Термин «алгоритм» используется не в общепринятом смысле, скорее в смысле алгоритма в человеческой деятельности как последовательности действий, что ближе к понятию потока работ (workflow). В русской версии понятия алгоритма используемая трактовка упоминается, в английской – нет. В связи с этим в тексте полезно вставить оговорку по поводу используемого смысла этого понятия и раскрыть это подробнее, чтобы последующие разделы были понятнее.

Почему алгоритмы трудны для понимания? Потому, что существующий способ записи алгоритмов (принятый во всем мире) выбран неудачно. Он устарел и превратился в досадное препятствие.
Устарел. Перебор. Вы верите, что языки С, С++, Паскаль и др. – неудачны и что Дракон исправит ситуацию?

Стр. 7 Существует несколько способов для записи алгоритмов.

Какие - перечислить

Стр. 10 АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

Дать какое-то определение, например, я определяю программирование как кодирование алгоритма в конструкциях языка программирования. Но я использую общепринятое понятие алгоритма.

Стр. 11-12 100%-е автоматическое доказательство – нет тут доказательства.
Безошибочное проектирование графики дракон-схем – важный шаг вперед, повышающий производительность труда при практической работе.
Слишком громкое заявление. Здесь нет ничего сверхестественного. Скажите просто: графический редактор гарантирует правильность графического синтаксиса. А Вы сможете указать графический редактор, допускающий ошибки графического синтаксиса?

Стр. 18 Следующие четыре – «действие».

Здесь какая-то ошибка

Стр. 32 **АНАЛОГИ ДРАКОН-СХЕМ**
А почему дракон-схемы лучше аналогов. Необходимо сравнение

Стр. 37 Они обеспечивают быстроту и легкость понимания по принципу: «Посмотрел – и сразу понял!».
Благодаря использованию формальных и неформальных приемов дракон-схемы дают возможность изобразить любой, сколь угодно сложный алгоритм в наглядной и доходчивой форме.
Это не получится для сложных вычислительных алгоритмов

Стр. 48 **§5. ТРИ «ЦАРСКИХ» ВОПРОСА**
Сталкиваясь с новой незнакомой задачей, мы желаем получить ответ на три царских (наиболее важных) вопроса:
1. Как называется задача?
2. Из скольких частей она состоит?
3. Как называется каждая часть?

Далеко не всегда задача декомпозируется таким образом

Стр. 54 §13. ЧТО ТАКОЕ ПОНЯТНОСТЬ АЛГОРИТМА?
Многие алгоритмисты

Большинству людей будет непонятно, кто такие алгоритмисты

Стр. 76. Появление метки М1 внутри ветки (не в начале) и оператора перехода на нее не очень хорошо. Возможно, лучшее решение – образование дополнительной ветки, начинающейся с М1.

Стр. 84 **§9. ТЕОРЕМА РОКИРОВКИ**

Слово теорема уместно лишь для строгой формальной формулировки, допускающей формальное доказательство, т.е. предварительно нужна формализация всего графического представления.

Стр. 93 Может быть вместо слова «теорема» поставить «утверждение».

Стр. 105. Таким образом, язык ДРАКОН позволяет устранить любое пересечение соединительных линий, используя строгие математические методы.
Не вполне правильно говорить здесь про строгие математические методы.

Стр. 114. Существующие циклы, используемые во всем мире, имеют серьезный недостаток. Они накладывают на творческую мысль алгоритмиста неоправданные ограничения. Графика позволяет снять многие из этих ограничений.
Проблемы с циклами гораздо серьезнее. Циклы в вычислительных программах (программах-функциях) и в автоматных программах (а также бытовых) – это циклы разной природы. Все циклы в вычислительных алгоритмах (кроме **for** j = 1..n **do**) – плохие; причем они плохие и в тексте, и в графике. Циклы автоматной программы предпочтительней представлять конструкцией силуэт.

Начало второй части замечаний

Стр. 125. Правило: один выход основной, а остальные досрочные – действует не всегда. Из цикла может быть несколько выходов, часто они являются однотипными и равноправными – между ними нет принципиальной разницы. Деление на основные и не основные – это наше деление, не имеющее ничего общего с алгоритмикой. Наконец, вместо «досрочные выходы» возможно лучше использовать термин «дополнительные выходы».

Стр. 132. В веточных циклах искусственность деления на основной и досрочный становится более заметной.

Глава 10. Есть сомнение в ценности представления примеров разных комбинаций циклов в составе двойных и тройных циклов. Есть ли польза в примерах. Из общих соображений ясно, что разные комбинации возможны. Далее, циклы бытовых алгоритмов выглядят простыми. А циклы, даже одинарные в реальных программах – сложные, не говоря о двойных.

Стр. 153. «да» и «нет» можно использовать вместо «истина» и «ложь». Но это не есть устранение ненужной сложности. Сложность останется той же. Можно говорить об эргономике, о предпочтении одного или другого варианта в определенных приложениях.

Однако использование «да» и «нет» вместо «истина» и «ложь» привело к следующей особенности: логическое выражение де факто перестало быть таковым. Теперь, чтобы определить значение логического выражения Q мы пишем Q = да, хотя ранее писали просто Q.

Стр.156. Рис.106. Алгоритмы справа и слева не эквивалентны. Пример:

```
if (a != 0 && b / a > 3)      { /* какие-то действия */ };
```

Исполнение справа приведет к прерыванию, тогда как слева прерывания не будет. Это эффект так называемого Маккаритиевского «и». В изложении необходимо сделать соответствующую оговорку.

Аналогичное замечание о неэквивалентности вариантов на Рис.111

Стр. 157. Опыт показывает, что большинство людей выбирают визуальный способ как более легкий.
Какие группы людей Вы опрашивали?

Специалисты по математике, особенно математической логике, наверняка предпочтут левый вариант, причем в привычной для них форме: Q & R & S

Стр. 158. Когда мои студенты пишут в формулах или в программе

Q & R & S = true

я заставляю их менять на

Q & R & S

По тем же соображениям не следует использовать запись вида

Q & R & S = да

Аналогичное замечание относится к пояснению справа на Рис.108

Стр.190. Приведенные восьмисимвольные имена не самые лучшие. Функция имен – идентификация и различение. Поэтому достаточно одного ключевого слова. По моему мнению, лучше использовать следующий набор имен: ехать, зеленый, желтый, красный, перекресток, помехи. Мнемонические однобуквенные имена: Е, З, Ж, К, Н, П – на порядок лучше произвольных.

Стр.193. Действительно, TRUE и FALSE – избыточны в тексте, но тогда то же самое можно сказать о значениях «да» и «нет».

Стр.194. Рис 135. Надо было бы хотя бы минимально структурировать формулу, чтобы, например, каждый операнд дизъюнкции начинался с новой строки.

Стр.196. Если идти дальше в устранении визуальных помех, то «Формирование признака» - тоже лишнее. На рис. 137 текст «Можно.ухать.через.перекресток» следует переставить в заголовок вместо текста «Формирование признака». На рис. 136 убрать вставку «Формирование признака», а вставку «Можно.ухать.через.перекресток» поставить внутрь иконки «вопрос».

Стр. 200. В формуле (11) пропущен знак отрицания.

Стр. 200. Желательно, чтобы конкретные идентификаторы в зависимости от сложности понятия имели длину не менее 25 и не более 32 символов.

Предположение об оптимальности 32-символьных идентификаторов согласуется с анализом истории развития алгоритмических языков, который обнаруживает отчетливую тенденцию: Описка: который -→ которая

- переход от абстрактных кодов и имен к 6- или 8-символьным мнемоническим именам;
- затем – переход к 32-символьным смысловым идентификаторам.

Вместе с тем многие специалисты, следуя устоявшимся привычкам, «застряли» на этапе 8-символьных имен.

Конечно, вводить ограничение в 8 символов – неправильно. Но требование от 25 до 32 символов в идентификаторе – другая крайность. Более взвешенным решением является возможность использования таких имен для определенных приложений. Для большинства случаев реализация таких требований не приведет к хорошим результатам.

В программах вычислительной математики – идентификаторы в основном от однобуквенных до трехбуквенных. И на это есть серьезные причины – имена переменных используются в нескольких математических формулах. Эти формулы станут нечитабельными для длинных имен. Программа – тоже. Что будет, если вместо НОД(a, b) ,будем писать:

Наибольший_общий_делитель(первый_аргумент_НОД, второй_аргумент_НОД) ??

Поскольку Дракон – всего лишь графическая оболочка, а подавляющее число примеров – бытовые алгоритмы, упускается важнейший аспект программ – наличие разнообразных связей между переменными – информационных, логических, набора условий, где участвуют переменные. Во многих вычислительных программах адекватное представление этих связей – важнее, чем именование переменных. В математических текстах используются преимущественно однобуквенные – двухбуквенные имена из очевидных соображений, чтобы лучше изобразить математические зависимости.

Стр. 201. Заменить заголовок «Теорема» на «Утверждение». Теорема, конечно, может формулироваться словесно. Тем не менее, она должна допускать экспликацию в виде строгой логической формулы, допускающей формальное доказательство ее истинности. Здесь предварительно необходимо формализовать Дракон-схему и процедуру, фрагмента Дракон-схемы и замены фрагмента одной иконой.

Стр. 240. Язык Z – это известный язык спецификаций. Лучше использовать другое имя вместо Z.

Стр. 242. Алгоритм можно назвать *красивым* (эргономичным) в том случае, если процесс зрительного восприятия, понимания и постижения алгоритма протекает с максимальной скоростью, наименьшими усилиями и максимальным эстетическим наслаждением.
Такое определение красоты вызовет недоумение не только у специалистов из мира искусства. Максимальное эстетическое наслаждение – это следствие воздействия красоты, но никак не может служить ее определением. Остальное к красоте не имеет отношения. Наверное, скорость восприятия и объем усилий правильно было бы рассматривать независимо от понятия красоты.

Стр. 243.
ДРАКОН – язык красивых (эргономичных) зрительных образов.
Да, в каких-то местах можно говорить о красоте алгоритмов, и это будет являться положительным бонусом. Но утверждение о красоте всех Дракон-схем не очевидно и выглядит спекулятивным. Кроме того, «красивый» выглядит в этом разделе как эквивалент «эргономичный», а это, наверное, не так. Отметим, что эргономичность Дракон-схем не вызывает сомнений, а вот с красотой – сложнее.

Стр. 253. Дракон-схемы подчиняются: строгим математическим правилам
Вряд ли правильно здесь писать, что правила – математические. Выше это ничем не подтверждается.

Стр. 265. Наверное, преждевременно утверждать, что сочетание языка Дракон с языками высокого уровня – это этап в развитии языков программирования. Пока что это лишь эпизод. Интерес в мире здесь пока не виден. Каков опыт применения гибридных языков?

Стр. 266. Как показывают первые опыты подобной работы, переход от языков высокого уровня к гибридным языкам программирования свидетельствует о заметном повышении производительности труда программистов.

Вот здесь были бы уместны ссылки и хоть какие-то подтверждения в повышении производительности.

Стр. 275. Вот здесь бы сослаться на книги, которые Вы мне показывали.

Стр.277. Есть понятие workflow – поток работ. Есть системы – графические языки, поддерживающие потоки работ на метауровне. Надо бы сравнить с Драконом. Имеется книга:

Shukla D., Schmidt B. Essential Windows Workflow Foundation / Addison Wesley Professional. – 2006.

Ее можно выкачать в Интернете

Стр.302. Более того, чем сложнее проблема, тем больше выигрыш от использования языка ДРАКОН

Кроме вычислительных программ, которые я называю программами-функциями

Стр.324. Вы получали какую-либо реакцию от ученых биологов?

Стр.351. Ваше описание алгоритма Евклида выглядит совершенно тривиально. А почему именно такой алгоритм? Является ли он правильным и как в этом убедиться? А если нужно будет доказать правильность алгоритма формально, например, в системе автоматического доказательства PVS, то нужно будет также формализовать постановку задачи.

Приведу соответствующий фрагмент из моего учебника по предикатного программирования 2008г.

Пример 5.2. Программа D(a, b**:** c) вычисления наибольшего общего делителя (НОД) положительных a и b.

Определим свойство «x является *делителем* a» следующим предикатом:

$$x \text{ – делитель } a \cong \text{divisor}(x, a) \cong \exists z \geq 0.\ x * z = a\ .$$

Предикат:

$$\text{divisor2}(a, b, c) \cong \text{divisor}(c, a)\ \&\ \text{divisor}(c, b)$$

определяет c в качестве *общего делителя* значений a и b. Свойство *наибольшего общего делителя* определяется предикатом:

$$\text{НОД}(a, b, c) \cong \text{divisor2}(a, b, c)\ \&\ \forall x.\ (\text{divisor2}(a, b, x) \Rightarrow x \leq c)\ .$$

Приведенная далее программа вычисления наибольшего общего делителя базируется на следующих известных базисных свойствах НОД:

$$a = b\ \Rightarrow \text{НОД}(a, b, a) \qquad (5.21)$$

$$a < b\ \&\ \text{НОД}(a, b, c) \Rightarrow\ \text{НОД}(a, b - a, c) \qquad (5.22)$$

$$\text{НОД}(a, b, c) \equiv \text{НОД}(b, a, c) \qquad (5.23)$$

Из базисных свойств НОД легко доказать истинность следующих свойств:

$$\text{НОД}(a, b, c)\ \&\ a = b\ \Rightarrow c = a \qquad (5.24)$$

$$\text{НОД}(a, b, c)\ \&\ a < b\ \Rightarrow\ \text{НОД}(a, b - a, c) \qquad (5.25)$$

НОД(a, b, c) & a > b ⇒ НОД(a - b, b, c) (5.26)

Свойства (5.24-5.26) определяют *логику решения* задачи НОД. Предикатная программа вычисления НОД непосредственно переписывается из логики решения (5.24-5.26).

```
D(nat a, b:  nat c) ≡ pre a ≥ 1 & b ≥ 1
{       if (a = b) c = a
        else   if (a < b) D(a, b – a: c)
               else D(a – b, b: c)
} post НОД(a, b, c);
```

Предикатная программа получается непосредственно по логике решения. Связь императивной программы с логикой решения сложнее. Императивная программа вычисления НОД получается из предикатной трансформацией замены хвостовой рекурсии циклом:

```
D(nat a, b: nat c) {
M:      if (a = b) c = a
               else   if (a < b) {|a, b| = |a, b - a|; goto M}
               else  {|a, b| = |a - b, b|; goto M}
}
```

Раскроем групповые операторы присваивания, а также заменим фрагмент с операторами перехода на цикл **for**. Получим:

```
D(nat a, b: nat c) {
        for ( ; ; ) {
               if (a = b) {c = a; break; }
               if (a < b) b = b – a
               else a = a – b
        }
}
```

Вывод: описывать математические алгоритмы без математики, лежащей в их основе, неправильно.

Стр.434. В этой главе много чего не так. Визуальное логическое исчисление надо строить по строгим математическим правилам, а этого нет. На самом деле надо строить математическую формализацию Дракон-схем в традиционном математическом стиле. Это возможно на базе теории графов. Дракон-схема – ориентированный граф определенной структуры. 37 тезисов предыдущей главы 33 следует формализовать как операции над графами. Тогда здесь появятся настоящие теоремы со строгим математическим доказательством в обычном смысле. Как следствие, вся терминология станет на свои места.

Думаю, что и не Ваше это дело строить формализацию. Виктор Касьянов – общепризнанный специалист по теории графов. А задача формализации Дракон-схем – замечательный полигон для студенческих работ. Думаю, он не откажется и года через три эту дыру закроет. Разумеется, на это нужно Ваше согласие.

Стр.434. *Вывод.* Метод Ашкрофта-Манны можно рассматривать как математическое обоснование основной алгоритмической структуры языка ДРАКОН – структуры «силуэт».

Я Вам писал ранее, что обоснование на базе метод Ашкофта-Манна – сложное, поскольку использует эквивалентные преобразования полученных Дракон-схем. Возможно более простое обоснование – описать алгоритм преобразования произвольной блок-схемы в Дракон-схему. Он несложный. Оператор блок-схемы назовем *оператором начала цикла*, если он имеет более одной входной дуги. Конечно, одна из дуг должна быть возвратной. Но для алгоритма это не обязательно. Связный фрагмент блок-схемы, начинающийся с корневого оператора или с оператора начала цикла и заканчивающийся оператором начала цикла или конечным оператором назовем *веткой*. Далее последовательным обходом блок-схемы (как графа) блок-схема разбирается на ветки.

Стр.457. Моя позиция по отношению к концепции структурного программирования не помещается в Вашу классификацию. Возможно, кто-то еще придерживается подобной точки зрения, но я таких не знаю. Я считаю эту концепцию ошибочной и вредной. Оператор **goto** совершенно безобидный оператор по сравнению с циклом типа **while** и указателями, которые представляют серьезную беду для программирования. Положительным было лишь постановка задачи построения хороших программ и толчок к исследованиям в этом направлении. Структурное программирование было внедрено как религия и до сих работает как религия. Неудавшаяся попытка устранения оператора **goto** из языков программирования, вакханалия структурного программирования долгие годы – это процесс в ложном направлении.

Стр. 487. 3. Традиционные алгоритмические языки создавались без учета требований когнитивной эргономики.
Слишком сильное утверждение. Некоторые эргономические детали все же присутствуют

Стр. 489. **В ЧЕМ ОШИБСЯ ЕРШОВ?**
Он не ошибался.
«Программирование – вторая грамотность» появился как лозунг, чтобы привлечь людей осваивать программирование. Первый доклад на эту тему Ершов сделал на нашем институтском семинаре. В то время число программистов было в десятки раз меньше чем сейчас. Шел разговор о том, чтобы учить программированию детей. Так возникла школьная информатика. Но никогда не говорилось о тотальной компьютерной грамотности, т.е. что каждый гражданин СССР должен уметь программировать.

Что касается реплики Нарильяни, то надо аккуратно посмотреть, в каком контексте она была сделана. Нарильяни в то время работал в нашем институте и был учеником Ершова, и потому все прекрасно знал.

У меня тоже есть похожий лозунг: «умение доказывать теоремы на компьютере становится частью математического образования», но это не значит, что уметь доказывать на компьютере должны все математики.

Стр. 490. Многое из того, что Вы называете алгоритмами, на Западе назвали бы требованиями.

Стр. 493. А здесь уже с большей определенностью нужно говорить о требованиях, а не алгоритмах. Я примерно также мог бы сказать о тотальной безграмотности в нашей стране по отношению к инженерии требований.

Стр. 505. Язык ДРАКОН имеет две опоры. Первая – математика.

Возможно, лучше не делать такого заявления, потому что трудно будет это объяснить. Да и с математикой не все в порядке.