6k

Оригинал: Kragen Javier Sitake…

Предлагаем вашему вниманию перевод статьи, посвящённой рассмотрению используемых в программировании моделей памяти.

Сегодня в программировании доминируют шесть основных моделей памяти (не путать с моделями памяти Intel 8086). Три из них проистекают из трех исторически наиболее важных языков программирования 1950-х годов — COBOL, LISP и FORTRAN, а остальные связаны с тремя исторически важными системами хранения данных: магнитная лента, иерархическая файловая система в Unix-стиле и реляционная база данных.

Эти модели на гораздо более глубоком уровне, чем синтаксис или даже система типов, определяют, что наши языки программирования могут или не могут делать. Давайте подробно рассмотрим эти модели, а затем обсудим некоторые возможные альтернативы и причины, почему они могут быть интересны.

Введение

Каждая современная среда программирования, в той или иной степени, использует все шесть моделей памяти, и это одна из причин того, почему наши системы столь сложны и трудны для понимания.

Здесь мы проанализируем, как каждая из этих моделей памяти:

1. представляет атрибуты сущностей,
2. взаимодействует с сериализацией,
3. осуществляет и поддерживает модульность программ, ограничивая доступ к некоторым аспектам, делая их локальными или приватными.

Пролог: программы только с атомарными переменными

Давайте начнем с простого языка программирования, лишенного возможности структурирования данных, поскольку он не имеет замыканий и не содержит иных типов данных, кроме булева и чисел конечной точности. Вот его BNF-описание с обычной семантикой и обычным приоритетом операций:

```
program ::= def*
def ::= "def" name "(" args ")" block
args ::= "" | name "," args
block ::= "{" statement* "}"
statement ::= "return" exp ";" | name ":=" exp ";" | exp ";" | nest
nest ::= "if" exp block | "if" exp block "else" block | "while" exp
block
exp ::= name | num | exp op exp | exp "(" exps ")" | "(" exp ")" |
unop exp
exps ::= "" | exp "," exps
```

```
unop ::= "!" | "-" | "~"
op ::= logical | comparison | "+" | "*" | "-" | "/" | "%"
logical ::= "||" | "&&" | "&" | "|" | "^" | "<<" | ">>"
comparison ::= "==" | "<" | ">" | "<=" | ">=" | "!="
```

Пример перевода температуры из шкалы Фаренгейта в шкалу по Цельсию на этом языке:

```
def f2c(f) { return (f - 32) * 5 / 9; }
def main() { say(f2c(-40)); say(f2c(32)); say(f2c(98.6));
say(f2c(212)); }
```

Давайте условимся, что в нашем языке программирования запрещена рекурсия, и применяется жадная (eager) стратегия вычислений с вызовом по значению (call-by-value). Также договоримся, что все переменные являются неявно локальными и инициализируются нулями при вызове подпрограмм. Таким образом, ни одна подпрограмма не может иметь каких-либо побочных эффектов. В такой форме наш язык программирования применим только для программирования конечных автоматов. Если скомпилировать такую программу под реальный процессор с регистрами, то каждой переменной, встреченной в тексте программы, может быть назначен один регистр. Каждой подпрограмме также может быть назначен регистр для адреса возврата. Кроме того, потребуется ещё один регистр для счетчика команд. Запуск программы на этом языке на машине с гигабайтами оперативной памяти не даст никакой выгоды. Она никогда не сможет использовать больше переменных, чем было в начале.

Это не делает такой язык бесполезным. Есть много полезных вычислений, которые можно сделать в ограниченном

пространстве. Но это действительно не позволяет в полной мере использовать его абстрактную мощь, даже для таких вычислений.

Для доступа к дополнительной памяти, вы можете использовать функции peek() и poke(), читающие или записывающие один байт по указанному адресу. Таким образом, можно было бы действительно эффективно использовать память:

```
def strcpy(d, s, n) {
    while n > 0 { poke(d + n, peek(s + n)); n := n - 1; }
}
```

Однако многие языки программирования не предоставляют даже peek() и poke(). Взамен они предоставляют некую структуру поверх аскетичного однородного массива байтов.

Например, даже при программировании конечных автоматов, вложенные записи (nested records), массивы (arrays) и объединения (unions) уже предоставляют огромные преимущества.

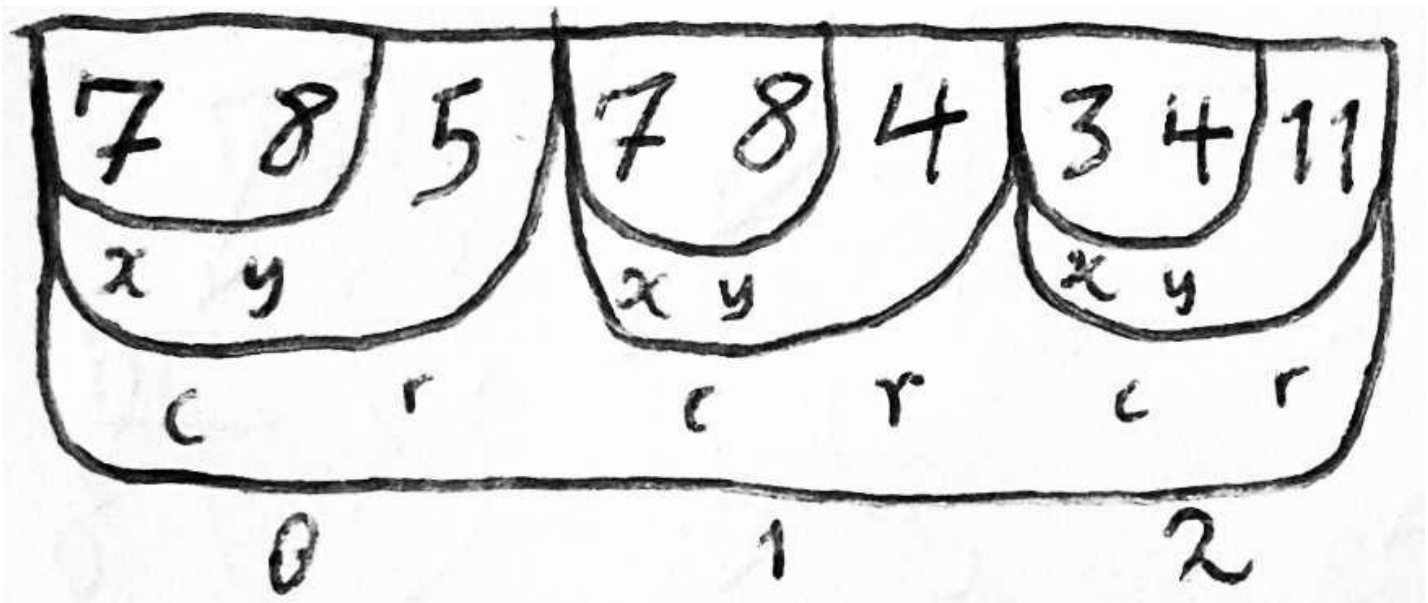
Вложенные записи и модель памяти COBOL: память как декларация о доходах

В COBOL объектом данных является либо что-то неделимое:

- основополагающий объект типа строки или числа определенного размера,
- либо совокупность объектов, такая как:

- запись (объекты различного типа, сохраненные рядом один за другим);
- объединение (альтернативные объекты, которые хранятся, занимая одно и то же место);
- или массив (определенное количество объектов одного и того же типа, сохраненные рядом один за другим).

(Здесь мы значительно отклонились от терминологии и таксономии COBOL, чтобы упростить понимание того, что предлагает этот язык).



В такой модели памяти, если у вас есть несколько подобных сущностей, то каждая из них будет иметь запись одного и того же типа (и по размеру, и по подполям), предназначенную для хранения информации. Таким образом, вся информация о данной сущности будет располагаться в памяти последовательно. Вы сможете очень просто загружать и хранить эти куски смежных данных на диске, ленте, перфокартах или любых других носителях информации. Если несколько из таких записей находятся в памяти

одновременно, их можно поместить в массив.

Любой атрибут сущности определяется двумя байтами, представляющими собой смещение начала и конца этого атрибута относительно базового адреса записи, хранящей данные этой сущности. Например, объект Счет может иметь поле Владелец, занимающее байты с 10 по 35, внутри которого с 18 по 26 байт будет храниться Отчество владельца счета.

О таком подходе можно сделать несколько интересных замечаний.

Там вообще нет никаких указателей. Это означает, что не существует никакого способа сделать динамическое выделение памяти, не получится разыменовать нулевой указатель, никак не перезаписать область памяти, используя висячий указатель (хотя, если две переменные объявлены как REDEFINES, то они, конечно, могут затереть друг друга, и избежать этой проблемы помогут Tagged unions), никакой нехватки памяти, никакого наложения и никакого расхода памяти на указатели.

С другой стороны, это также означает, что каждая структура данных в вашей программе имеет строгое ограничение, и единственный способ использовать ту же память для различных вещей в разное время — это рискнуть использовать их одновременно.

Вложенные записи весьма экономны с памятью. Вам нужно держать в памяти только данные тех сущностей, с которыми вы работаете в данный момент. Это означает, что вы сможете

успешно обрабатывать мегабайты данных на машине с килобайтами памяти, что и делали программисты на COBOL в 1950-х годах.

Каждая часть данных (поля, подполя и т.п.) имеет одного уникального родителя (за исключением самого верхнего уровня), который сразу же содержит все дочерние данные.

В этой модели памяти, если одна часть программы имеет собственную память (например, стековый кадр или статическую приватную переменную), она может сделать приватными некоторые сущности за счет хранения их данных в этой собственной памяти. Это полезно, если нужно создать локальную временную переменную и быть уверенным, что это не повлияет на выполнение остальной части программы. Однако такая модель памяти не позволяет любой части программы сделать атрибут приватным.

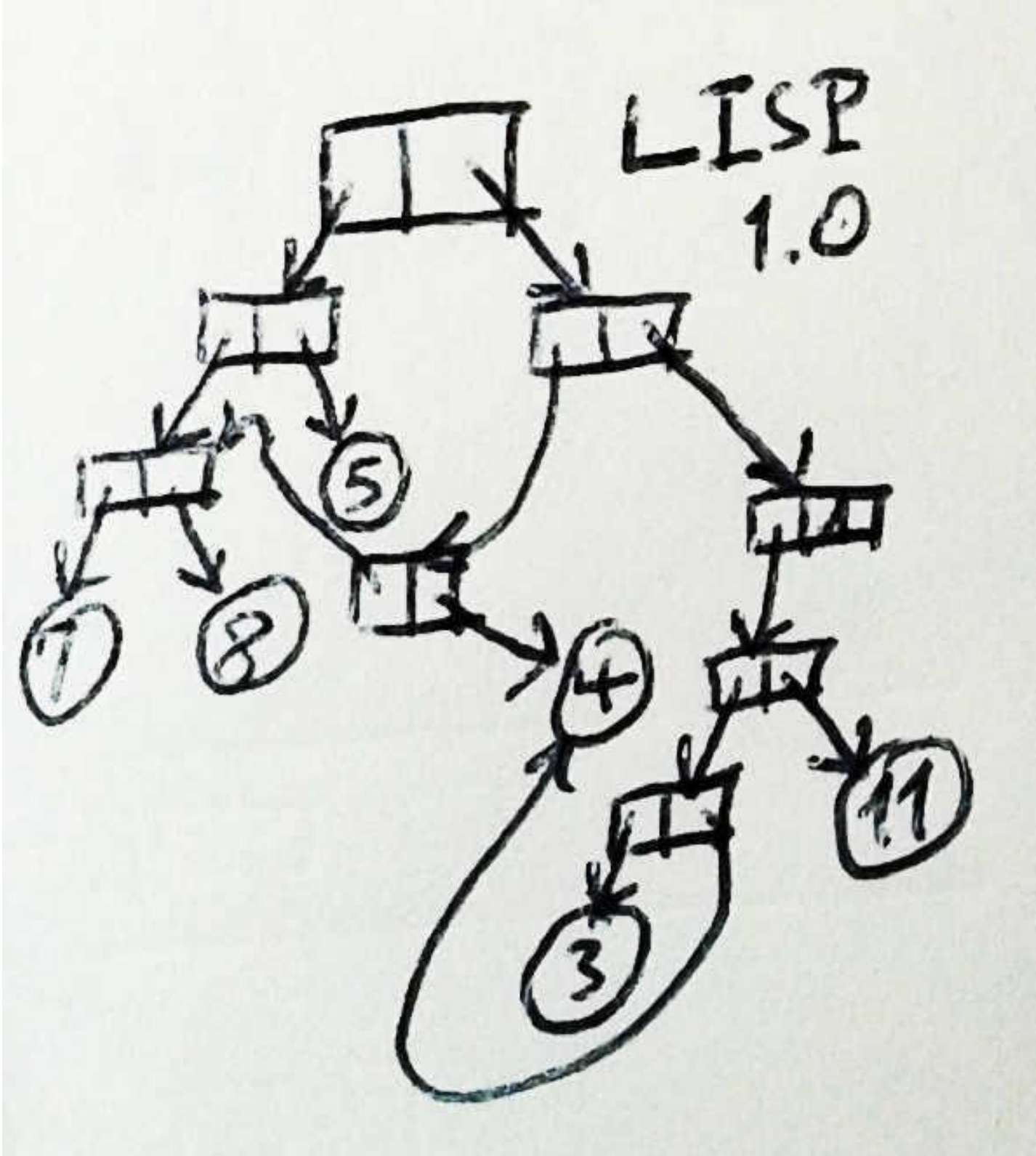
ALGOL (и ALGOL-58 и ALGOL-60) позаимствовал запись (record) из COBOL в качестве основного, помимо массивов, механизма структурирования данных. И именно от Алгола почти все другие языки программирования унаследовали его в той или иной форме.

Язык Си имеет почти весь набор средств для структурирования данных: примитивные типы (char, int и т.д.), структуры, объединения и массивы. Однако в Си также имеются указатели и подпрограммы, которые не только принимают аргументы, но являются рекурсивными, требуя чего-то вроде выделения стека.

Оба этих расширения модели COBOL пришли из LISP.

Объектные графы и модель памяти LISP: память как маркированный ориентированный граф

LISP (это сейчас он Lisp, а в 1959 был LISP) вряд ли мог бы ещё сильнее отличаться от COBOL. Мало того, что в нем есть указатели, так кроме них в LISP больше почти ничего и нет. Единственным механизмом структурирования данных в LISP является нечто, называемое cons и состоящее из двух указателей, один из которых называется «car», а другой «cdr». Значением любой переменной является указатель. Это может быть указатель на cons, или указатель на символ, или указатель на число, или иногда даже указатель на подпрограмму, но это указатель.



Кроме того, там есть рекурсивные подпрограммы с аргументами, и благодаря этому, а также оптимизации хвостовой рекурсии, вы можете писать программы, которые делают что-либо, вообще никогда не изменяя значение переменной.

На любой объект может ссылаться любое количество указателей, посредством любого из которых объект может быть модифицирован. Таким образом, объект не имеет уникального родителя.

Эта модель является чрезвычайно гибкой в том смысле, что она позволяет довольно легко писать программы для обработки естественных языков, программной интерпретации и компиляции, полного перебора и символьных вычислений. Она также позволяет легко создать структуру данных (например, красно-черное дерево) один раз, а затем применить его ко многим различным типам объектов. В противоположность этому, COBOL-производные языки, такие как Си, испытывают значительные трудности с такого рода обобщениями, в результате чего программистам снова и снова приходится писать огромное количество повторяющегося кода для реализации всё тех же хорошо известных структур данных и алгоритмов для новых типов данных.

В то же время, однако, эта модель плохо справляется с ограничениями памяти, подвержена ошибкам и требует много изобретательности для эффективной реализации. Поскольку каждый объект идентифицируется только указателем, то каждый объект может иметь псевдонимы. Каждая переменная может быть пустым указателем. Поскольку указатель может указывать на что угодно, ошибки типов (когда указатель на объект одного типа хранится в переменной, которая, как ожидается, должна указывать на какой-то другой тип) встречаются повсеместно, и объектно-графовые языки для сокращения длительности отладки

традиционно используют проверки типов во время выполнения.

В этой объектно-графовой модели памяти, если у вас есть несколько объектов одного и того же типа, каждый из них будет идентифицирован указателем, и нахождение конкретного атрибута объекта включает в себя навигацию по графу объектов, начиная с этого указателя. Например, если у вас есть объект Счет, то вы можете представить его в виде ассоциативного списка. Затем, чтобы найти его владельца (объект, который может использоваться совместно с другими объектами счетов), вы идете по списку, пока не найдете cons, чей car будет ACCOUNT-HOLDER, и берете его cdr. Затем, чтобы найти отчество владельца счета, возможно, ищете в векторе атрибутов владельца счета, получаете указатель на соответствующее имя, которое может быть либо строкой, либо символом, как в старых Лиспах, в которых нет строк. Обновление второго имени может включать в себя изменение этой строки, обновление вектора для указания на новую строку, или построение нового ассоциативного списка с новым объектом владельца счета, в зависимости от того, используется ли этот объект владельца с другими объектами счетов, и является ли желательным для других счетов, чтобы отчество тоже обновилось.

Сборка мусора — почти необходимость в этих языках. Начиная с 1959 года, когда Маккарти создал Лисп, и до 1980, когда Либерман и Хьюитт придумали использовать поколения при сборке мусора, программы, использующие такую модель памяти, тратили от трети до половины времени своей работы на сборку мусора. Некоторые компьютеры даже специально создавались с несколькими процессорами для того, чтобы сборщик мусора можно было

запустить на отдельном процессоре.

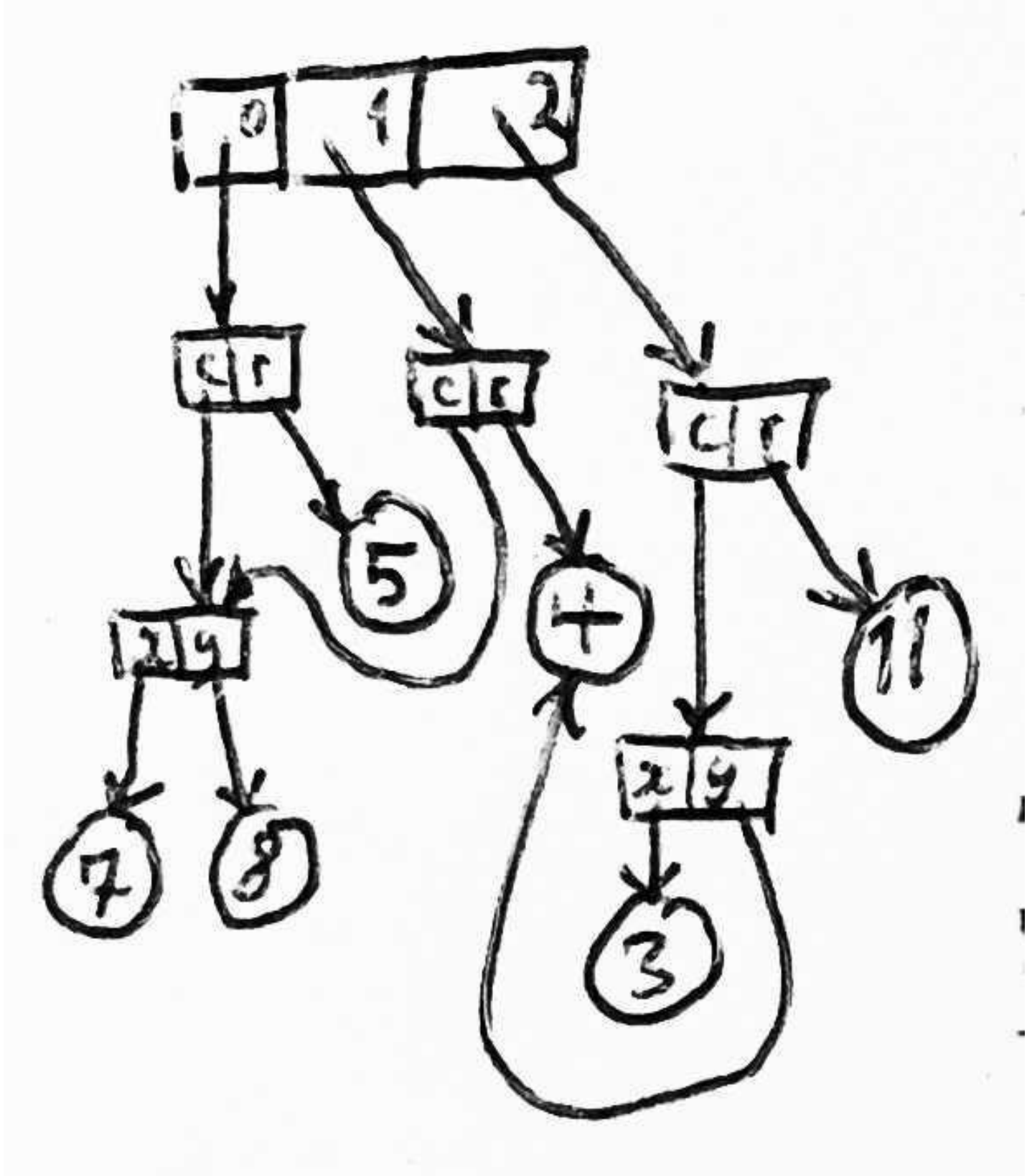
Объектно-графовые языки предъявляли высокие требования к сборщикам мусора, и не только потому, что они предпочитают создавать новые объекты, а не изменять существующие, но и потому, что они, как правило, имеют много указателей. В COBOL-производных языках, таких как Си и Golang, сборщикам мусора работать легче, потому что там выполняется меньше крупных операций с памятью; эти языки имеют тенденцию модифицировать объекты вместо того, чтобы заново выделять память для измененных версий. И программисты, как правило, стараются использовать вложенные записи, где это возможно, а не связывать их с указателями, так что указатели встречаются только там, где желательны полиморфизм, допустимость пустых указателей (который можно рассматривать как частный случай полиморфизма) или наложения.

Сериализация объектного графа немного сложнее потому, что он может содержать циклические ссылки, а также потому, что часть, которую вы хотите сериализовать, может содержать ссылки на части, которые вы не хотите сериализовать, и оба варианта вы должны рассматривать как специфичные случаи. Например, в некоторых системах экземпляр класса содержит ссылку на свой класс, а класс содержит ссылки не только на текущие версии всех методов, но и на класс-родитель. А вы, возможно, не хотите сериализовать весь байт-код класса при каждой сериализации объекта. Кроме того, когда вы десериализуете два объекта, которые ранее имели общие ссылки (два счета для одного и того же владельца счета), то вы, вероятно, хотите сохранить их

совместное использование. В общем, конкретная политика по этому вопросу может варьироваться в зависимости от того, с какой целью вы сериализуете данные.

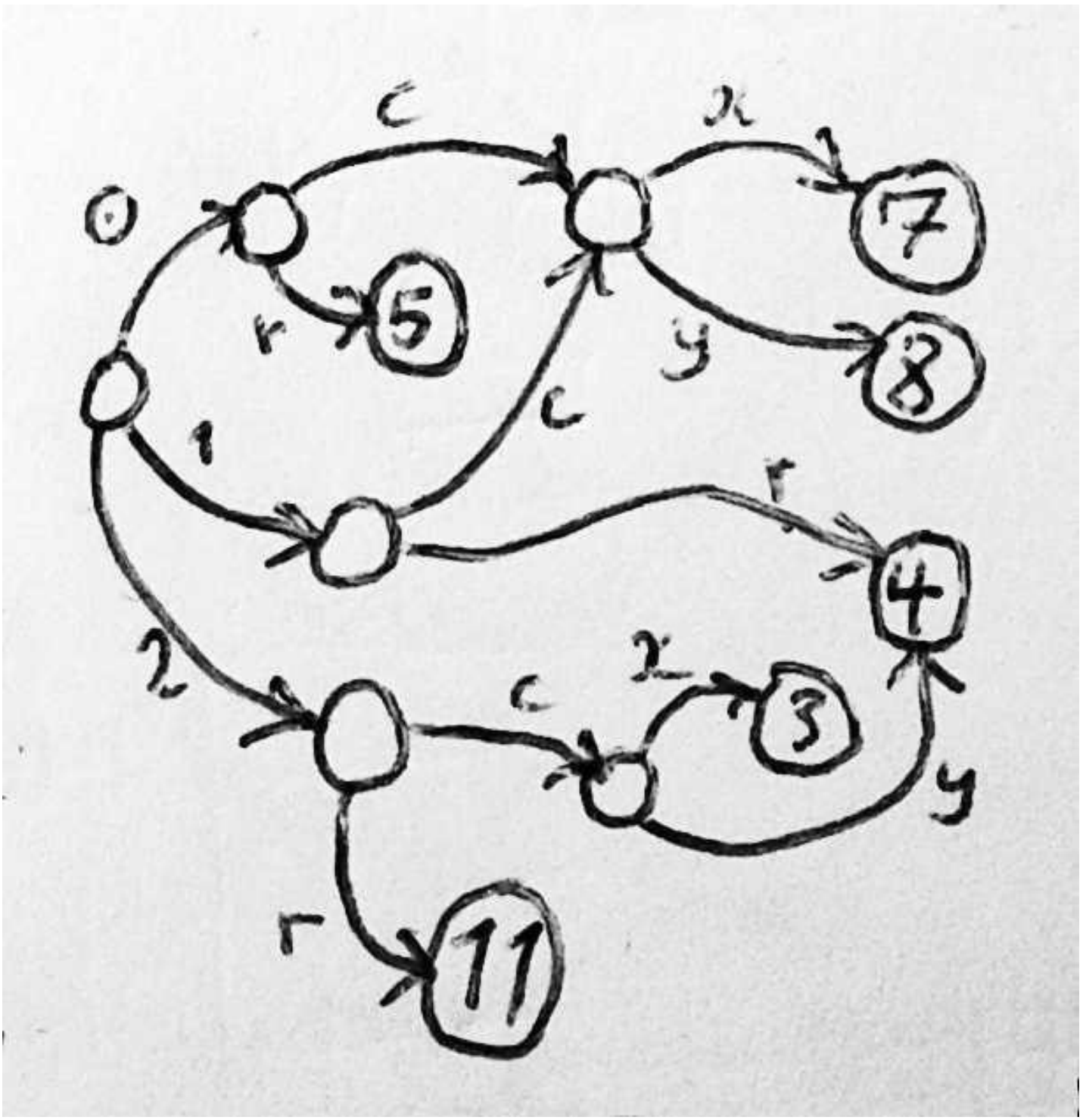
Подобно модели памяти с вложенными записями, модель объектного графа позволяет сделать все атрибуты конкретной сущности локальными по отношению к конкретной части вашей программы — вы просто не даете каких-либо ссылок на её структуру данных для остальной части программы, — но не позволяет сделать приватным конкретный атрибут всех сущностей. Однако, в отличие от модели памяти с вложенными записями, модель объектного графа уменьшает зависимость от размера памяти любого узла, что открывает дверь в объектно-ориентированное наследование и позволяет делать атрибуты приватными, несмотря на ряд серьезных проблем.

Эту модель используют большинство популярных сегодня языков программирования. Не только текущие Лиспы, но также Haskell, ML, Python, Ruby, PHP5, Lua, JavaScript, Erlang и Smalltalk. Все они расширили набор типов объектов, которые существуют в памяти за пределами простой пары; как правило, они включают в себя массивы указателей и хэш-таблиц строк, или указателей на другие указатели. Некоторые из них также включают в себя Tagged unions и неизменяемые записи. Хэш-таблицы, в частности, предлагают своего рода способ добавить новые свойства существующих объектов, в большинстве случаев не затрагивая другой код, который использует эти объекты.



В общем, в этих языках вы можете только следовать по ребрам графа в направлении, которое они указывают, и метки ребер должны быть уникальными в пределах узла-источника (у cons есть

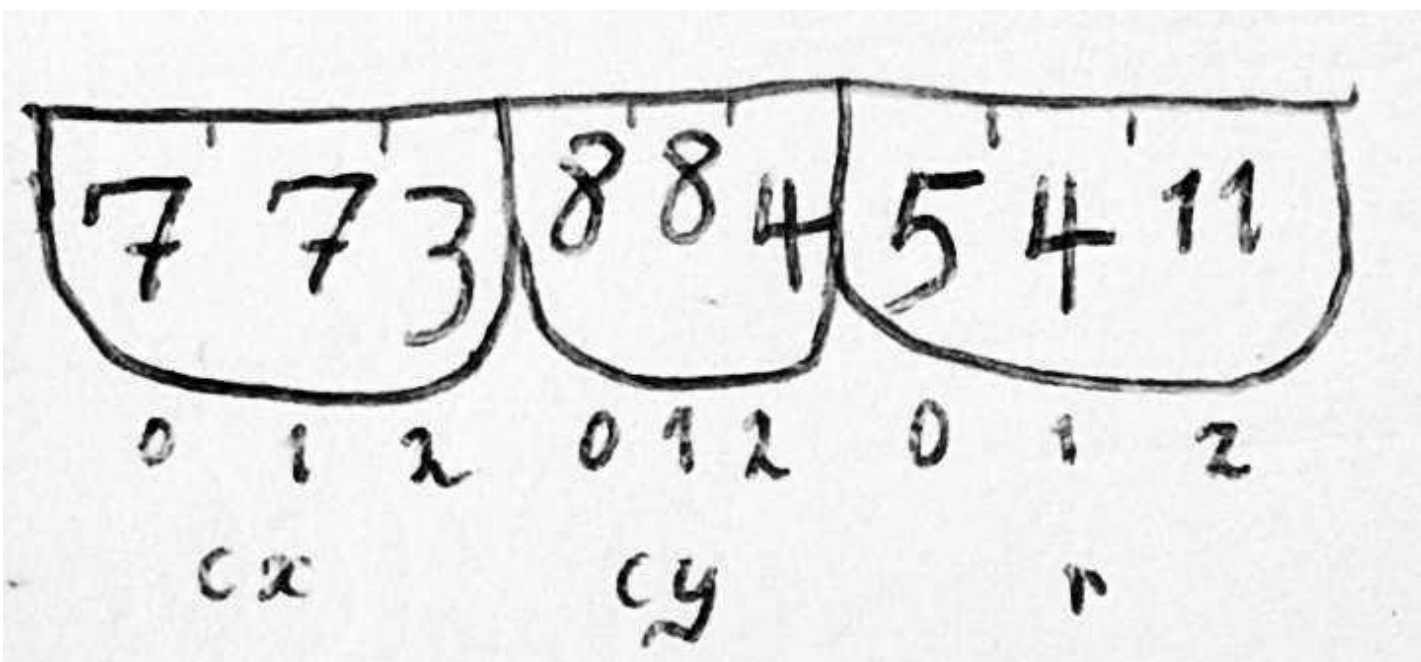
только один car, не два и не десять), но не узла-назначения. Структура данных ZigZag Теда Нельсона представляет собой исследование ситуации, когда вам требуется, чтобы они были уникальны и в источнике, и в назначении. UnQL представляет собой, в каком-то смысле, исследование, которое полностью исключает уникальность.



Java (и C#) используют слегка модифицированную версию этой модели памяти: в Java, например, есть такие вещи, как «примитивные типы», которые не являются указателями.

Параллельные массивы и модель памяти FORTRAN: память как группа массивов

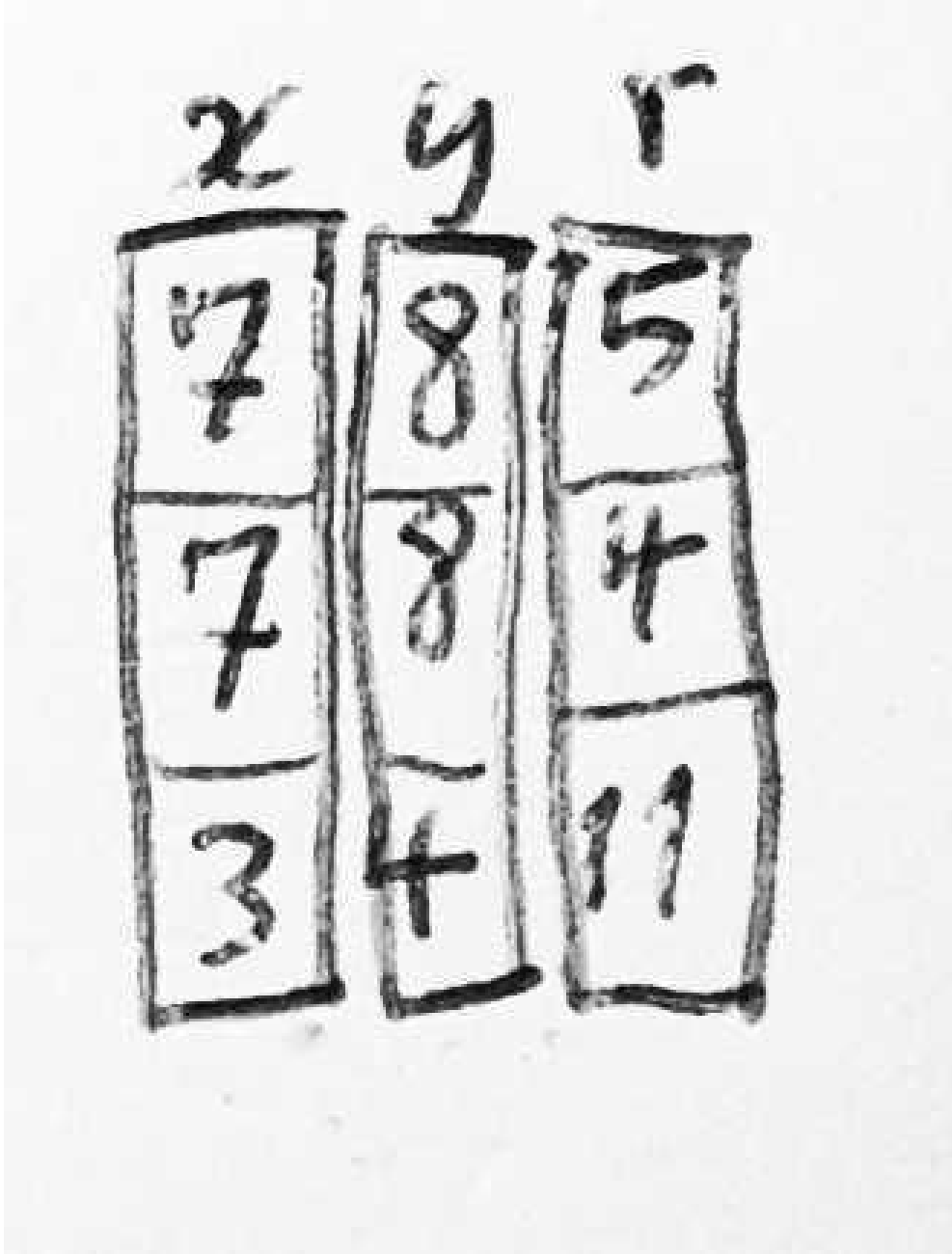
Фортран был разработан для численного моделирования физических явлений, — так называемые «научные вычисления», — что было одним из самых ранних применений компьютеров. В то время научные компьютеры отличались от «бизнес-компьютеров» на языке COBOL целым рядом особенностей: они использовали двоичные числа вместо десятичных; у них не было типа данных байт — только слово; они поддерживали математические операции с плавающей запятой; и у них было более быстрое вычисление и более медленный ввод/вывод.



Как правило, такое моделирование подразумевало много

линейной алгебры с большими массивами чисел, которые должны быть обработаны как можно скорее. И FORTRAN был оптимизирован как раз для этого — для эффективного использования многомерных массивов. В FORTRAN не только не было рекурсивных подпрограмм, указателей и записей, сначала в нем вообще не было подпрограмм!

Когда подпрограммы в нём появились, то имели параметры, которые могли быть массивами. В Алгол 60 это так и не было толком реализовано. Так как массивы были единственными непримитивными типами, то единственно возможными типами элементов для массивов были примитивные типы, такие как целые числа или числа с плавающей запятой.



В модели памяти параллельных массивов, которая развилась в среде FORTRAN, если у вас есть несколько объектов одного и того же типа, то каждый из них будет идентифицирован числовым смещением, которое валидно для нескольких массивов. И поиск приватного атрибута конкретного объекта включает в себя индексацию массива для этого атрибута с индексом такого объекта. Если немного отойти от 1950-х и позволить себе примитивный тип символа, из которого можно формировать массивы, придерживаясь параллельных массивов для

структурирования данных, то определить второе имя владельца счета из предыдущего примера можно следующим образом:

1. IM = IMDNAM(ICCHLD(IACCTN))
   IA = ISTR(IM)
   IE = ISTR(IM+1)
   После этих четырех операций индексации массивов второе имя владельца счета находится в символах [IA, IE) массива CCHARS.
2. IM = IMDNAM(IACCTN), затем действуйте как в предыдущем варианте, если у вас нет отдельного набора символов для атрибутов владельцев счетов.
3. Вместо IMDNAM используйте CMDNAM, N*12 массив символов, с одной 12-символьной колонкой для второго имени каждого владельца счета.

В этой модели памяти подпрограмма может получить доступ к любому индексу в массиве, который был передан в качестве аргумента или доступен ему как-то иначе, считывать или записывать его любое количество раз в случайном порядке.

Это именно то, что означает фраза «писать на FORTRAN можно на любом языке»: почти в любом языке программирования есть массивы примитивных типов. Даже на ассемблере или Forth несложно делать массивы. Awk, Perl4, и Tcl вдобавок предоставляют словари, которые не являются объектами первого класса, поскольку это не объектно-графовые языки, хотя они отлично работают вместо массивов для хранения атрибутов объектов, позволяя идентифицировать объекты строками вместо

целых чисел.

Интересно, что на машинном уровне, в простом случае, параллельные массивы генерируют почти такой же код, как и структурные элементы, ссылающиеся через указатели, как во вложенной модели. Например, здесь `b->foo`, где b — указатель на struct с 32-х битным членом foo:

```
40050c:   8b 47 08                mov    0x8(%rdi),%eax</code>
А также здесь <code>foos[b]</code>, где b – индекс в массиве foos,
состоящем из 32-х битных элементов:

<source>400513:   8b 04 bd e0 d8 60 00    mov
0x60d8e0(,%rdi,4),%eax
```

В обоих случаях мы добавляем непосредственную константу, представляющую нужный нам атрибут, к переменной в регистр, который указывает, какой объект мы рассматриваем. Во втором случае отличие в том, что мы умножаем индекс на размер элемента, и непосредственная константа немного больше.

(Вы заметите, что, хотя формат инструкции отличается, и не все типы архитектур процессоров поддерживают подобные большие подмножества постоянных, базовый адрес массива или достаточно большие подмножества полей структур на некоторых машинах могут быть загружены в регистры).

Адам Н. Розенберг защищал программирование в стиле параллельных массивов на протяжении целой книги. Субъективно, это не такая уж хорошая идея, но Розенберг дает наиболее современное ее объяснение, какое я только встречал.

Если вкратце, то параллельные массивы являются cache-friendly, поддерживают разную видимость для разных атрибутов и установку контрольных точек, обеспечивают последовательность, которая может быть значимой, и поддерживают многомерное индексирование (где атрибут — это свойство ряда объектов, а не одного из них). Нужно добавить, что они также позволяют писать подпрограммы, которые абстрактны для атрибутов, поскольку они овеществляют каждый атрибут в среде выполнения: вы можете написать функцию sum или функцию covariance, которые могут быть применены для произвольных атрибутов.

Второй момент особенно интересен: хотя параллельные массивы не позволяют делать объект приватным в определенной части программы, они позволяют делать приватные атрибуты.

К сожалению, использование параллельных массивов может приводить к ошибкам. Компилятор не может сказать, какие массивы в определенном индексе валидные, а какие нет, также этого не сделает отладчик или другие инструменты. Использование параллельных массивов означает, что ошибка несоответствия типов (type error) — сохранение идентификатора для определенного объекта в переменной, которая должна включать идентификаторы для других типов объектов — во время компилирования или выполнения очень часто дает неправильные ответы вместо сообщения об ошибке. (Особенно, если у вас не включена проверка границ массива). Кроме того, поскольку в среде выполнения параллельные объекты овеществляют атрибуты, а не объекты, то создание и уничтожение объектов

может приводить к ошибкам, подпрограммы будут содержать большое количество параметров (что увеличивает размер подпрограмм), и у вас часто будет возникать та же проблема, что возникает при модели вложенной записи (nested-record model).

Параллельные массивы очень просты и эффективны в сериализации, особенно в том случае, если вы не слишком озабочены проблемой портируемости на различные архитектуры, но они часто требуют сериализации или десериализации атрибута для всех объектов, к которым он одновременно применяется.

FORTRAN — не единственный язык программирования, который стимулирует организовывать память при помощи параллельных массивов. Octave, Matlab, APL, J, K, PV-WAVE IDL, Lush, S, S-Plus и R тоже ориентированы на использование параллельных массивов. Numpy, Pandas и OpenGL — «параллельные» библиотеки, Perl4, awk, и Tcl, как говорилось выше, тоже в какой-то степени ориентированы на использование параллельных массивов. Некоторые из них компенсируют часть рисков параллельных массивов, или хотя бы побуждают создавать новые массивы. Pandas, K и параллельно-словарный вариант (parallel-dictionary variant) снижают риск появления ошибок, побуждая вас индексировать ваши массивы без использования целых чисел.

Различные функции современного аппаратного обеспечения вынуждают использовать параллельные массивы для повышения производительности: увеличивается разрыв между скоростью CPU и скоростью памяти, в GPU применяется SIMT-архитектура, в CPU добавлены SIMD-инструкции ради увеличения отношения ALU к

управляющим структурам процессора. В результате разработчики игр вынуждены возвращаться к использованию параллельных массивов, что называют «архитектурой, ориентированной на обработку данных» (data-oriented design) или (в некотором роде) «системами объектов» (entity systems). В целом, в среде «научных» вычислений никогда не забывали о параллельных массивах.

Интерлюдия: почему нет моделей памяти Lua, Erlang или Forth?

Эти три модели памяти примерно соответствуют трем основным типам структур данных: запись (record), связный список и массив. Но мы знаем, что есть несколько других структур данных, которые применяются повсеместно:

- Lua использует конечные карты (finite maps) (словари, которые здесь называют «таблицами»),
- Erlang использует активные процессы без разделения ресурсов, которые ставят сообщения друг другу в очередь (вариант модели акторов, созданной Хьюиттом (Hewitt) и Агой (Agha)),
- Forth использует стеки.

Lua и Erlang по сути объектно-графовые языки. В программах на Forth часто используются параллельные массивы. Похоже, вы можете создать систему, где всё организовано как хэш-таблица или определенная разновидность не-граф-структурированного объединения, или два или больше стека. Одним из примеров исследований в области не-граф-структурированного

взаимодействия между акторами в пространстве кортежей является Linda, которая может быть расширена до целого языка программирования.

Но в конце мы рассмотрим другие альтернативы.

Пайпы и модель памяти для магнитных лент

Пайпы Unix — самый простой тип памяти (и у них есть аппаратные аналоги). Единственный вид операций, которые они поддерживают — запись байта (или, при оптимизации, нескольких байтов), чтение байта и, вероятно, закрытие (с любой стороны). (Реальная магнитная лента пишет и читает блоками, но мы этот факт пропустим). В целом, вы не сможете читать и записывать в один и тот же пайп одной и той же программы.

Такой тип памяти позволяет лишь добавлять и хранить данные, что приемлемо для некоторых алгоритмов. MapReduce недалеко ушел от этого, но типичная проблема токенизации с помощью lex, например, заключается в использовании такого минимального интерфейса для его ввода.

Итераторы и генераторы Python, однонаправленные итераторы (forward iterators) C++ STL, однонаправленные диапазоны (forward ranges) D, каналы Golang —всё это примеры таких пайпов или каналов, с их чисто последовательным доступом к данным.

На что может быть похожа модель памяти языка программирования, основанная лишь на пайпах? Вам понадобятся

операции для считывания элементов данных из пайпов (вероятно, примитивные типы) и их записи. Давайте рассмотрим пример языка программирования из введения к этой статье. Даны пайпы с подпрограммами empty, get, и ``put, вы можете написать функцию слияния, которую можно использовать для сортировки слияния, хотя это не так просто сделать на Python или чем-то еще:

```
def merge(in1, in2, out) {
    have1 := 0;
    have2 := 0;
    while !empty(in1) || !empty(in2) {
        if 0 == have1 && !empty(in1) {
            val1 := get(in1);
            have1 := 1;
        }
        if 0 == have2 && !empty(in2) {
            val2 := get(in2);
            have2 := 1;
        }
        if 0 == have1 {
            put(out, val2);
            have2 := 0;
        } else {
            if 0 == have2 || val1 < val2 {
                put(out, val1);
                have1 := 0;
            } else {
                put(out, val2);
                have2 := 0;
            }
        }
    }
}
```

Обратите внимание, что указанная выше функция и сортировка слиянием лишь предполагают, что здесь минимум четыре пайпа. Их ёмкости достаточно для данных, и они могут отличаться между собой по объёму. Здесь не требуется создавать или уничтожать

пайпы, или пропускать пайпы через пайпы. Аргументы in1, out и т.д. не должны быть чем-то вроде объектов пайпа первого класса, они должны быть лишь целыми числами. (Именно так Unix-программы работают с Unix-пайпами: по индексу дескриптора файла). Или они также могут быть объектами пайпа первого класса, которые могут быть переданы в качестве аргументов, но не через сами пайпы.

Можете представить себе многопоточную (multithreaded) систему управления потоком команд (control flow system), в которой вы можете форкнуть различные потоки (threads), а также потоки, которые пытаются считывать из блока пустых потоков (empty streams block), пока данные не станут доступными. Вы должны использовать пайпы вместо массивов или записей; вероятно, вам пришлось бы использовать один поток для каждого атрибута. Затем среда выполнения может по вашему указанию назначить различные потоки обработки пайпов, вероятно, с запуском некоторых из них на разных машинах.

Пи-исчисления (π-calculus) — язык, который использует только пайпы. По сути, это параллельная (concurrent) канало-ориентированная (channel-oriented) альтернатива λ-исчислениям. Как объясняет Джаннет Винг:

> Давайте введем P и Q в качестве обозначения процессов. В этом случае:
>
> - P|Q обозначают процесс, составленный из P и Q, выполняющихся параллельно;

- a(x).P обозначает процесс, который ожидает записи значения x из канала a, и потом, получив значение, ведет себя как P;
- ā〈x〉.P обозначает процесс, который ожидает отправки значения x в канал a, и после того, как x успешно одобрен каким-либо вводным процессом, ведет себя как P.
- (νa)P гарантирует, что a — это свежий канал в P;
- ! P обозначает бесконечное количество копий P, все выполняются параллельно;
- P + Q обозначает процесс, который ведет себя как P или Q;
- 0 обозначает инертный процесс, который ничего не делает.

Пример кода пи-исчисления:

```
!incr(a, x).ā〈x+1〉 | (νa)( i‾n‾c‾r‾〈a, 17〉 | a(y)
)
```

В результате запускается «сервер» с ограниченным числом процессов, каждый из которых слушает канал incr и ждет одно сообщение, раскладывает его на a и x, затем отправляет x+1 в канал a и завершается. Всё это происходит параллельно с двумя другими параллельными процессами, вместе использующими свежий канал с названием «a». Один из них отправляет 17 в канал incr, в то время, как другой ожидает отсылки сообщения в этот канал, прикрепляя его к y.

Пи-исчисление, вероятно, не станет практичной системой программирования, равно как и другие подобные системы исчисления, но она хотя бы предполагает возможность программирования с использованием в роли памяти только

пайпов. Однако обратите внимание, что вышеприведённая версия тянет за собой использование кортежей и помеченных графов процессов (labeled graphs)! И если вы будете ими пренебрегать, то я не уверен в универсальности применения этой системы исчисления.

«Потоковое программирование», реализованное в проекте NSA, известном как Apache NiFi, представляет собой другой подход к генерализации парадигмы Unix-пайпы-и-фильтры.

Директории и модель памяти Multics: память представляет собой дерево строк (string-labeled tree) с blob-листьями

Иерархическая файловая система Unix (или Windows, или MacOS, или Multics) — это иной тип организации памяти, как правило это долговременная память, созданная на основе жёстких дисков и активно используемая shell-скриптами. В своей исходной форме она представляет собой ограничение объектно-графовой модели, в которой у каждого узла — единственный родитель, и это либо «директория» со строчно-проиндексированными словарями линков на дочерние узлы, либо «обычный файл», изменяемый блоб (последовательность байтов). Качественным улучшением является третий тип узла, «символический линк», который представляет собой путь следования от корня дерева или своей родительской директории для нахождения желаемого узла — если он вообще существует.

Обычно в иерархической файловой системе для представления

свойств (атрибутов) объекта вы сериализуете их в обычный файл, вместе со многими другими объектами! Но так делается, в основном, потому, что интерфейсы системных вызовов медленные и неповоротливые, и потому что накладные расходы на узел обычно составляют порядка сотен байтов. Вы можете использовать директорию каждого объекта, сохраняя значение свойств (атрибутов) «x» в файле «x». Именно так поступает Unix при хранении данных о пользователях, например, или различных версий программного пакета, или о данных том, как компилировать различные программные пакеты.

Иерархическая файловая система — это компромисс в случае локальных переменных. Обычно любые узлы дерева доступны при навигации по нему, но вы можете быть уверенными в том, что ваше ПО, запущенное в одной части файловой системы, не сможет обнаружить файлы, которые вы создали в другой части системы. В большинстве случаев вы можете добавлять новые файлы в директории без остановки ПО, которое уже использует эти директории, хотя работа этой схемы не гарантируется.

Уникальное родительское свойство делает сериализацию и десериализацию относительно простой.

Есть несколько систем программирования, которые работают примерно по такому принципу. На это более-менее похожа MUMPS, до сих пор используемая Министерством по делам ветеранов США (хотя обычно «файлы» ограничены 4096 байтами, а узел может быть как «директорией», так и «обычным файлом»). База данных IBM IMS, которая также до сих пор в ходу, имеет

схожую модель данных, здесь узлы называются «сегментами». Также в базу данных внедрена схема. Несколькими годами ранее Марк Ленцнер (Mark Lentczner) начал разрабатывать современное, объектно-ориентированное программное окружение, которое он назвал Wheat. Оно основано на этой модели для разработки веб-приложений и более качественной альтернативе PHP. В Wheat каждая запись активации функции является «директорией» с переменными в качестве «файлов» (динамически типизированными, в отличие от блобов). Некоторые поддеревья «файловой системы» постоянны, но не все. Обеспечивается прозрачный доступ к удаленно сохраненным данным. Wheat позволяет предотвратить потерю согласованности между устойчивыми, иерархически именованными, глобально доступными веб-ресурсами и собственной концепцией памяти программы.

Отношения и модель памяти SQL: память — это коллекция изменяемых многозначных конечных функций

Пожалуй, это самая абстрактная из всех моделей памяти.

«Многозначная функция» в математике обычно называется «отношением» или «зависимостью». cos — это функция: для любого θ у cos(θ) существует единственное значение. Cos-1 — это отношение: у $\cos^{-1}(0.5)$ много значений, хотя мы превращаем выражение в функцию, выбирая лишь какое-то одно. Вы можете думать о cos как о наборе упорядоченных пар, например: (0, 1), (π/2, 0), (π, -1), (3π/2, 0) и так далее. И операция инверсии — это

лишь вопрос изменения порядка следования пар: (1, 0), (0, π/2), (-1, π), (0, 3π/2) и так далее.

Однако, отношения, которые мы рассматриваем здесь, более общего характера, чем бинарные отношения: вместо замещения двухкортежных наборов они представляют собой n-кортежные для определённого значения n. Вы можете рассмотреть в качестве примера тернарное отношение между углом, его косинусом и синусом: (0, 1, 0), (π/2, 0, 1), (π, -1, 0) и т.п.

Большинство систем для реляционного программирования имеют лишь ограниченную поддержку для бесконечных отношений вроде cos, поскольку с ними очень легко получить нерешаемые проблемы.

Если говорить по существу, то вот, например, секция из таблицы permissions.sqlite установщика моего Firefox:

```
sqlite> .mode column
sqlite> .width 3 20 10 5 5 15 1 1
sqlite> select * from moz_hosts limit 5;
id   host                 type        permi  expir  expireTime
a  i
---  -------------------  ----------  -----  -----  --------------
-  -
1    addons.mozilla.org   install     1      0      0
2    getpersonas.com      install     1      0      0
5    github.com           sts/use     1      2      1475110629178
9    news.ycombinator.com  sts/use     1      2      1475236009514
10   news.ycombinator.com  sts/subd    1      2      1475236009514
```

С точки зрения реляционной модели, каждый из этих столбцов в некотором смысле является функцией основного ключа, который

находится в колонке id слева; поэтому вы можете заявить host(1) как addons.mozilla.org, host(2) как getpersonas.com, type(5) как sts/use. Пока что это лишь хэш-таблица.

Но есть и отличия от обычного функционального программирования. Речь идет о том, что вы можете вместо host(9), запросить $host^{-1}$('news.ycombinator.com'), который оказывается многозначным:

```
sqlite> select id from moz_hosts where host =
'news.ycombinator.com';
id
---
9
10
```

Более того, вы можете скомбинировать вместе эти многозначные функции:

```
sqlite> .width 0
sqlite> select min(expireTime) from moz_hosts where host =
'news.ycombinator.com';
min(expireTime)
---------------
1475236009514
```

Обычно для того, чтобы представить объект в SQL, вы используете строку в таблице («кортеж» в «отношении»), а для представления связей с другими объектами вы определяете набор уникальных свойств каждого объекта, что называется ключом, как id в вышеприведённой таблице. Затем вставляете ключ одного объекта в какие-то столбцы других объектов. Вернёмся к примеру

с отчеством владельца счета:

```
select accountholder.middlename
from accountholder, account
where accountholder.id = account.accountholderid
and account.id = 3201
```

Или можно просто оставить то, что было в прошлом варианте, и получить отчества всех держателей аккаунтов, а не кого-то одного.

SQL — это не единственная реализация реляционной модели, но наиболее популярная. Недавно она стала полной по Тьюрингу и теперь может завоевать мир.

Всё это может показаться неуместным уходом от реальных языков программирования вроде Lisp, FORTRAN и C, поскольку SQL в качестве языка программирования скорее курьёзный, чем практичный инструмент. Но вспомните язык программирования, приведённый в начале в качестве примера. Если дополнить его вызовом SQL-операторов, хранить результаты в их переменных в виде примитивных типов, и перебирать результирующие наборы, то он становится вполне практичной системой программирования, хотя и несколько неудобной (фактически, этому описанию более-менее соответствует PL/SQL).

Речь идет не об изменении обозначаемых или выраженных типов в языке. Пусть даже язык программирования может обрабатывать числа лишь как переменные значения или значения выражений, но если он способен хранить эти числа в отношениях и извлекать их посредством запросов, то это разительно увеличивает

функциональность такого языка.

У SQL есть проблемы из-за некоторых ограничений: имена таблиц размещаются в глобальном пространстве имен, плюс строки и столбцы в таблицах доступны глобально (у вас не может быть отдельных строк или столбцов, хотя результат запроса в определенном смысле представляет собой отдельную таблицу, и может быть использован в этой роли); каждый столбец может обычно содержать только примитивные типы данных вроде чисел (по крайней мере, современные базы данных не ограничивают ширину строк, как в COBOL). И обычно выполняется всё это очень медленно.

В определенном смысле операции SQL, а также большинство его преимуществ и недостатков, имеют сходство с такими параллельными массивами:

```
for (int i = 0; i < moz_hosts_len; i++) {
    if (0 == strcmp(moz_hosts_host[i], "news.ycombinator.com")) {
        results[results_len++] = moz_hosts_id[i];
    }
}
```

Другой пример того, как SQL может облегчить вам жизнь: выполнение объединения сортировкой/слиянием с параллельными массивами никак не сравнится с неуправляемым (unmanageable) вариантом. Вместо такого SQL:

```
select accountholder.middlename
from accountholder, account
where accountholder.id = account.accountholderid
```

Вы можете сделать что-то вроде этого на Си с параллельными массивами:

```
int *fksort = iota(account_len);
sort_by_int_column(account_accountholderid, fksort, account_len);
int *pksort = iota(accountholder_len);
sort_by_int_column(accountholder_id, pksort, accountholder_len);
int i = 0, j = 0, k = 0;
while (i < account_len && j < accountholder_len) {
    int fk = account_accountholderid[fksort[i]];
    int pk = accountholder_id[pksort[j]];
    if (fk == pk) {
        result_id[k] = fk;
        result_middle_name[k] = accountholder_middlename[pksort[j]];
        k++;
        i++;                // Supposing accountholder_id is unique.
    } else if (fk < pk) {
        i++;
    } else {
        j++;
    }
}
free(fksort);
free(pksort);
```

Здесь iota представляет собой:

```
int *iota(int size) {
    int *results = calloc(size, sizeof(*results));
    if (!results) abort();
    for (int i = 0; i < size; i++) results[i] = i;
    return results;
}
```

А задать sort_by_int_column можно примерно так:

```
void sort_by_int_column(int *values, int *indices, int indices_len)
{
    qsort_r(indices, indices_len, sizeof(*indices),
            indirect_int_comparator, values);
}

int indirect_int_comparator(const void *a, const void *b, void *arg)
{
    int *values = arg;
    return values[*(int*)a] - values[*(int*)b];
}
```

В SQL-реализациях используются различные хитрости для повышения эффективности работы со сложными запросами, но это делается в ущерб эффективности выполнения более простых запросов, хотя это и не влияет на абстракции, видимые пользовательской программой. Ну, не сильно влияет.

Многие говорят, что SQL — это декларативный язык, где вам нет необходимости говорить, как именно вычисляется результат, ведь только результат и важен. Пожалуй, это справедливо лишь отчасти, и декларативность — не бинарный предикат, это, скорее, вопрос уровня абстракции описания, то есть бесконечный континуум. И на практике неизбежно возникают проблемы ниже уровня абстракции, на котором вы программируете, хотя бы по соображениям эффективности.

Более интересны реализации реляционной модели Prolog и miniKANREN, объединяющие в себе реляционное программирование с рекурсивными структурами данных объектных графов. Относительно простая программа miniKANREN может, например, генерировать бесконечную серию программ с выводом их собственного кода за вполне разумное время.

Сегодня также растёт популярность систем программирования с учётом ограничений (constraint programming), позволяющих указать некоторые свойства желаемого ответа («ограничений», которым должен удовлетворять ответ), после чего система принимается искать этот ответ. Популярность этих систем обусловлена разными причинами, но, в основном, это происходит благодаря год от года растущей эффективности алгоритмов решения SAT и SMT.

Похожие публикации

Segmentation Fault (распределение памяти компьютера)

10

ПЕРЕВОД

NIX_Solutions • 26 февраля 2016 в 11:08

Популярное за сутки

Яндекс открывает Алису для всех разработчиков. Платформа Яндекс.Диалоги (бета) 69

BarakAdama • вчера в 10:52

Почему следует игнорировать истории основателей успешных стартапов 20

ПЕРЕВОД

m1rko • вчера в 10:44

Как получить телефон (почти) любой красотки в Москве, или интересная особенность MT_FREE 24

ИЗ ПЕСОЧНИЦЫ

cab404 • вчера в 20:27

Java и Project Reactor 10

zealot_and_frenzy • вчера в 10:56

Пользовательские агрегатные и оконные функции в PostgreSQL и Oracle 6

erogov • вчера в 12:46

Лучшее на Geektimes

Как фермеры Дикого Запада организовали телефонную сеть на колючей проволоке

31

NAGru • вчера в 10:10

Энтузиаст сделал новую материнскую плату для ThinkPad X200s

49

alizar • вчера в 15:32

Кто-то посылает секс-игрушки с Amazon незнакомцам. Amazon не знает, как их остановить

85

Pochtoycom • вчера в 13:06

Илон Маск продолжает убеждать в необходимости создания колонии людей на Марсе

139

marks • вчера в 14:19

Дела шпионские (часть 1)

16

TashaFridrih • вчера в 13:16