СТАНДАРТЫ и КОНЦЕПЦИИ

Никлаус Вирт

ВОЗВРАЩЕНИЕ К ХОРОШИМ ИДЕЯМ

От редакции. Доклад Никлауса Вирта, сделанный им после церемонии присуждения ему звания почетного доктора СПбГУИТМО (основная часть доклада была сделана на английском языке), был записан на магнитофон. Однако в связи с тем, что Вирт иногда отходил от микрофона, а иногда говорил очень тихо, с написанием стенограммы возникли проблемы. Не желая терять уникальный для публикации материал, редакция решила взять на себя ответственность за вольный пересказ этого выступления, проиллюстрировав его кадрами оригинальной презентации Н. Вирта. Оцифрованная аудиозапись доклада (вместе с презентацией) находятся на диске к журналу.

ВСТУПЛЕНИЕ

(Вирт его делает на русском языке)

Уважаемые соратники, дамы и господа!

В информатике как науке и технологии присутствуют быстрые изменения и иногда (!) прогресс. Из-за этого любому человеку довольно трудно быть постоянно в курсе последних изменений, особенно тому, кто уже несколько лет как отошел от активной работы.

Недавние достижения быстро забываются, и кажется устаревшим то, чем мы гордились 20 лет назад. Иногда их важность мы понимаем только через несколько лет, поэтому мы должны распознавать и тщательно отбирать те, которые представляются фундаментальными, а не модными. Я попытался сделать это в программировании.

Инструментом программиста служит язык. Хороший язык служит не только для программирования компьютеров, но снабжает наш ум рассуждениями, позволяет правильно ставить задачи и ведет нас к правильным решениям.

Так было, есть и будет.

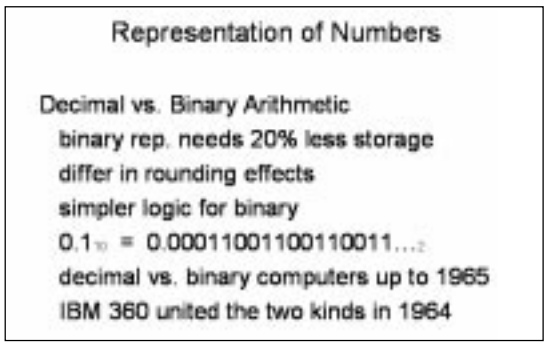
Тем самым, язык программирования есть очень важный аспект обучения. А выбор языка для обучения имеет большее значение, чем обычно считается.

Я высоко ценю присуждение мне звания почетного доктора этого известного университета, в особенности потому, что вы хорошо понимаете важность обучения ясному и систематическому мышлению и проектированию, а также вероятно считаете, что программы, помимо того, что приносят деньги, часто демонстрируют красоту и элегантность. Я глубоко благодарен вам за честь, оказанную мне сейчас здесь в этом замечательном городе, где более 200 лет назад жил и работал мой великий соотечественник Леонард Эйлер...

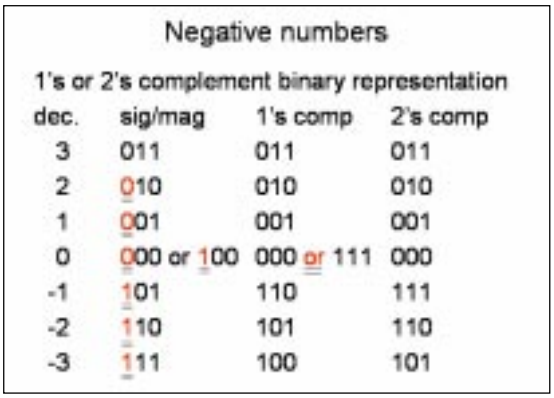
Простите меня, пожалуйста, за мои ошибки в русском языке. Я навещал вашу безграничную страну первый раз 15 лет назад. Второй раз (а первый раз в этом городе) я был 9 лет назад. И тогда я сказал себе, что в следующий раз, может быть, буду немного говорить и уметь читать слова по-русски. Когда я пошел на пенсию, я стал посещать курсы русского языка. В течение трех лет я периодически встречался с заме-

чательной женщиной, чтобы учить русский язык, а чтобы она учила немецкий. Она – элегантная москвичка – конечно умнее и говорит по-немецки лучше, чем я по-русски. Извините также за то, что поэтому я сделаю свой доклад на английском языке. Если бы я говорил все по-русски, это было бы слишком ужасно для вас.

Информатика – предмет не только быстро развивающийся, но уже знаменитый, «бородатый», с множеством красивых идей, и мы, информатики, по праву гордимся проделанной работой. Совсем недавно я слышал лекцию коллеги в Американском колледже, которая называлась «Давайте подумаем о качестве хороших идей». Он сказал, что некоторые идеи, великие идеи совершенно устарели. И тогда я воспарил от гордости, ведь те идеи, которые были предложены мной двадцать лет назад, не нуждаются в изменениях. Я отобрал несколько таких идей и хотел бы представить их в сегодняшнем докладе.



Первой темой я выбрал совсем элементарный предмет – формы представление чисел. Лет 40–50 назад эта тема была важна, ведь тогда как раз нужно было выбрать удобное представление чисел. При вводе данных использовались десятичные числа, но работали компьютеры с двоичной арифметикой. И производители выбрали, что и понятно, двоичное представление, требовавшее на 20% меньше памяти. Тем не менее, сейчас, спустя многие годы, десятичная арифметика выжила, и я бы очень хотел напомнить, что примерно до 1965 г. все большие компьютерные компании имели как серии компьютеров для научных вычислений, так и другие – для коммерческих расчетов. Различие этих двух серий была в представлении чисел. Что касается языков программирования, в научном мире преобладал FORTRAN, а в коммерческом – COBOL. Логические схемы для двоичных чисел проще. Однако было желание помочь и коммерции, ибо бухгалтеры не имели точного представления, как конкретно могли бы быть представлены числа в бинарной системе (и как начислить зарплату «двоичными купюрами»). Двоичные и десятичные числа сосуществовали до 1964, когда компания IBM решила эту проблему, путем продвижения машин IBM 360.

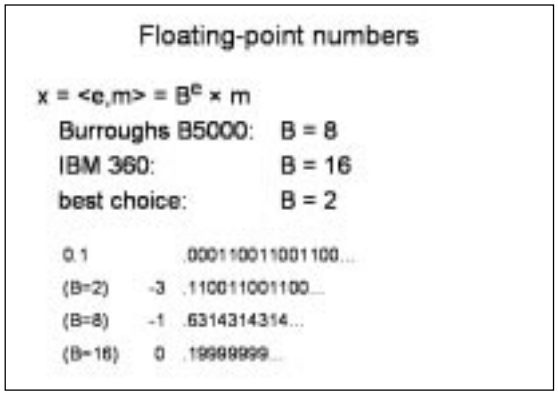


Другая проблема была в том, как представлять отрицательные числа.

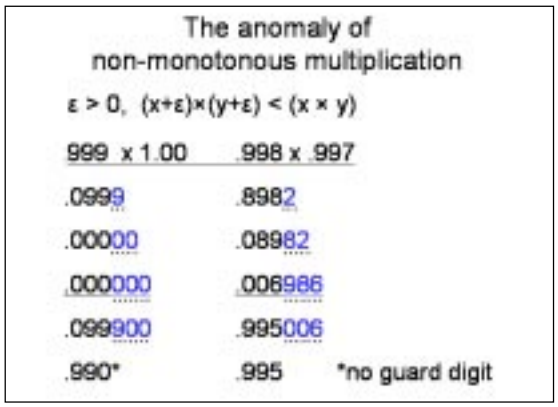
На рисунке* показаны различные способы использования крайнего левого двоичного разряда для записи отрицательных чисел. В левом столбике отдельно представлены знак и величина числа: минус кодиру-

* В презентации Н. Вирт использовал выделение цветом, за неимением такой возможности, на этом и последующих рисунках мы выделяем соответствующий текст подчеркиванием: точками – синий цвет, двойной линией – красный. Оригинальное выделение вы можете увидеть в презентации на диске к журналу. *(Прим. ред.)*

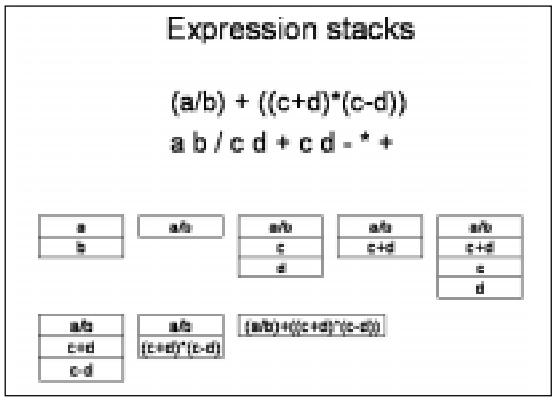
ется как 1, а плюс как 0. Во втором столбике отрицательные числа закодированы двоичными цифрами, дополняющими цифры исходного числа до 1, а в третьем столбце – до 2 (с учетом переноса разряда). Я думаю, что в настоящее время молодежь больше не знает способа с дополнением до 1. Все современные компьютеры используют систему дополнения до 2. Почему она оказалась предпочтительной? Она проста, легка для технического воплощения, понятна для чтения (при сложении противоположных чисел получается ноль, если игнорировать последний перенос разряда – прим. ред.). Кстати, еще один недостаток записи с дополнением до 1 в том, что получаются два различных обозначения для одного и того же нуля.



Числа с плавающей точкой – следующая тема. Здесь я только упомяну, что наибольшая плотность хранения чисел достигается при B = 3, однако троичная система так и не вошла в практику конструирования компьютеров.



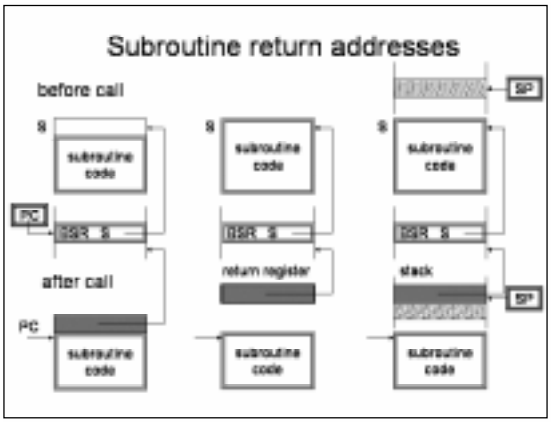
Иногда простые вещи оказываются очень существенными, хотя иной об этом и не подумает. Посмотрим, например, как нарушаются привычные законы вследствие особенностей машинной арифметики. Пусть наш компьютер имеет для хранения цифр числа три разряда. Пример показывает парадоксальный результат умножения: произведение меньших чисел оказалось больше произведения больших. Это произошло из-за отсечения цифр младших разрядов (в первом случае цифра 9 отсеклась, во втором цифра 5 сохранилась).



Следующая идея из моего списка снова из области архитектуры компьютера, это возможность использования стека для работы с арифметическими выражениями. Конечно, вы знаете, что арифметические выражения «по научному» описываются бинарными деревьями, которые могут быть преобразованы в так называемую постфиксную форму (бесскобочную запись – *прим. ред.*). При использовании такой записи, можно видеть, что стек – идеальный механизм для вычислений значений выражения: при появлении во входной строке переменной, ее значение загружается в стек, а при появлении знака арифметической операции из стека извлекаются числа, над ними выполняется операция, а результат помещается в стек.

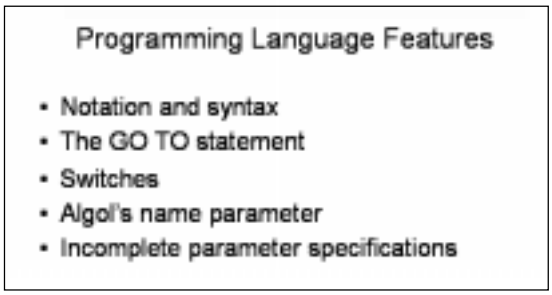
Однако до сих пор не создано стековых компьютеров, которые очень эффективно использовали бы регистры компьютера, а идея стека целиком остается в области программного обеспечения.

И снова из области архитектуры компьютера. Поговорим об изобретении подпрограммы, или, говоря техническим язы-

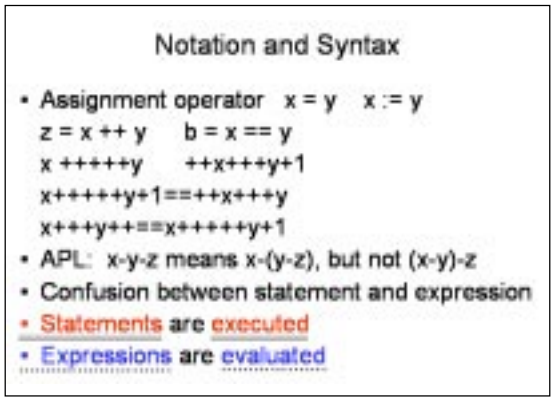


ком, об изобретении конструкции сохранения текущего адреса и возврата к нему, когда подпрограмма завершается.

В некоторых ранних компьютерах использовались технические средства хранения структуры вызова, которые позволяли восстанавливать состояние вычислительного процесса до вызова. Тогда этим гордились, теперь же видно, что это плохая идея, так как восстановление старой информации препятствует организации рекурсии. Можно вспомнить время, когда мнения разделились, быть ли рекурсивному вызову, допустить ли его или нет в языках программирования. Теперь это не вопрос. Рекурсия в современных языках программирования в порядке вещей, для ее организации, вместо регистра, хранящего возвращаемый адрес, используется стек вызовов процедур.



Теперь вернемся к нашей теме, к главе о хороших идеях. Я подготовил пять пунктов, о которых хотел бы упомянуть. Это система обозначений и синтаксис (я не согласен с тем, что форма и грамматика языка должны быть проще). Затем я хочу остановиться на операторе **GO TO** и переключателях. Затем обсудить параметры имен в языке ALGOL и неполные спецификаций параметров.



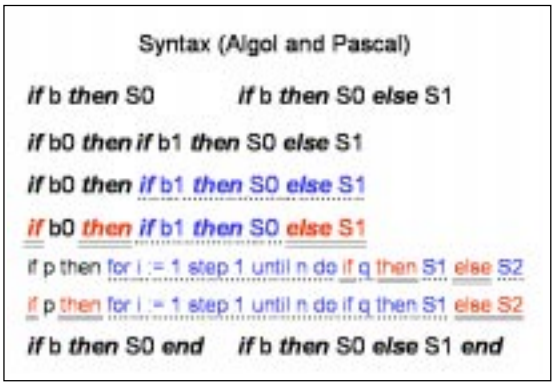
Первый пункт посвящен системе записи и синтаксису. Оператор присваивания в некоторых языках обозначается знаком «равно», что приводит к путанице между утверждениями и выражениями. В подобных ужасных вещах преуспели так называемые «современные языки программирования» типа C/C++ и JAVA. В них появились такие странные обозначения как **Z=X++Y**. Разве это нормально? Невозможно прочитать выражения подобные **b=X= =Y**. А затем, если вы должны решить правильно это или не правильно, попробуйте понять смысл выражения **X+++++Y+1= =++X+++Y**! (смех в зале – *прим. ред.*).

Странные вещи могут встретиться!

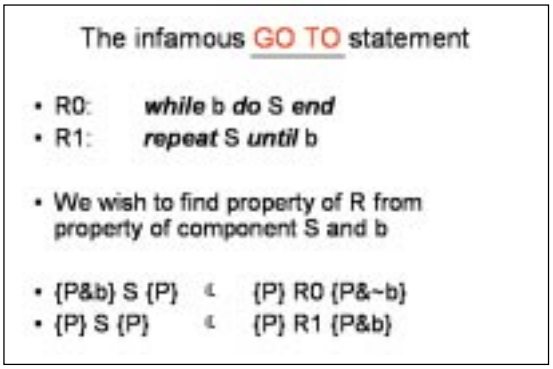
А вот пример из языка APL, где **X-Y-Z** в действительности означает **X-(Y-Z)**, но не **(X-Y)-Z**!

Разве это не высокомерно со стороны создателей таких языков программирования претендовать на создание математической системы обозначения? В центре всего этого базовая путаница между оператором и выражением. И если вы используете естественный язык, вы должны быть уверены, что все используемые операторы выполняются, а выражения вычисляются так, как положено.

Теперь обсудим синтаксис языков программирования. Возьмем два оператора: **if b then s0** и **if b then s0 else s1**. Следуя правилам грамматики из них можно построить конструкции, внешне выглядящие одинаково, как **if b0 then if b1 then s0 else s1**, но имеющие различный смысл: не ясно будет ли это третья



или четвертая строчка (см. рисунок). А одно из основных требований языка программирования – однозначность интерпретаций предложений языка. Эта проблема идет еще от языка ALGOL. Вы можете совершенно справедливо спросить, почему я не откорректировал это еще в ALGOL'е? Я и сейчас стою за то, чтобы это исправить, но оставил это как в ALGOL'е, чтобы сохранить преемственность языков. Как сказал однажды Дейкстра: «Люди не могут различить общепринятое и удобное». И только недавно я нашел другое решение. В последних строчках вы видите примеры, которые делают более очевидной мою мысль.

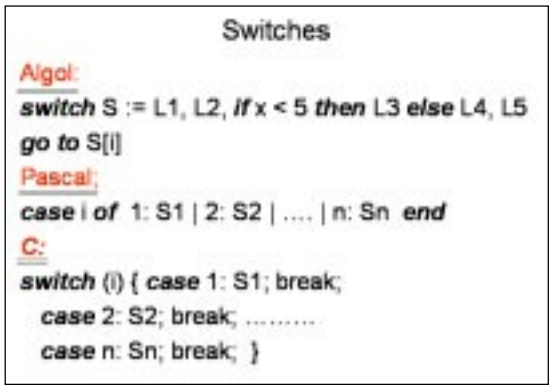


Бесславное **GO TO**.

Вы знаете, что в Фортране вы должны, а в Паскале можете, записывать цикл с помощью оператора скачка **GO TO**.

В Паскале, а позднее в Modula 2 мы выделили две формы циклов, вот они: **R1** и **R0** (см. рисуок).

Теперь разберемся, почему мы хотим делать вычисления без скачков. Для составления качественных программ, нужно иметь инструменты доказательства корректности программ: хотелось бы выводить свойства составляющих **R1** и **R0** только из свойств компонентов **S** и **b** Это было сделано Дейкстрой в конце 1960 годов в работе, посвященной теории верификации программ. (Использование структурного программирования привело к созданию понятия «инвариант цикла» и технологии доказательства корректности программы на основе математической логики).



Далее переключатели. Здесь я много не скажу. Замечу лишь, что если вы строите идею *A* на плохой идее *B*, то *A* может получиться еще хуже, чем *B*. Так в АЛГОЛе, а затем в Паскале была конструкция **CASE**, которая (внешне – *прим. ред.*) не содержала конструкции **GO TO**, но для доказательства корректности работы этой конструкции вам нужно было найти свойство, состоящее в том, что **1:S1|2:S2|... n: Sn**, что нелегко (*прим. ред.*).

Создатель языка Си знал об этих проблемах и считает, что решил их. На самом деле, произошло обратное. То, что хотели сделать создатели АЛГОЛА и Паскаля – обеспечить последовательный ход и анализ программ, перечеркнуто конструкцией переключателей в языке С, которая ломает (дословно, BREAK – прерывает – *прим. ред.*) непрерывный ход программы, что, несомненно, плохой стиль языка.

Теперь я подхожу к обсуждению параметров имен в Алголе. Это пример структуры, которая замышлялась как великая, но в реализации оказалась жалкой, использование обычных параметров стало менее эффективным и использовалась она чрез-

Algol's name parameter

```
real procedure square(x); real x; square := x*x
square(a)  literally means  a*a
square(sin(a)*cos(a))    stands for
  sin(a)*cos(a)*sin(a)*cos(a)

real procedure square(x);
  value x; real x; square := x*x
stands for
  begin real x'; x' := x; square := x'*x' end
```

вычайно редко. (А Паскале это было компенсировано использованием другого вида параметров – см. рисунок).

Позвольте мне объяснить, что идея параметров имен в комитете АЛГОЛа дебатировалась много лет. Основная идея была копирование, однако разработчики АЛГОЛа реализовали ее буквально. Рассмотрим идею на примере процедуры возведения в квадрат.

При вызове процедуры **`square(a)`** происходит замещение формального параметра **`x`** параметром **`a`**, и получается выражение **`a*a`**. Однако при другом вызове **`square(sin(a)*cos(a))`**, мы получим выражение **`sin(a)*cos(a)*sin(a)*cos(a)`**. Очевидно, этого результата нам получать не хотелось.

Algol's Jensen device

```
real procedure sum(k, x, n);
begin real s;
  for k := 1 step 1 until n do s := s+x;
  sum := s
end

a1 + a2 + ... + a100      sum(i, a[i], 100)
a × b                     sum(i, a[i]*b[i], 100)
```

Перед тем, как я продолжу, я прочту пример умного использования параметров имен. Впоследствии этот прием был назван ALGOL's Jensen device (механизм Дженсена в АЛГОЛе).

Для процедуры **`SUM`** (см. рисунок) **`s`** и **`x`** – параметры имен. Теперь, если вы хотите вычислить сумму $a_1+a_2+...+a_{100}$ вы можете написать тоже **`sum(i,a[i],100)`**. Элегантно! Теперь пусть нам надо вычислить скалярное произведение векторов **`a`** и **`b`** в координатах. Запишите элементарно **`sum(i,a[i],100)`** и получите ответ!

Incomplete parameter specification

```
real procedure f(x, y); f := (x-y)/(x+y)
u := f(2.7, 3.14); v := f(f(u, v), f(i, j))

procedure P(b, q); Boolean b; procedure q;
begin integer x;
  procedure Q; x := x + 1;
  if b then P(¬b, Q) else q;
  write(x)
end                          P(true, P)
```

А вот два примера, из которых видно, какие проблемы могут возникнуть в использовании параметров имен. Попробуйте, разобрать эти примеры и сформулировать проблемы самостоятельно (*прим. ред.*).

Dijkstra's "display"

```
procedure A(proc h);
begin integer x;
  procedure B;
  begin integer y;
    procedure C;
    begin integer z; h
    end ;
    C
  end ;
  B
end

procedure P;
begin integer i;
  procedure Q;
  begin integer j;
    procedure R;
    begin integer k;
    end ;
    A(R)
  end ;
  Q
end
```

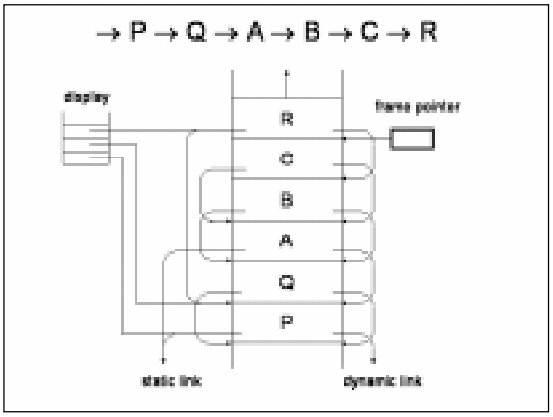
Далее речь пойдет о так называемом дисплее, изобретенном Дейкстрой. Это не дисплей подобный экрану, а программный механизм, изобретенный примерно в 1965 (тем более что сейчас уже и нет дисплеев, на смену им пришли мониторы).

Проблема появляется в связи с тем, что программа состоит из блоков, в которых есть локальные переменные. Рассмотрим пример программы, представленной на рисунке.

У нас есть процедура **`A`**, внутри нее процедура **`B`**, внутри **`B`** – процедура **`C`**. А справа представлена процедура **`P`**, которая

вызывает процедуру **A**. Процедура **P** имеет внутри себя процедуру **Q**, внутри которой находится процедура **R**, которая используется как параметр при вызове процедуры **A**, описанной слева.

А эффект в том, что когда вы вызываете **P**, то **P** вызывает **Q** и **Q** вызывает A с **R** как параметром, после чего **A** вызывает **B**, **B** вызывает **C**, а **C** вызывает **h**, где в качестве **h** используется процедура **R**. Таким образом, эффект в том, что нужно переключить контекст использования переменной. Это поясняет следующий рисунок.



Когда вы вызываете **C**, то получаете конфликт между **P** и **A**, а когда вызываете **R**, получаете конфликт между **R** и **C**. Теперь вы ожидаете, что сможете адресовать локальную переменную с помощью цепи, что показано на рисунке справа. Такая цепь создается в процессе выполнения программы, чтобы возвращать вызовы. Но это ведет к неправильному доступу. Тогда у вас есть возможность использовать вторую цепочку, так называемую статическую связь, которая помогает решить проблему, поскольку появляется средство спускаться по цепочке, но это, конечно, влияет на эффективность.

В этом случае локальная переменная получает несколько мест в памяти. Для разрешения этой трудности Дейкстра предложил так называемый регистрирующий дисплей, который дополняет направление регистраций, что всегда будет указывать на правильные блоки. Тогда, один раз вызванное действие будет только единожды доступно. Однако когда вы вызываете процедуру, нужно устанавливать новый входной дисплей, что делает ситуацию еще хуже, так как иногда приходится заменить всю последовательность входов. И не только при вызове, но иногда и при возврате. Это тот случай, когда происходит возврат от **R** к **C**.

Эта коллекция может удивить, но я бы хотел закончить тем, что побудить вас к тому, чтобы смотреть на новые идеи с сомнением. Действительно ли эта идея хорошая или нам сначала надо хорошенько над этим подумать? Наверное, вы ждете от меня, чтобы я в заключение сказал что-нибудь мудрое. Некоторые из хороших идей стали неверным в новом окружении. Не такими хорошими. Иногда из-за технологических изменений, иногда из-за изменения целей и традиций. То, с чем часто работают, оптимизируется. Может быть, поэтому мы придаем слишком много внимания эффективности вместо элегантности, простоты и прямоты. Вопрос, который я хотел бы задать – которая из нынешних хороших идей станет в будущем плохой? А может они уже плохие? Такие примеры есть, и одна из ошибок, что, на мой взгляд, в том, что мы используем старые стандарты, не переоценивая их. Особенно мы должны быть осторожны в отношении языков программирования. И в настоящее время множество языков, которые не оправданны, если брать в расчет количество людей их использующих. А что если подумать о так называемых современных языках, особенно C++. И мне не стыдно честно заявить, что это катастрофа. Это главное препятствие в развитии языков. Область программирования, инжиниринг. Люди тренируются в изучении и деталях одного конкретного языка. То,

что через 10 или 20 лет обернется плохой стороной. То есть обо всем этом можно в дальнейшем будет забыть. Вот почему я думаю, что в обучении какому-либо языку не следует гнаться за наиболее популярными и новыми особенностями, но всегда нужно концентрироваться на главном, ориентироваться на то, что останется значимым даже через 5–10 лет. Надо доказать основную концепцию, а затем вы сможете понять как эта концепция представляется в формальной конструкции записи языка. Иначе вы этого не поймете. Пожалуйста, не делайте ошибок. Не дайте себя запутать! Я не говорю, что все плохо. Я не говорю, что все, что делается в инженерии программного обеспечения, жалко. Как всегда говорилось с 1968, когда случился так называемый кризис программного обеспечения. В этом кризисе разработка программного обеспечения понесла чудовищные потери. Я хочу сказать, что сегодня мы имеем настолько переусложненные системы, что остается загадкой, как они могут работать. И здесь мы должны признать, что специалисты создающие их – это умелые и преданные своему делу люди. Вопрос не в том насколько они хороши, а в том, насколько долго мы еще сможем их делать. Я думаю сегодня, особенно в программировании, разработчики всегда находятся под давлением. У них конечный продукт должен быть закончен задолго до начала. А когда ты работаешь под давлением, очень сложно сохранить преданность делу и осторожность. При создании языков мы должны не только оценивать, хорошо ли выполнена работа или нет, но следует проводить сравнение с другими языками. Если это трудно, то на это потребуется много времени. Хотя это и не очевидно, но преимущество такого подхода в том, что он дает путь для научного подхода к таким вопросам. Я имею в виду громадные проекты, которые занимают сотни программистских часов и усугубляются тем, что выбор конкретного языка программирования делается людьми, которые не представляют результата работы, не умеют программировать, а просто покупают проекты. Поэтому, чуть больше тщательности, чуть больше научного подхода будут вашим вкладом в наше общее дело. Конечно, если мы говорим о лучшей идее, почему не отбросить все эти раздумья, почему не отбросить такие тесты, я имею в виду, что это технический бизнес. Делайте это.

Закончить я бы хотел словами благодарности за внимание, и нашу с вами встречу. Для меня это большая честь, огромное удовольствие, великий день. И вы помогли этому событию состояться. Действительно здорово, когда кто-то где-то припомнит, что чья-то преданная работа в течение нескольких десятилетий где-то оценена.

Большое спасибо.