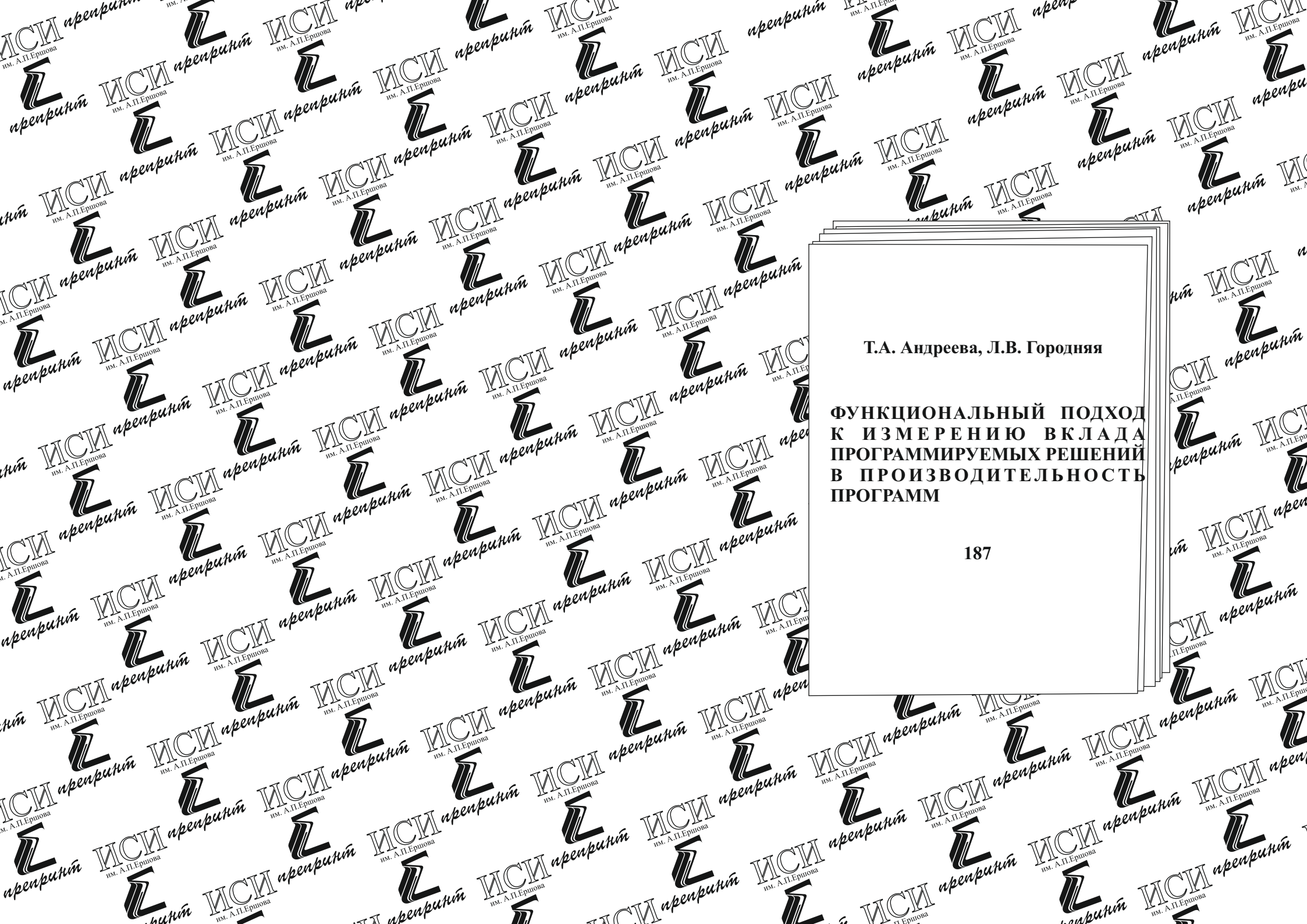
Т.А. Андреева, Л.В. Городняя
ФУНКЦИОНАЛЬНЫЙ ПОДХОД К ИЗМЕРЕНИЮ ВКЛАДА ПРОГРАММИРУЕМЫХ РЕШЕНИЙ В ПРОИЗВОДИТЕЛЬНОСТЬ ПРОГРАММ
187

Российская академия наук
Сибирское отделение
Институт систем информатики
им. А.П. Ершова

Т.А. Андреева, Л.В. Городняя

ФУНКЦИОНАЛЬНЫЙ ПОДХОД К ИЗМЕРЕНИЮ ВКЛАДА ПРОГРАММИРУЕМЫХ РЕШЕНИЙ В ПРОИЗВОДИТЕЛЬНОСТЬ ПРОГРАММ

Препринт
187

Новосибирск 2022

Рецензент — к.ф.-м.н. Д.С. Мигинский

Препринт посвящен экспериментам по вопросам оценки влияния программируемых решений на продуктивность программирования и производительность программ в процессе обучения программированию и улучшения программ на практике. При непосредственных измерениях производительности программ вместо оценки программируемых решений измеряется производительность комплекса из машины, компилятора и программы. Такие измерения слабо отражают вклад принятых программистом решений в производительность полученной программы. Рассматривается гипотеза, что функциональные модели могут быть полезны как метрическая шкала, позволяющая отделять особенности используемых языков и систем программирования от характеристик программ и запрограммированных решений. Одно из возможных решений — измерять производительность программ на основе нормализованных функциональных форм и тем самым устранять зависимость оценки от аппаратуры и системы программирования.

Siberian Branch of the Russian Academy of Sciences
A.P. Ershov Institute of Informatics Systems

T.A. Andreyeva, **L.V. Gorodnyaya**

FUNCTIONAL APPROACH TO MEASURING THE CONTRIBUTION OF PROGRAMMING SOLUTIONS INTO THE PERFORMANCE OF PROGRAMS

Preprint
187

Novosibirsk 2022

Reviewer: Dr. Miginsky

The preprint is dedicated to experiments in measuring the effect that programming solutions have on the productivity of programming and on program performance in educational programming and program improvement in practice. As a rule, while program effectiveness is measured, instead of measuring the productivity of programming solutions, the productivity of a complex consisting of a computer, compiler, and program is measured. These measurements do not fully reflect the contribution of programming solutions to the overall productivity of the program. A hypothesis is considered stating that functional models can provide a metric scale capable of separating the features of programming languages and systems from these of programs and programming solutions. A possible approach is to measure program productivity on the basis of normalized functional forms and thereby to reduce the dependence of the measurements on hardware and programming systems.

Т.А. Андреева, Л.В. Городняя Функциональный подход к измерению вклада программируемых решений в производительность программ. Новосибирск, 2022. 62 с.

Аннотация. Препринт посвящен вопросам, возникающим в связи с проблемой оценки влияния программируемых решений на продуктивность программирования и производительность программ в процессе обучения программированию и улучшения программных приложений. При рассмотрении этого вопроса принято во внимание, что жаргон практического программирования понятие «язык программирования» использует как «компилятор входного языка типовой системы программирования, функционирующей на базе определённой конфигурации оборудования». При непосредственных измерениях производительности программ вместо оценки программируемых решений измеряется производительность комплекса из машины, компилятора и программы. Такие измерения слабо отражают вклад принятых программистом решений в производительность полученной программы, что при обучении программированию препятствует мотивации улучшать работающие решения, а также неудобно для сравнения с подобными решениями в других программах.

Рассматривается гипотеза, что функциональные модели могут быть полезны как метрическая шкала, позволяющая отделять особенности используемых языков и систем программирования от характеристик программ и запрограммированных решений. Описаны результаты предварительного эксперимента, направленного на исследование зависимости производительности программ от выбора компилятора и отдельно от представления программируемых решений на определённом языке программирования. В результате намечается подход к созданию методики, позволяющей выяснять такие зависимости. При создании методики учтён многолетний опыт оценки учебных и олимпиадных работ по программированию, проявивший ряд не вполне очевидных аспектов проблемы. Одно из возможных решений — измерять производительность программ на основе нормализованных функциональных форм и тем самым устранять зависимость оценки от аппаратуры и системы программирования.

T.A. Andreyeva, L.V. Gorodnyaya Functional approach to measuring the contribution of programming solutions into the performance of programs. Novosibirsk, 2022. 62 p.

Annotation.The preprint is dedicated to experiments in measuring the effect programming solutions have on the productivity of programming and on program performance in educational programming and program improvement in practice. As a rule, program effectiveness measurements involve, instead of measuring the productivity of programming solutions, measuring the productivity of a complex consisting of a computer, compiler, and program. These measurements do not fully reflect the contribution of programming solutions to the overall productivity of the program. A hypothesis is considered stating that functional models can provide a metric scale capable of separating the features of programming languages and systems from these of programs and programming solutions. A possible approach is to measure program productivity on the basis of normalized functional forms and thereby to reduce the dependence of the measurements on hardware and programming systems.

Введение. Бум языкотворчества в области проблемно-ориентированных языков программирования (ЯП), знаменующий переход практики программирования от накопления опыта на уровне эффективных библиотечных модулей к накоплению удобных подъязыков, показывает важность измерения различий в определениях новых ЯП, улучшаемых версий программ и выборе инструментов для практических работ. В разных источниках упоминается от 20-ти до 70-ти парадигм программирования, список которых видоизменяется и расширяется в зависимости от актуальности тех или иных затруднительных проблем программирования, а также моды на особенности популярных ИТ.

Стремительно растущее число ЯП требует наличия методики, позволяющей оценивать пользу от их появления, обосновывать целесообразность трудозатрат на их изучение и улучшение практиками. Не исключено, что многие проблемы объясняются довольно сложной фрактальной природой самого феномена «знание» [1]. В этом направлении выполнено много работ, показавших, с одной стороны, важность проблемы измерения производительности программ, с другой, необходимость продолжения исследований и уточнения самого, весьма неоднозначного, понятия «производительность» с интеграцией знаний в этой трудно формализуемой сфере [2-8]. Развитие этого понятия, парадигм и технологий программирования отражает историю борьбы за повышение продуктивности программирования и преодоление трудно решаемых проблем [9, 10]. Известно, что перенос программируемых решений на уровень аппаратуры повышает производительность приложений примерно в 150 раз, а смена компилятора может повысить её в в 50 раз и более. Остаётся не вполне ясным, точнее, трудно измеримым, влияние собственно программируемых решений на производительность программ.

В последние годы в ИСИ СО РАН разработана парадигмальная методика анализа и сравнения ЯП [11]. На её основе формируется информационный стенд ПРИЗМА (**пр**ограммирование **изм**ерений), предназначенный для представления экспертных оценок возможностей ЯП и систематизации результатов анализа эксплуатационных характеристик программ, написанных на разных ЯП и отлаживаемых на конкретных системах программирования (СП) при определённой конфигурации оборудования [12]. Полноценное применение стенда подразумевает возможность экспертные оценки сопровождать численными характеристиками наблюдений за компьютерным экспериментом. Часть таких наблюдений фактически происходит при решении учебных и олимпиадных задач. Подобно тому, как техника верификации программ может оттачиваться на логических моделях, лишь отчасти удостоверяющих правильность программ, технику измерения производительности программ можно опробовать на функциональных моделях, частично характеризующих отдельные аспекты поведения программы, обусловленные программируемыми решениями [13]. Методы представления моделей и вывода логики программ подробно изучены при исследовании и применении систем верификации и автоматической проверки доказательств [14]. Такой подход позволяет проблемы общего измерения характеристик сложной системы сводить к измерению отдельных характеристик её менее сложных частей.

Учитывая отсутствие общепризнанной методики измерения производительности программ, опубликованы результаты разных экспериментов по анализу разных методов измерения отдельных характеристик программируемых решений, их зависимости от выбора ЯП и учёта особенностей реализации компиляторов. Например, в данной статье при анализе первых таких экспериментов использованы

результаты довольно известного бенчмарка «Какой язык быстрее всех»[1], заслужившего внимание программистов-практиков и студентов[2], результаты сравнения производительности языков Lisp, C/C++ и Java,[3] и привлечены механизмы бенчмарка «Курсы и задания»[4], нацеленного на поддержку обучения и самообучения практическому программированию. Такой бенчмарк обеспечивает возможность измерять отдельные характеристики производительности программ и программируемых решений в учебном процессе с помощью доступных онлайн-компиляторов для 76 ЯП.

Изложение начинается с анализа результатов игры «Какой язык быстрее всех» на примере языков C++, Go, Clisp, Java, Haskell, Pascal, Python и сравнения Lisp, C/C++ и Java, затем приведены результаты измерения производительности этих ЯП и языка Closure на бенчмарке «Курсы и задания» с использованием в качестве первого демонстрационного примера программы решения задачи вычисления чисел Фибоначчи, опубликованные на сайте «Энциклопедия языков программирования»[5].

В заключении сформулированы выводы относительно перспектив измерения вклада программируемых решений в продуктивность программирования и производительность программ при обучении программированию и дальнейшие планы по развитию стенда ПРИЗМА для экспериментов по созданию методики измерений.

---

[1] Which programming language is fastest? https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

[2] Некоторые студенты включаются в соревнование за создание более быстрых решений эталонных задач

[3] https://psachin.gitlab.io/lisp-java-notes.html

[4] Courses And Assignments https://www.jdoodle.com

[5] progopedia.ru

1. «Какой язык быстрее всех»

Организаторы эксперимента по проверке скорости ЯП сразу предупреждают: «Будем реалистами: большинство людей, как правило, ничуть не озабочены производительностью программ».[6] Чтобы узнать, какой ЯП, точнее, его компилятор, самый быстрый, они решили взять и измерить их в стиле спортивного чемпионата. Проект формировался с 2002 по 2021 год; в отчёте о его работе представлены данные об испытаниях языков C++, C, Rust, Fortran, Julia, C#, Ada, Chapel, Haskell, Go, Pascal, F#, OCaml, Java, Swift, Java, Script, Lisp, Dart, Racket, PHP, Erlang, Lua, Python, Smalltalk, Ruby, Perl. Измерялась скорость, память и объём программы.

Действительно, при мощности современных ИТ проблема производительности программ и программируемых решений интересует сравнительно узкий круг программистов и пользователей, причём критерии качества программ у этих двух сообществ достаточно различны. Разработчики компиляторов дорожат прежде полнотой реализации ЯП, а потом эффективностью их реализации. Специалисты в сфере особо важных и трудных задач прежде всего выбирают наиболее удобные, технологичные и достаточно продуктивные средства для решения задач в реальные сроки. Менее заметна заинтересованность в методики измерений для программистской практики улучшения ПО. Сложилась не вполне благоприятное положение дел, при котором заказчики улучшения ПО понимают работу по усовершенствованию интерфейсов, согласны оценивать её продуктивность, в то время как польза работ по оптимизации процессов обработки данных не столь очевидна.

---

[6] «It's important to be realistic: most people don't care about program performance most of the time»

Независимо от этого существует желание знать, что лучше всех: какие компиляторы, языки, программы, программируемые решения дают гарантированно хороший результат, позволяют выстроить надёжную практичную технологию производственного программирования в конкретных обстоятельствах. У организаторов эксперимента возникла «трезвая» мысль, точнее, гипотеза, что можно собрать коллекцию эталонных задач и написать программы их решения на разных языках. Рассматривая постановки эталонных задач и программы их решения, программист может интуитивно уловить сходство, аналогию или подобие со своей задачей и таком образом выбрать подходящий компилятор, спрогнозировать производительность будущей программы, а может, и продуктивность своей работы по её написанию. Обнаружение подобия означает существование неявной общей модели, которую современные методы представления знаний могут выражать в форме онтологий, довольно популярного направления, в котором уже имеются практичные результаты [15, 16]. Онтологии позволяют анализировать постановки задач на полноту, а программы их решения — на соответствие ожиданиям пользователя, т.е. на решение проблемы так называемой «юзабилити» [17].

Здесь стоит вспомнить разницу между подобием и эквивалентностью. Если для подобия достаточно уловить какую-либо общую модель, представляющую частичное сходство, то эквивалентность определяется более точно. Существуют разные уровни эквивалентности; для измерения производительности программ важна разница между подобием постановок задач и функциональной и/или семантической эквивалентностью программ их решения. Функциональная эквивалентность означает совпадение функций, т.е. одинаковость отображения аргументов функции в её результаты (ответ на вопрос ЧТО?), метод реализации отображения не имеет значения. Семантическая

эквивалентность означает совпадение способов реализации процесса отображения, выбор которых влияет на его эффективность (ответ на вопрос КАК?).

Например, функция может быть реализована как процедура, обладающая заметной алгоритмической сложностью, или представлена как таблица, что называют теперь мемоизацией, сложность которой сведена к выбору данного из памяти. Большинство оптимизирующих компиляторов выполняют преобразования семантики программ, сохраняющие их функциональную эквивалентность. В зависимости от состава таких преобразований полученный в результате компиляции код программы может различаться по производительности.

Организаторы эксперимента столкнулись с рядом проблем, решить которые удалось лишь частично.

Первая проблема — нет критериев выбора идеальных эталонов, поэтому проект «The Computer Language Benchmarks Game» — не более чем хорошая опорная точка для экспериментальной подготовки и дальнейшей проверки подходов к выбору эталонных задач, решения которых могут стать ориентиром при выборе или оценке компиляторов и создании метрической системы для измерения производительности программ.

Вторая проблема — как сравнивать по скорости столь разные ЯП, как C или C++ с C# или Python? Разработчика компилятора, который реализовывал языки программирования, обычно интересует, как «сравнить результаты измерения качества сгенерированного кода, когда два компилятора реализуют то, что составляет одну и ту же программу», подразумевая одни и те же запрограммированные решения, обладающие семантической эквивалентностью. Пока требования к решению эталонных задач ограничивали лишь метод на уровне функциональной эквивалентности, а не средства или конструкции на уровне

семантической эквивалентности.

Третья проблема — установление эквивалентности программ. При прогоне программ не вполне очевидна разница между функциональной и семантической эквивалентностью программ. Поэтому на первый случай требования к решению эталонных задач ограничены реализацией одного метода решения конкретной задачи на уровне функциональной эквивалентности, допускающей легко проверяемое соответствие между значениями аргументов и результатов. Примерно такой подход характерен и при проверке решений учебных и олимпиадных задач, хотя и преподавателей и организаторов олимпиад по программированию очень огорчает отсутствие проверки стиля программирования. Функционально эквивалентные программы могут не обладать семантической эквивалентностью, а потому их производительность может резко различаться.

Четвёртая проблема — не все важные особенности программ вообще допускают непосредственное измерение. Трудно измерить различия в подходах к дисциплине памяти, параллельному программированию, регулярным выражениям, арифметике с произвольной точностью, технике реализации и кодирования структур данных, хотя они являются неотъемлемой частью практического программирования.

Учитывая ещё больший комплекс проблем организаторы эксперимента выбрали нечто среднее между хаосом и жёсткостью — гибкостью и игрой, чтобы можно было независимо создавать программы на разных ЯП, а не просто механически конвертировать с одного ЯП на другой, устанавливая достаточное сходство в основных рабочих показателях и результатах тестирования. Они отдавали себе отчёт в том, что пока нет научных оснований для такого сравнения. Оно может появиться лишь после достаточного

объёма экспериментальных данных, а более точно, после интерпретации результатов анализа таких данных и появления на такой основе научных оснований для создания методики измерения производительности программ. Не ясен вопрос насколько измерение производительности программ отражает производительность компиляторов. Достижение высокой производительности программ может быть обременено заметным снижением производительности компилятора.

Организаторы эксперимента рассудили, что лучший выбор эталонов для измерения производительности — это реальные приложения. По их мнению, попытки экспериментировать с программами, которые намного проще, чем реальное приложение, приводят к проблемам с прогнозом производительности будущих результатов. Поэтому организаторы не включали в качестве эталонных задач ядра, представляющие собой небольшие ключевые части реальных приложений, игрушечные программы из начальных заданий по программированию и синтетические тесты, представляющие собой небольшие «поддельные» программы, созданные для того, чтобы попытаться сопоставить профиль и поведение реальных приложений, что было бы представлением модели. Для большинства задач было выполнено несколько вариантов решений, показавших различную производительность.

Всё это напоминает попытки в 1970-е годы накапливать пакеты прикладных программ (ППП) с целью создания пакетов для решения вновь возникающих проблем. В действительности же оказалось, что для новых задач в конце концов приходится создавать новые пакеты, а переделка старых слишком трудоёмка и ненадёжна. Было признано, что реальные ППП слишком разнообразны и несравнимы. Не исключено, что на уровне приложений ситуация подобна. Не существует исчисления приложений,

позволяющего их сравнивать, если они решают разные задачи и применяются в совершенно разных условиях.

При этом в разных приложениях могут быть представлены функционально эквивалентные модули, программируемые решения в которых отличаются и по продуктивности, и по производительности. Модули могут быть типовыми, функционально эквивалентными для разных приложений, их можно сравнивать и делать выводы, что модуль конкретного назначения в одном приложении по производительности или продуктивности отличается от модуля этого же назначения в другом приложении. Разнообразие приложений похоже на разнообразие картинок в калейдоскопе, формирующихся из небольшого числа разноцветных элементов.

Принятый в эксперименте ход мысли выглядит логично, хотя он в принципе закрывает путь достижению ясной метрики для измерения производительности программ и оценки вклада программируемых решений. Нет оснований полагать, что вдруг возникнет нечто вроде исчисления приложений, пригодного для прогнозов, особенно на фоне отсутствия заинтересованности ИТ-индустрии в разработке метрики для измерения производительности программных приложений. Именно чрезмерным разнообразием программ и приложений экономисты объясняют отсутствие принятой методики измерения качества программ. В этом отношении образовательная система обычно должна выполнять функцию оценки уровня знаний и умений обучающихся, и это достаточная причина для заинтересованности в создании методики оценки результатов выполнения учебных заданий по программированию. Пусть дистанция от учебных и синтетических программ до реальных программных приложений велика, но она может быть постепенно, систематически преодолена, дав необходимые научные основания для более общей методики измерений

продуктивности программирования и производительности программ.

Для экспериментальных измерений была выбрана небольшая коллекция эталонных задач (n-body, spectral-norm, mandelbrot, pidigits, fasta, k-nucleotide, reverse-complement, binary trees, simple)[7], к решению которых предъявлены чёткие требования по методу их решения и объёму обрабатываемых данных, гарантирующие функциональную эквивалентность программ для решения одной и той же задачи. Измерение характеристик таких сравнительно небольших, программ не вполне отражает производительность любых реальных приложений; тем не менее, они могут служить ориентиром при обнаружении аналогий и отладке механизма измерения.

Характерно, что при этом организаторы эксперимента признали, что за почти 20 лет проекта у них не было времени проверять исходный код реальных приложений, чтобы убедиться, что разные реализации программы сопоставимы, что действительно это одна и та же программа, написанная на разных ЯП. Не было проверки программ на семантическую эквивалентность, что означает зависимость результата от программируемых решений, а не только от эталонных задач. Эта проблема связана и с тем, что ЯП развиваются и эталонность задач на 2012 год порой могла утрачиваться к 2021 году, как это произошло с JavaScript. Впрочем, эталоны в палате мер и весов приходится время от времени подвергать проверке.

Сравнение программ друг с другом происходило, будто разные ЯП были созданы для одной и той же цели, что отнюдь не так. Остаются не рассмотренными проблемы, связанные с многоядерными процессорами, сетевыми

[7] Постановки задач, требования к их решению и программы их решения на всех испытуемых ЯП представлены на сайте проекта

системами, вычислительными кластерами и веб-программированием. Изменился масштаб: сегодняшние серверные программы состоят из десятков миллионов строк кода, над ними работают сотни, а то и тысячи независимых программистов; такие программы обновляются буквально каждый день. Для многих программ критерии скорости и объёма менее важны, чем надёжность и безопасность обработки данных непрерывно изменяющимися инструментами.

Организаторы сравнения ЯП признали, что им пока не удаётся определять относительную производительность ЯП,; накоплен лишь опыт, чтобы понять, что следует или можно измерять и тестировать, чтобы получить простое представление об общей производительности того или иного ЯП, точнее, его компилятора[8]. Результаты проведённого чемпионата для небольшой группы ЯП (C++, Go, Clisp, Haskell, Java, Pascal, Python) приведены в таблице 1.

Видно, что разброс скорости редко превосходит 1 к 10, в то время как он мог бы быть и 1 к 50. Возможно, это объясняется малым числом выбранных ЯП. Интересно, что набирающий популярность Python в этом чемпионате часто оказывается на последних местах. Это подтверждает исходное предположение о слабом интересе практиков к измерению производительности программ и даже систем программирования. Не исключено, что Python был бы чемпионом в соревновании на скорость получения работоспособной программы, на технологичность процесса разработки программ, на продуктивность программирования, что говорит о полезности производительность компиляторов понимать шире, чем качество кода.

[8] Gouy, Isaac. The Computer Language Benchmarks Game. Web. https://benchmarksgame-team.pages.debian.net/benchmarksgame/

Таблица 1. Результаты измерения скорости программ на конкретных ЯП для разных задачах
(результат измерения под наименованием языка)

Задача	Порядок ЯП[9]	Пропорция и диапазон
Fannkuch-redux	**C++** 03.26 **Go** 8.31 **Clisp** 9.38 **Haskell** 10.33 **Java** 10.48 **Pascal** 10.56 **Python** 5 min	1/3 3.26 – 10.56 **/5 min Python**
n-body	**C++** 2.17 **Pascal** 6.28 **Go** 6.37 **Haskell** 6.43 **Java** 6.77 **Clisp** 7.70 **Python** 9.00	1:4 2.17 – 9.00
spectral-norm	**C++** 0.72 **Go** 1.43 **Clisp** 1.44 **Pascal** 1.44 **Haskell** 1.47 **Java** 1.55 **Python** 112.97	1/2 0.72 – 1.55 **/112.97 Python**
mandelbrot	**C++** 03.26 **Haskell** 8.31 **Go** 9.38 **Pascal** 10.33 **Java** 10.48 **Clisp** 10.56 **Python** 5 min	1/5 0.84 – 4.17 / **177.35 Python**
pidigits	**C++** 0.59 **Pascal** 0.73 **Java** 0.79 **Go** 0.86 **Python** 1.16 **Haskell** 1.44 **Clisp** 3.17	1/6 0.59 – 3.17
fasta	**C++** 0.77 **Haskell** 0.86 **Java** 1.20 **Go** 1.27 **Clisp** 4.38 **Pascal** 5.58 **Python** 36.90	1/6 0.77 – 5.58 **/ 36.90 Python**
k-nucleotide	**C++** 1.96 **Java** 4.83 **Go** 7.46 **Haskell** 11.69 **Clisp** 15.12 **Python** 46.31	1/8 1.96 – 15.12 **/46.31 Python**
reverse-complement	**C++** 0.64 **Go** 1.34 **Java** 1.57 **Haskell** 3.16 **Pascal** 3.69 **Clisp** 5.69 **Python** 6.63	1/10 0.64 - 6.63
binary-trees	**C++** 0.94 **Pascal** 2.04 **Java** 2.51 **Haskell** 4.42 **Clisp** 5.44 **Go** 12.48 **Python** 44.70	1/13 0.94 – 12.48 **/44.70 Python**

9 Clojure был потерян в процессе эксперимента вместе с ещё рядом ЯП

Для наглядности можно результаты выразить в стиле итогов олимпиады как на таблице 2.

Таблица 2. Результаты чемпионата на скорость

Золото:	**C++**		
Серебро:		**Pascal Haskell Java Go**	
Бронза:			**Clisp**
Приз зрительских симпатий	**Python**		

В этом плане показательны результаты другого проекта, посвященного сравнению языков Lisp и Java (Lisp as an Alternative to Java), выполненного 14-ю программистами-добровольцами, написавшими 16 программ (12 на Common Lisp и 4 на Scheme) [10], производительность которых сравнима с C/C++ и Java с точностью до минимизации рисков при выполнении проектов [11]. Результаты эксперимента показали, что быстрейший Clisp или Scheme уступают по скорости программ быстрейшим C/C++[12]. При сравнении производительности с C/C++ и Java констатировано, что Clisp обладает более быстрым циклом отладки программ, в сравнении с C/C++ или Java.

При этом программисты с меньшим опытом программирования на Clisp или Scheme нередко программируют быстрее и лучше, чем более опытные программисты на C/C++ или Java. Авторы эксперимента полагают, что программирование на языке Lisp имеет тенденцию повышать квалификацию программистов, делать

---

[10] В источнике нет намёка на характер задач.

[11] https://psachin.gitlab.io/lisp-java-notes.html Erann Gat, Jet Propulsion Laboratory,
California Institute of Technology, Pasadena, CA 91109, gat@jpl.nasa.gov

[12] Кто бы сомневался!

их хорошими программистами на любом ЯП. По их мнению, имеет значение и то, что Lisp обладает многими особенностями, делающими его языком, лёгким для изучения и применения.

Не исключено, что, программируя на языке Lisp, программисты повышают свой уровень потому, что этот язык даёт им больше времени на хорошее программирование, на продумывание и улучшение программ. Справедливости ради следует отметить, что многие рекомендации по постановке обучения программированию рекомендуют начинать с функционального программирования.

Традиционный вопрос: почему столь замечательный язык, как Lisp, незаслуженно непопулярен, пока не получил простого ответа. Одна из гипотез исторических причин объясняет это разочарованием в возможностях искусственного интеллекта, который «выпал из тележки», потерял финансовую поддержку в середине 1980-х из-за неспособности выполнить свои высокие обещания. Тут и Lisp, как основной язык искусственного интеллекта, попал «под раздачу». Однако именно в эти годы выполнена стандартизация языка, констатирующая два стандарта — академический LISP1, включающий в себя Scheme, и производственный LISP2, представленный языком Clisp. Более вероятно, что другие причины сильнее, ещё одна из которых — это предрассудок или миф, что Lisp слишком большой и медленный одновременно с утверждением, что он лёгкий и элегантный. В противовес этому мнению авторы сравнения языков Lisp и Java приводят результаты своего эксперимента, часть которых показана в Таблице 3. Интересно, что в этом сравнении представлены характеристики продуктивности программирования и квалификации программистов. Не исключено, что корни причин предстоит искать в особенностях формирования интуитивной грамматики деятельности при обработке

информации человеком.

Таблица 3.Результаты сравнения производительности языков

	Lisp	**C, C++**	**Java**
Development time	2 to 8.5 hrs.	3 to 25 hrs.	4 to 63 hrs.
Avg. Programmer's experience	6.2 yrs.	9.6 yrs.	7.7 yrs.
Execution time	Fast	Faster	-
Runtime	- Better than C & C++		
	- Much better than Java		
Runtime(mean)	41 sec.	165 sec.	-
Runtime(media)	30 sec.	54 sec.	-
Runtime(standard deviation)	11 sec.	77 sec.	-
Memory consumption	- Higher than C & C++		
	- Comparable to Java		

В целом результаты вышеописанных экспериментов показывают, что проблема измерения производительности программ далека от конструктивного решения. При организации эксперимента просматривается мысль, характерная для обоснования лёгких путей, будто рассматривая постановки эталонных задач и программы их решения на разных языках, программист сможет интуитивно уловить аналогию или подобие со своей задачей и таком образом выбрать подходящий компилятор. Вызывает сомнение, что такая интуитивная аналогия будет достаточным основанием на практике для отказа от знакомого компилятора и перехода на незнакомый, возможно, более производительный, но чреватый неожиданностями. Организаторы эксперимента приняли ряд важных решений и констатировали ряд не преодолённых трудностей.

1) Они признали, что нужны эталонные задачи, но не дали пояснений относительно формата и содержания

эталонных задач, а также механизма сохранения эталонности в условиях непрерывного развития ЯП. Можно согласиться, что метрика для решаемых задач выражается в терминах эталонных задач, а метрика для приложений — в терминах эталонных приложений, но выбор эталонов требует чёткости, пока не достигнутой. Представленные результаты не содержат выводов относительно того, какие из представленных задач оказались успешными в роли эталонов.

2) Нет рекомендаций по сопоставлению реальных приложений с предложенными эталонными задачами. Идея улавливать интуитивно сходство между эталонами и решаемыми задачами означает отсутствие удостоверяющего механизма использования таких метрик, а это почти признание невозможности измерений.

3) Отсутствует освещение взаимосвязи между продуктивностью программирования и достижением производительности программ. Анализ данных по итогам проведённого чемпионата показывает, что организаторы не сочли полезным представить или собрать сведения о трудозатратах на подготовку и отладку программ решения эталонных задач. Обычно любая экономика приходит к пониманию оптимального соотношения цена-качество, допускающего вычисления оптимума методами типа мини-макс. Если качество — это производительность программ, то цена — это продуктивность разработки или улучшения программ. Поэтому измерение производительности имеет смысл при умении измерять и/или прогнозировать продуктивность. В этом отношении системы обучения программированию и проведения конкурсов по программированию обычно имеют данные или правдоподобные оценки относительно предполагаемых трудозатрат на выполнение учебных и олимпиадных задач.

4) Ещё одна сторона оценки продуктивности и производительности связана с зависимостью от

квалификации, способностей, образования и опыта программистов. Эта сторона соприкасается с вопросами управления проектами, ещё более трудными для исследований, чем программирование [18, 19].

Обратим внимание, что шаги прогресса технологий программирования обусловлены повышением его технологичности, а именно, выбором методов и техники программирования, позволяющих процесс разработки и применения программ сделать не только продуктивным, но, что более важно, надёжным, выполнимым в реальные сроки и менее требовательным к квалификации пользователей. Такие приоритеты связаны с повышением продуктивности программирования одновременно с расширением сферы применения программ, чреватым потерями в производительности программ. Нередко происходит и снижение требований к квалификации программистов. Остаётся открытым вопрос об оптимальном соотношении между технологичностью программирования и производительностью программных приложений.

2. «Дайте мне точку опоры!…»

Эти слова Архимеда в практическом программировании понимают как «Дайте мне доступ к компилятору с самой маленькой работающей программой, типа вывода строки или числа, остальное я пойму сам в непосредственном эксперименте». Работающий компилятор для практика является единственно достоверным документом о реализованном ЯП. Пользовательская документация и описание стандарта ЯП — не более чем справочный материал, нуждающийся в проверке на соответствие тому, что фактически поддерживает компилятор.

Именно такое понимание процесса изучения ЯП лежит

в основе бенчмарка «Курсы и задания»[13], содержащего онлайн-компиляторы, калькулятор и встроенный редактор, позволяющие осваивать программирование восходящим методом снизу вверх на базе любых из 76-ти ЯП и двух баз данных. Разработчики бенчмарка призывают организовывать онлайн-студии, позволяющие обучение программированию сопровождать автоматической оценкой результатов, получаемых на основе измеримых показателей. Измеримость таких показателей позволяет в постановки учебных задач включать оценку качества программирования, а не просто работоспособность программ. По их мнению такой инструмент может выполнять роль онлайн-тестов для само-диагностики уровня квалификации и для проведения собеседований при приёме на работу.

Появление таких бенчмарков, — а их уже немало и становится всё больше, — возможно, является симптомом смутного понимания профессиональным сообществом беспомощности систем образования в поддержке обучения программированию. Об этом же свидетельствуют массовые обращения студентов за помощью в Интернет при решении учебных задач, даже очень простых.

Среди рекомендаций встречаются и вполне компетентные, заслуживающие коллекционирования и встраивания в системы обучения, но это отдельная задача, пока не доросшая до кристаллизации. Всё это затуманивается массой обучающих инициатив, утверждающих, что программированию обучиться можно быстро и без особых усилий, буквально за считанные дни.

Бенчмарк «Курсы и задания» не гарантирует точности или надежности предложенных материалов, что не мешает использовать его в экспериментах и в учебном процессе. Калькулятор содержит команды, позволяющие быстро

[13] Courses And Assignments https://www.jdoodle.com

накопить опыт работы на уровне базовой семантики ЯП и отладки программ. В их числе команды управления ходом выполнения программы, управления файлами и общими ресурсами, основные категории функций: арифметические, битовые, логические, матричные, вероятностные и другие, что позволяет освоить элементарные средства обработки данных на универсальном доязыковом уровне, сформировать интуитивную грамматику процессов обработки данных, начиная с ручного оперирования.

Независимо от калькулятора имеется возможность работы с большим числом компиляторов для ЯП, поддерживающих разные парадигмы программирования. Доступен простой интерфейс для выполнения серии сеансов, включая работу со списком недавно выполненных программ, который хранится локально в браузере. Выполнение учебных программ на любом из доступных ЯП сопровождается выводом процессорного времени, потребовавшегося программе, и максимального объёма памяти, использованной программой. Это даёт основания ставить учебные задания на оптимизацию программируемых решений и сравнение стилей программирования. Вход в каждый компилятор оснащён маленькой работоспособной программой вывода приветствия, вроде (print "welcome to CLISP online ide from Jdoodle.com") или

```
#include <iostream>
using namespace std;
int main() {
   int x=10;
   int y=25;
   int z=x+y;
   cout<<"Sum of x+y = " << z;
}
```

Этого достаточно для технической поддержки большинства учебных задач по системному программированию и тестированию программ. Бенчмарк разработчики предлагают встраивать в дистанционные курсы. Предложенный механизм достаточен для самостоятельного изучения ЯП, а заодно и для экспериментов по измерению производительности программ. Остаются за бортом вопросы решения новых или слабо исследованных задач, характерных для современного программирования в условиях стремительного совершенствования аппаратуры и резкого расширения круга пользователей, не обязанных обладать познаниями в сфере теоретического и практического программирования.

3. Опыт оценивания учебных и олимпиадных программ решения учебных и олимпиадных задач “А у нас в квартире газ. А у вас?”

Автоматизированное оценивание эффективности программ во многом напоминает автоматическое тестирование олимпиадных задач по программированию. Но если первое еще находится на этапе становления, то второе уже прошло большой путь развития, поэтому для того чтобы иметь возможность позаимствовать удачные решения и не совершить похожих ошибок, совершим небольшой экскурс в историю олимпиадного тестирования.

На заре школьного олимпиадного движения по программированию (напомним, что предмет “Основы информатики и вычислительной техники” появился в программе российских — тогда еще советских — общеобразовательных школ в 80-х годах прошлого столетия) олимпиады всех уровней проводили с обязательным включением в них теоретического тура, причем на олимпиадах нижних уровней — школьном и даже районном

— практического тура могло вообще не быть. Объяснялось это, конечно же, весьма небольшим количеством имевшихся в школах компьютеров.

Как проходили теоретические туры по программированию? Текст программы или алгоритма участники писали на бумаге, а проверяющие затем читали эти тексты, выступая в роли своеобразных синтаксических анализаторов и виртуальных машин, имитируя выполнение программ у себя в голове.

Задания практического тура уже тогда проверялись при помощи тестирования по типу “черного ящика” (способ тестирования, когда заключение о внутренних свойствах исследуемого объекта делается лишь на основании его откликов на различные раздражители). Организаторы олимпиады вручную запускали каждую программу с заранее подготовленными входными данными и фиксировали результаты выполнения.

Даже на олимпиадах городского уровня участникам разрешалось писать программы с чтением данных не из файла, а с клавиатуры, и ввод перекладывали опять же на проверяющих.

Тем не менее, даже в таких условиях основные черты тестирования по типу “черного ящика” просматривались уже тогда.

Во-первых, разрешалось пользоваться почти любым известным участнику языком программирования (к счастью, тогда их выбор был довольно ограничен). В число разрешенных входил даже язык РАЯ (Русский Алгоритмический Язык, разработанный под руководством А. П. Ершова специально для первого школьного курса программирования).

Во-вторых, хотя организаторы олимпиады принимали исходные тексты программ, а компиляция и выполнение тестов производились вручную, собственно тексты

рассматривались только в исключительных — спорных или непонятных — случаях.

В-третьих, общее время работы каждого теста не фиксировалось — следили лишь за тем, чтобы в процессе выполнения не происходило зацикливания.

И, в-четвертых, вывод о правильности или неправильности решения и реализующей его программы строился на основании результатов выполнения единых для всех участников тестов.

Автоматизация процесса проверки олимпиадных решений началась практически одновременно с возникновением программистских олимпиад, была, разумеется, постепенной и на каждом этапе использовала все имеющиеся на тот момент возможности вычислительной техники. Однако от принципа проверки по типу “черного ящика” не отступали никогда. Результат мог выглядеть двояко: “*решение прошло все тесты из тестового набора – участник получил оговоренное количество баллов, иначе получил 0 баллов*” либо “*решение прошло только некоторые тесты из тестового набора — участник получил не максимальное, но и не нулевое количество баллов*”. Конечно же, во втором случае стоимость “тяжелых” тестов была заметно выше, и участники, выбравшие более эффективные алгоритмы решения, получали преимущество. Эти два подхода к оценке прекрасно уживаются и по сей день.

Когда развитие компьютерных сетей и операционных систем сделало возможным автоматизировать сбор решений, их компиляцию и выполнение, появились некоторые ограничения, обусловленные особенностями тестирующих систем: только файловый ввод-вывод данных, использование только “одобренных” компиляторов и т.п. — и дополнительный критерий: эффективность выбранного алгоритма. Алгоритм решения не должен приводить к зацикливанию, переполнению памяти, критическим ошибкам

типа “деление на ноль” и т.п. даже в редких частных случаях, а также должен уметь справляться с большими объемами данных, укладываясь при этом в заданный временной отрезок. Например, для того чтобы “отсеять” решения, использовавшие не алгоритм Дейкстры (его сложность, напомним, пропорциональна N^2, где N — количество вершин в графе), а рекурсивный полный перебор (сложность пропорциональна 2^N), в тестовый набор вводились тесты с большим количеством вершин N, что заведомо выводило переборный вариант за временные ограничения.

Автоматический подсчет количества времени, затраченного на выполнение программы, ведется и в системе сравнения эффективности программ. Но здесь на первый план выходит еще одна тонкость: применение оптимизирующих компиляторов. На многих олимпиадах жестко оговаривается не только набор разрешенных компиляторов, но и набор разрешенных опций компилирования. Одна из таких спорных опций – оптимизация. Поскольку ее результат довольно сильно зависит от конкретной реализации выбранного участником алгоритма, наиболее правильным подходом, уравнивающим всех участников, является отключение этой опции: она может исказить результаты измерений как в ту, так и в другую сторону.

Другим результатом автоматизации тестирования стало исчезновение семантического анализа представленных решений. Если человек-проверяющий может понять смысл программы, то автомату это недоступно. Тестирование олимпиадных программ свелось только к проверке их работоспособности на некотором тестовом наборе. “Черный ящик” победил. И если в первые годы школьного олимпиадного движения вполне была возможной ситуация “*участник взял эффективный алгоритм, но в его реализации допустил ошибку, — мы добавим ему поощрительные баллы,*

хотя его решение и не прошло ни одного теста”, то при тестировании по типу “черного ящика” учитывается только факт успешного прохождения тестов.

Но является ли программа, успешно прошедшая все тесты из тестового набора, правильным решением поставленной задачи? Разумеется, это напрямую зависит от тестового набора. Например, при отсутствии в тестовом наборе тестов, описывающих какой-либо редко встречающийся частный случай, нельзя утверждать, что программа является правильной. Она лишь правильно работает на некоторых наборах входных данных.

На многих олимпиадах в конце соревнования набор проверочных тестов становится доступным всем участникам и на основании анализа этого тестового набора даже принимаются апелляции. Но в системе бенчмарка «Какой язык быстрее» набор тестов, на основании которых делается вывод о эффективности или неэффективности предложенной программы, скрыт, что делает не очень достоверными результаты проверки. Чаще всего “камнем преткновения” в вопросе эффективности выбранных языка программирования, алгоритма и его реализации являются именно частные или граничные случаи. Понятно, что при $N=10$ разница в эффективности упомянутых ранее алгоритма Дейкстры и рекурсивного алгоритма полного перебора будет сравнима с ошибкой измерений. И совсем другая картина сложится при $N=10\,000$. Видим, что невозможно выстроить достоверную систему сравнения эффективности программ без всестороннего анализа области допустимых входных данных.

Таким образом, автоматическое тестирование решений задач по программированию и измерение эффективности реализаций различных алгоритмов на различных языках программирования действительно имеют много общего. Хочется, чтобы измерение эффективности в своем дальнейшем развитии учло и “обезвредило” те особенности

автоматического тестирования, которые способны исказить результаты измерений. Здесь мы намеренно обходим вопрос невозможности сравнить несравнимое и верим, что читатель не станет пытаться сравнить апельсин с симфонией и ограничится сравнением апельсина с мандарином, лимоном, грейпфрутом и лаймом.

Подготовка и проведение лабораторных работ и олимпиад по программированию включает в себя ряд уже зарекомендовавших себя средств и методов комплектации и проверки заданий и отдельно оценивания и тестирования программ решения задач [20-23].

Достигнуто понимание ряда особенностей процесса выбора задач и статистика оценивания успешности их решения:

- требования к постановкам задач,
- планируемое время на их решение,
- известно кто и как реально справился с заданиями,
- известны типичные категории задач, упорядоченные по сложности.

Первые эксперименты можно продолжить на числовых функциях, легко масштабируемых одним параметром: факториал, Фибоначчи, простые числа, степенные ряды и т.п. [18]. Можно попробовать вещественные числа в форме решения квадратных уравнений или извлечения корней или дробных формул с риском деления на ноль. Можно и что-то наворочать с символьной обработкой, типа перестановок или разрастающихся цепочек [19].

Шкала сложности программируемых решений для начала может различать элементарные операции, простые переменные, If, доступ к соседнему элементу последовательности, строки, вектора, списка, стека — прямой перебор.

Второй уровень — функции/процедуры, иерархия

областей действия, разные виды ветвлений, структуры данных.

Третий уровень — ограничения, диагностики, ловушки, контроль типов данных, рекурсия-циклы.

Четвёртый уровень — отображения, фильтры, свёртки, преобразования, генераторы/конструкторы, макросы.

Пятый уровень — много-поточность и имитация обменов данными между потоками без сложных взаимодействий.

Важно, что связывание производительности программ с программируемыми решениями при таком подходе допускает сопоставление со спектром используемых средств языка или эталонными задачами. Разброс производительности программ, кроме того, обусловлен продуктивностью программирования, т.е. квалификацией программистов, владеющих не только методами решения таких задач, но и определёнными способностями, средствами, техникой программирования. Всё это дополняется умением самостоятельно решать проблемы в пространстве, выходящем за пределы языка и системы программирования, что более подробно описанном в работах [24-32], представляющих парадигмальную методику анализа и сравнения языков программирования, а также неформализуемые особенности процесса разработки программ.

Конечно, полное изложение почти сорокалетнего опыта олимпиадного, спортивного программирования заслуживает отдельной статьи; здесь приведём лишь краткую справку, связанную с продуктивностью программирования и олимпиадный опыт оценки программ через тестирование с учётом времени решения задач в комплекте. Это показывает проблему измерения качества программ с точки зрения

принятия решений с одной стороны участниками, с другой стороны организаторами программистских чемпионатов. Отражает ли интегральная оценка продуктивность работы участников и производительность сделанных ими программ? Что из опыта оценки олимпиадных работ может получить развитие как методика оценки учебного и производственного программирования, или что обладает сходством, а что резко отличается?

4. Эксперимент по измерению производительности программ

В данном разделе представлены результаты предварительного точечного эксперимента по измерению производительности программируемых решений на базе бенчмарка «Курсы и задания»[14], выполненного на примере вычисления чисел Фибоначчи. Целью эксперимента является «зондирование почвы» — показ возможности создать методику выделения вклада программируемых решений из непосредственно измеримых характеристик комплекса, включающего в себя программу, компилятор и аппаратуру. Эксперимент назван точечным, т. к. для полноценного измерения нужен сбор статистики, что в данном случае не сделано. Выбор вычисления чисел Фибоначчи не случаен. Наивный алгоритм интересен тем, что даёт быструю потерю производительности. Теоретическая оценка сложности алгоритмов недостаточна по той причине, что производительность зависит ещё от диапазонов значений и структур данных. Здесь нет акцента на обучение, а только на измерение, показывающее бесспорную зависимость производительности программ от алгоритма, рассматриваемого как программируемое решение. Этот и

[14] Courses And Assignments https://www.jdoodle.com

другие алгоритмы решения этой задачи присутствуют во многих источниках.

В Приложении 1 приведены запрограммированные определения функций вычисления чисел Фибоначчи, выбранные для эксперимента на компиляторах для ЯП C++, Go, Clisp, Java, Haskell, Pascal, Python, Clojure. Результаты измерения производительности семантически эквивалентных решений на этих ЯП приведены в таблице 4.

Смысл показателей в таблицах 4, 5, 6, 8 и 10:

Грань — величина наибольшего полученного числа.

Не ноль — число с первым ненулевым значением времени.

Время — показатель процессорного времени по завершении вычисления.

Память — максимальный объём памяти для всей серии измерений.

Min — обнаружение ненулевого процессорного времени.

Max – самое большое число и показатели на нём.

Esc — причина аварийного завершения вычислений.

Предварительный характер эксперимента проявляется и в том, что может быть по разному определена граница с операционной системой, дающая разницу в показателях на разных прогонах одной и той же программы — эффект разогрева. Разница в показателях Java и Clojure, вероятно, объясняется различием в границах счёта. Нет и учёта эффектов типа «перегрева». Это неудивительно, ведь языки программирования создаются для разных целей.

Разница по номинациям показана в Таблице 5, в которой каждый столбец показывает разницу чемпиона и сравниваемых с ним остальных языков.

Семантическая эквивалентность измеренных программ позволяет разницу в производительности относить на счёт разницы в компиляторах и аппаратуре, запрограммированные решения одинаковы. Можно обратить внимание на то, что по каждому показателю выделяются свои чемпионы и

Таблица 4. Семантически эквивалентные функции на C++, Go, Clisp, Java, Haskell, Pascal, Python, Clojure

Язык	Показатели						Комментарий
	Грань	Не ноль	Время		Память	**Esc**	
			Min	Max			
C++	**46**	**32**	0.20	**10.27**	**3436**	**Time out**	До первого ненулевого показателя времени выполняется большой объём работы
Go	**46**	2	0.14	**10.65**	**42808**	**Time out**	Похоже, код выполняет много предварительного
Clisp	**35**	19	0.01	**13.06**	**9720**	**Time out**	Объём мал, остальные показатели без особенностей
Java	**48**	2	0.07	**14.24**	**35408**	**Time out**	Похоже, код выполняет много предварительного
Haskell	**42**	27	0.01	**12.95**	**5452**	**Time out**	Компилятор весьма хорош для такого языка
Pascal	**44**	29	0.01	**11.00**	**412**	**Time out**	Поразительно экономен
Python	**38**	3	0.01	**10.41**	**7756**	**Time out**	Код выполняет много предварительного, зато быстро был готов тест
Clojure	**44**	2	3.54	**8.74**	**236728**	**Time out**	Код выполняет много предварительного и беспечно расходует память

Таблица 5.Чемпионы по номинациям: объёмы данных, наблюдаемое и максимальное время, объём памяти

№	*Показатели*					*Комментарий*
	Грань	*Не ноль*	*Время*		*Память*	
			Min	Max		
1	48 Ja.va	33 C++	0.01 Clisp	8.74 Clojure	412 Pascal	C++ – эффективность по времени, Pascal – по памяти
2	46 C++	29 Pascal	0.01 Pascal	10.08 C++	3408 C++	Pascal – самых экономный по памяти и наглядный
3	46 Go	27 Haskell	0.01 Python	10.94 Python	5452 Haskell	Python – низкий быстрый старт, средне-затратный без чрезмерного расхода памяти
4	44 Clojure	19 Clisp	0.01 Haskell	11.00 Pascal	7756 Python	Clisp – ранний финиш, медленнее других, малая память
5	44 Pascal	3 Python	0.07 Java	11.56 Go	9720 Clisp	Java при широком диапазоне медленный и затратный
6	42 Haskell	2 Go	0.14 Go	12.95 Haskell	35408 Java	Haskell – быстрый для малых, затратный, экономный по памяти
7	38 Python	2 Java	0.20 C++	13.06 Clisp	42808 Go	Go – низкий старт, средне-затратный
8	35 Clisp	2 Clojure	3.54 Clojure	14.24 Java	236728 Clojure	Clojure — низкий старт, медленный для малых, быстрый для больших при растрате памяти
От и до	35 - 48	2 - 33	0.01 - 3.54	8.74 - 14.24	412 - 236728	Разброс по времени и объёму невелик, заметнее разброс по памяти

аутсайдеры, производительность является относительной не только по ЯП, но и по показателю измерения.

Ни одна вертикаль не совпадает с последовательностью «C++, Go, Clisp, Haskell, Java, Python» из таблицы 1, решения эталонных задач там не анализировались на семантическую эквивалентность. Варианты программируемых решений одного и того же метода могут использовать разные средства ЯП и потому быть семантически не эквивалентными.

Из таблицы 5 сразу видно, что программа на языке Java допускает наибольший диапазон данных. Язык C++ работает столь быстро, что обработку малых объёмов там трудно замерить. Язык Clisp поддерживает заботу об обработке малых заданий, даже во многом давая им приоритет. Похожий на него Clojure — лидер на больших объёмах и активный «пожиратель памяти». Компилятор языка Haskell во многом компенсирует недостатки языка. Go и Java близки по ряду показателей. И конечно, язык Pascal показывает, что для решения большинства учебных задач многого не надо.

Результаты измерения производительности функционально эквивалентных вариантов, не обладающих семантической эквивалентностью, приведены в Таблице 6.

Разница в производительности таких вариантов 1 к 1000 уже объясняется программируемыми решениями, вклад которых может быть выделен из общих результатов непосредственных измерений. Интересно, что близкий разброс показателей был получен в начале 1970-х годов фирмой IBM при исследовании продуктивности труда программистов
с учётом производительности полученных программ [18]. Сравнительно очевидно, что более высокую производительность дают средства более низкого уровня, учитывая, что переход от ассемблера к языкам высокого уровня пожертвовал именно производительностью программ

Таблица 6. Варианты используемых средств одного ЯП: Go, Clisp, Haskell

Вариант	Показатели					Esc	**Разброс показателей**
	Грань	Не ноль	Время		Память		
Go итер	4453	2	0.19	0.28	40668	- 1	Без прерываний и диагностики переходит в отрицательную область
Go рек	46	2	0.14	10.65	42808	Time out	Исчерпано время
От и до	**46 - 4453**			**0.28 - 10.65**			**Грань — 100, время – 5**
1Clisp рек	36	11	0.01	'01.12	9720	Time out	Уровень базовой семантики
2Clisp рек	36	14	0.01	13.83	10524	Time out	Сравнимо с языком высокого уровня
3Clisp рек	35	19	0.01	13.16	9636	Time out	Сравнимо с языком высокого уровня
4Clisp итер	10000	20	0.01	0.01	14404	stack overflow	Стек и буфер вывода не достаточно универсальны

5Clisp итер	2972	2800	0.01	0.01	14048	stack overflow	
6Clisp line	3384	15	0.01	0.01	14764	stack overflow	
7Clisp прог	34056	2	0.01	0.36	10584	output Limit reached	
8Clisp loop	35	20	0.01	12.40	9748	Time out	Исчерпано время
От и до	**35 - 34056**	**2 - 2800**		**0.01 - 13.83**	**9720 - 14048**		**Грань — 1000, не ноль- 1000, время — 130, память — 1.5**
Haskell fn	42	27	0.01	12.95	5452	Time out	Исчерпано время
Haskell opt	33832	32000	0.01	0.01	5776	output Limit reached.	Буфер вывода не достаточно универсален
От и до	**42 - 33832**	**27 - 32000**			**5452 - 5776**		**Грань — 1000, не ноль- 1000**

в пользу повышения продуктивности программирования и его технологичности.

Завершение вычислений с диагнозом «output Limit reached» или «stack overflow» означает, что разработчики компилятора не придавали особого значения существованию поддержки чисел произвольной длины. Получилось, что вычислить очень длинное число можно, а вывести его на печать не удаётся.

Со стеком сложнее — здесь, видимо, не учтено, что активная часть стека обычно не слишком велика, и поэтому, контролируя реальные его границы, придонную часть стека можно своевременно выгружать во вспомогательную память, а потом возвращать.

Разброс показателей намного превышает разницу между прогонами на разных компиляторах и даже переносом на уровень элементной базы. Подобную разницу можно видеть и по результатам игры в самый быстрый компилятор, но, как правило, с заметно меньшим разбросом (таблица 7).

Располагая таким данными, можно сравнить исходные тексты программ на одном ЯП. Беглый взгляд на самую быструю и самую медленную программы, написанные на языке Clisp одним и тем же программистом (Jon Smith), показывает, что заметная разница в скорости, скорее всего, объясняется использованием в быстрой программе макросов и многопоточности.

Особо следует отметить императивный стиль представления этих программ, использующих преимущественно циклы и присваивания, что не позволяет на основе таких измерений соглашаться с выводами

относительно производительности функционального программирования в сравнении с императивным или объектно-ориентированным. Можно провести сравнение вариантов из таблицы 5 с их моделями на языке Clisp,

семантически эквивалентными этим вариантам (см. таблицу 8).

Разница между вариантами функционально эквивалентных программ, не обладающих семантической эквивалентностью, на одном и том же ЯП для приведённых примеров существенно выше, чем разница между семантически эквивалентными программами на разных ЯП.

Пока рассмотрено слишком мало примеров и ЯП для уверенных выводов. Их достаточно лишь для гипотезы, что кроме эталонных задач, следует выделять эталонные ЯП, приведение к которым снимает зависимость результатов непосредственных измерений от компилятора и аппаратуры, а также от спектра оптимизирующих преобразований программ.

Пожалуй, первые кандидаты в эталонные языки – это языки функционального программирования, такие как Clisp, Scheme или Clojure, обладающие высокой моделирующей силой и приспособленные к варьированию методов обработки программ.

Далее, для каждого ЯП можно составить понятийную таблицу и представить ряд фрагментов программ, достаточный для создания краткого описания ЯП, удобного для перехода к практике программирования на на нём. Таким образом создаётся шкала сопоставимых программ на разных ЯП, при тестировании и измерении характеристик которых на общих платформах модно делать выводы о вкладе языковых конструкций в производительность программ на основе прямых измерений.

Оценка продуктивности программирования требует дополнительной методики оценивания интуитивных механизмов решения задач человеком, что выходит за пределы данной статьи.

Использование шкалы сопоставимых методов их решения задач, включая прагматику СП, даёт критерии для

Таблица 7. Разница скоростей вариантов программ на эталонных задачах чемпионата на одном ЯП *(результаты измерения в нижней строке под наименованием языка, выше самый быстрый, ниже самый медленный)*

Задача	Порядок ЯП							Разброс скорости
Fannkuch-redux	**C++**	**Go**	**Clisp**	**Haskell**	**Java**	**Pascal**	**Python**	1/100 **Java**
	03.26	8.31	9.38	10.33	**0.48**	10.56	5 min	
	43.07	11.87	47.26	50.62	**46.08**		18 min	
n-body	**C++**	**Pascal**	**Go**	**Haskell**	**Java**	**Clisp**	**Python**	1/3 **C++**
	2.17	6.28	6.37	6.43	6.77	7.70	9 min	
	6.13	6.28	6.93	7.18	7.81	13.18		
spectral-norm	**C++**	**Go**	**Clisp**	**Pascal**	**Haskell**	**Java**	**Python**	1/8 **Pascal**
	0.72	1.43	1.44	**1.44**	1.47	1.55	112.97	
	1.53	5.32	5.44	**11.76**	5.32	7.99	6 min	
mandelbrot	**C++**	**Haskell**	**Go**	**Pascal**	**Java**	**Clisp**	**Python**	1/30 **C++**
	0.84	1.59	3.73	3.89	4.10	4.17	177.35	
	28.50	36.38	6.84	26.52		11.07	11.07	

pidigits								1/60 **Haskell**
	C++	**Pascal**	**Java**	**Go**	**Python**	**Haskell**	**Clisp**	
	0.59	0.73	0.79	0.86	1.16	**1.44**	3.17	
	0.72	14.63	7.65	4.87	4.05	**100.25**	6.09	
fasta								1/5 **C++**
	C++	**Haskell**	**Java**	**Go**	**Clisp**	**Pascal**	**Python**	
	0.77	0.86	1.20	1.27	4.38	5.58	36.90	
	3.36	6.97	4.39	3.78	9.41	5.63	66.17	
k-nucleotide								1/7 **Java**
	C++	**Java**	**Go**	**Haskell**	**Clisp**	**Python**		
	1.96	**4.83**	7.46	11.69	15.12	46.31		
	7.00	**37.00**	9.05	24.50	58.99	78.50		
reverse-complement								1/50 **C++**
	C++	**Go**	**Java**	**Haskell**	**Pascal**	**Clisp**	**Python**	
	0.64	1.34	1.57	3.16	3.69	5.69	6.63	
	34.23	2.11	22.38	5.9	11.51			
binary-trees								1/22 **C++**
	C++	**Pascal**	**Java**	**Haskell**	**Clisp**	**Go**	**Python**	
	0.94	2.04	2.51	4.42	5.44	12.48	44.70	
	22.44	2.06	4.79	15.22	11.43	28.14	138.17	

Таблица 8. Сравнение вариантов используемых средств ЯП с их моделями: Go, Clisp, Haskell

Норма	Показатели					Esc	Комментарий
	Грань	Не ноль	Время		Память		
Go итер	**4453**	2	.19	**.28**	**0668**	- 1	Без прерываний и диагностики переходит в отрицательную область
4Clisp итер	**10000**	0	.01	**.01**	**4404**	stack overflow.	**Грань – 2, время — 28, память – 3**
Go рек	**6**	2	.14	**0.65**	**42808**	Time out	
2Clisp рек	**6**	4	.01	**3.83**	**0524**	Time out	**Грань – 1, время – 1, память – 4**
От и до	**6 - 4453**			**.28 - 10.65**			Грань – 100, время – 5, память – 1
От и до	**36 - 10000**	4 - 20		**.01 - 13.83**	**0524 - 14404**		Грань – 300, время – 1,3, память – 1,5
Разница с моделями на Clisp меньше, чем с вариантами Go							
Haskell fn	**42**	7	.01	**2.95**	**5452**	**Time out**	
3Clisp рек	**35**	9	.01	**3.16**	**636**	Time out	**Грань – 1, время – 1.5, память – 2**
Haskell opt	**3832**	2000	.01	**.01**	**776**	output Limit reached.	Буфер вывода не достаточно универсален
6Clisp line	**3384**	15	.01	**0.01**	**14764**	stack overflow	**Грань – 10, время – 1, память – 3**
Разница с моделями на Clisp меньше, чем с вариантами на Haskell							
От и до	**2 - 33832**	**7 - 32000**		**.01 - 12.95**	**5452 - 5776**		**Грань – 1000, время –1000**
Clisp прог	4056		.01	.36	**0584**	output Limit reached.	
От и до	**5 - 34056**	**2 - '19**		**.01 - 13.16**	**636 - 14764**		**Грань – 1000, время – 1000, память – 1.5**

разработки подходов к измерению вклада программируемых решений в производительность программ. Для каждого ЯП обычно определены списки поддержанных в нём парадигм, предшественники, сфера

влияния. Такие характеристики можно выводить из списков общих семантических систем, оценивая существенную разницу в традиционной прагматике функционально эквивалентных систем.

На этапе экспериментов достаточно выделять из программ ключевые фрагменты, представляющие программируемые решения, и приводить их к нормализованной форме на эталонном языке для измерения. Полученные таким образом результаты измерений пригодны для сравнения с характеристиками других программируемых решений, представленных на разных языках в различных приложениях.

5. План организации измерений для стенда ПРИЗМА

Разработка информационного стенда ПРИЗМА в ИСИ СО РАН предпринята для исследования и решения проблемы анализа и измерения характеристик языков и систем программирования (ЯиСП), влияющих на продуктивность разработки программного обеспечения и производительность программных приложений. Организация данных основана на представлении результатов парадигмально-семантической декомпозиции описаний языков программирования, результаты которой представляются как ряд категорий семантических систем. Каждой категории соответствует сравнительно простая постановка задачи, отражающая шаги изучения языка, приблизительно соответствующие приведённой выше пятиуровневой шкале сложности.

Для более точной оценки ЯиСП, их сравнения и привлечения измеримых характеристик используются комплекты сопоставимых фрагментов или шаблонов программ на оцениваемых ЯП в форме, приспособленной для прямого эксперимента с СП, поддерживающих изучаемые ЯП. В качестве примера небольшое число таких

фрагментов на языке Lisp приведено в описании проекта стенда [12]. Каждый фрагмент предполагается сопровождать результатом измерения показателей его прогона на СП и пояснением схемы применения таких фрагментов в подходящих задачах, соответствующих определённым категориям семантических систем. Таким образом создаётся шкала сопоставимых программ на разных ЯП, при тестировании и измерении характеристик которых на общих платформах можно делать выводы о вкладе языковых конструкций и программируемых решений в производительность программ, используя прямые измерения и модели программ на эталонных ЯП. Использование шкалы сопоставимых постановок задач и методов их решения, включая прагматику СП, даёт критерии для разработки подходов к оптимизации программ и систем программирования, их улучшения и уточнения методов измерения продуктивности программирования, а также вклада программируемых решений в производительность программ.

На этапе экспериментов достаточно выделять из программ ключевые фрагменты, представляющие программируемые решения, приводить их к нормализованной форме на эталонном языке и выполнять измерение и протоколирование характеристик прогона, Полученные таким образом результаты измерений пригодны для сравнения с характеристиками других программируемых решений, представленных на разных языках в различных приложениях. К проведению эксперимента планируется привлекать слушателей ежегодного спецкурса «Парадигмы программирования», поставленного на факультетах ММФ и ФИТ НГУ [26-30]. Для начала разрабатываемую методику можно рассматривать как средство оценки знаний и умений студентов, изучающих программирование. Производственное применение такой методики потребует

реализации синтаксически управляемого конвертера программ на эталонные языки, что несколько сложнее синтаксически управляемых анализаторов. Предполагается, что языки функционального программирования входят в число эталонных языков и будут использоваться для получения нормализованных форм сравниваемых фрагментов программ на разных ЯП. Суммарно стенд ПРИЗМА строится из четырёх модулей: визуализатор результатов парадигмального анализа языка программирования, визуализатор результатов сравнения двух парадигмально проанализированных языков программирования, база данных для хранения линеек постановок эталонных задач и их решений, накопитель измеримых характеристик программ. Примеры данных первых трёх модулей показаны в описании стенда ПРИЗМА, для этих модулей силами студентов ФИТ НГУ выполнен ряд экспериментов по анализу и сравнению языков программирования и разработке визуальных форм для представления полученных результатов. В данном препринте описаны данные модуля «накопитель», эксперименты по которому только начинаются.

Заключение

Ознакомление с описанием ЯП обычно требует от нескольких часов до нескольких дней. Сравнение и измерение производительности компиляторов требует заметно большего времени и значительного количества тестов — нескольких тысяч. Для экспериментального исследования в наши дни можно использовать свои и готовые примеры или шаблоны программ решения типовых задач на базе бенчмарков, предоставляющих доступ к СП для большого числа языков.

При изучении возможностей ЯП полезно учесть образующие его семантические системы, выделяемые с

помощью парадигмальной декомпозиции, позволяющей каждую систему ассоциировать с небольшой эталонной задачей. Есть гипотеза, что число семантических систем существенно меньше мощности пространства языков программирования. Соотношение таких характеристик можно представить как разницу между числом элементов калейдоскопа и мощностью пространства картинок, наблюдаемых через калейдоскоп. Модель реальной задачи или программы её решения может быть сведена к небольшому числу шаблонов решения таких задач, подобных определённым шаблонам эталонного ЯП. Примеры представляют постановку задачи и/или программу её решения, что позволяет чётко оценить возможности ЯП на уровне шаблонов. Известно, что понимание через шаблоны происходит результативнее, чем изучение формальных определений.

Отсутствие метрики для оценки качества программ, зависимости их производительности от программируемых решений влечёт опасный дисбаланс между ростом значимости программных приложений и снижением производительности их новых версий, отчасти маскируемым повышением эксплуатационных характеристик аппаратуры. Препятствует разработке таких метрик слишком высокий темп развития ИТ-индустрии, при котором доминируют быстрые интуитивные или волевые решения, потребность освоения новых возможностей. В результате у разработчиков программных приложений просто нет времени думать далее достижения быстрого успеха во внедрении программ.

В данной статье описаны результаты популярных подходов к оценке производительности программ и удобный бенчмарк для постановки обучения программированию, нацеленного на формирование профессионального умения оценивать и измерять вклад программируемых решений в производительность

программ. При анализе результатов 20-летнего эксперимента «Какой язык быстрее всех» установлен ряд трудно решаемых проблем, что показывает направление дальнейших исследований. Материал по сравнению языков Lisp и Java констатирует возможности повышения продуктивности программирования через овладение языком Lisp, а возможно и собственно функциональным программированием. Бенчмарк «Курсы и задания» предлагает удобную много-языковую платформу для самообучения программированию и поддержке обучения программированию. Изложение опыта оценивания решений олимпиадных задач по программированию показывает не только ряд отчасти проявленных проблем и важных для методики аспектов, но и перспективу формирования научно обоснованной методики измерения продуктивности программирования и производительности программ. Показаны результаты небольшого эксперимента, дающего основания для функционального подхода к разработке методики измерения качества программ. Такой подход, использующий учёт особенностей решаемых задач и ЯП, влияющих на выбор методов их решения, может быть полезен при развитии технологии надёжного и безопасного ПО. В дальнейшем предстоит экспериментальное исследование предложенной методики на более широком наборе задач и ЯП.

Основные выводы и гипотезы:

1. Исследование и разработку метрик производительности программ необходимо проводить одновременно с созданием методик прогнозирования продуктивности разработки программ и/или оценки технологичности программирования.
2. Задачи можно измерять в терминах задач, а программы в терминах программ, и то и другое можно специфицировать с помощью онтологий или моделей.
3. Существуют задачи, трудоёмкость решения

которых известна; среди них можно выбирать эталонные задачи, используемые в качестве учебных.

4. Доступны программы, производительность которых допускает измерение; среди них можно выбрать эталонные программы, допускающие прямые измерения производительности.

5. Первые эксперименты можно выполнить на материале учебных и олимпиадных задач, характеристики которых известны, что позволит методике измерения и её результатам быть понятными и обосновать общую методику измерения вклада программируемых решений в производительность программ.

Создание общей метрики для измерения производительности программных приложений потребует определённой, возможно большой, работы, учитывающей доступные данные, дающие научную основу для правдоподобных оценок. Усложняющим является требование понятности метрики как для программистов, так и для пользователей и менеджеров, но без этого метрика не может получить признание.

Список литературы

1. Массель Л.В. Фрактальный подход к структурированию знаний и примеры его применения. Статья в журнале Онтология проектирования. — 2016. — Т. 6. — № 2(20). — С. 149-161. — DOI: 10.18287/2223-9537-2016-6-2-149-161.
2. Липаев В.В. Очерки истории отечественной программной инженерии 1940-е — 80-е годы. — М.: СИНТЕГ, 2012. https://www.computer-museum.ru/articles/knigi-v-v-lipaeva/605 [электронный ресурс].
3. Лаврищева Е.М., Карпов Л.Е., Томилин А.Н., Подходы к представлению научных знаний в Интернет науке. //Сб. XIX Всероссийский научной конференции «Научный сервис в сети Интернет», Новороссийск, 18-23 сентября 2017. — С. 310-326.
4. Лаврищева Е.М., Карпов Л.Е., Томилин А.Н. Семантические ресурсы для разработки онтологии научной и инженерной предметных областей. //Научный сервис в сети Интернет: труды XVIII Всероссийской научной конференции (19-24 сентября 2016 г. Новороссийск). — М.: ИПМ им. М.В. Келдыша, 2016. — С. 223-239.
5. Липаев В.В. Экономика производства сложных программных продуктов. — М.: СИНТЕГ, 2008.
6. Лаврищева Е.М., Карпов Л.В., Томилин А.Н. Системная поддержка бизнес задач в глобальной информационной сети. // Труды конференции «Научный сервис в сети Интернет - 2015», 21-26 сентября 2015, г. Новоросийск. — С. 193-218.
7. Липаев В.В., Филинов Е.Н. Мобильность программ и данных в открытых информационных системах. — М.: РФФИ. 1997.
8. Лаврищева Е.М. Вопросы объединения разноязыковых

модулей в ОС ЕС. // Программирование. — 1978. — №1. — С. .22—27.
9. Городняя Л.В. О представлении результатов анализа языков и систем программирования. Научный сервис в сети Интернет: труды XX Всероссийской научной конференции (17-22 сентября 2018, г. Новороссийск). — М.: ИПМ им. М.В. Келдыша, 2018.
10. Городняя Л.В. Подход к оценке трудоёмкости программирования. // Научный сервис в сети Интернет: труды XXII Всероссийской научной конференции (21-25 сентября 2020г., онлайн). — М.: ИПМ им. М.В.Келдыша, 2020. — С. 192-209. DOI: https://doi.org/10.20948/abrau-2020-3.
11. Городняя Л.В. От труднорешаемых проблем к парадигмам программирования // XXVI Байкальская Всероссийская конференция с международным участием «Информационные и математические технологии в науке и управлении» (июль 2021, Иркутск).
12. Городняя Л.В., Демидов С.Е., Кириченко М.Д., Панфилов[15] Д.Д. Проект информационного стенда «Призма» для представления измеримых характеристих языков и систем программирования // Информационные и математические технологии в науке и управлении, — 2020. — Вып.1(17).. — . 105-119. ISSN 2413 - 0133.
13. Непомнящий В.А., Аргиров В.С., Белоглазов Д.М., Быстров А.В., Четвертаков Е.А., Чурина Т.Г. Моделирование и верификация коммуникационных протоколов, представленных на языке SDL, с помощью сетей Петри высокого уровня. // Программирование. 2008. Т. 34. № 6. с. 35-49.

[15] Фамилия соавтора «**Панфилов**», приношу соавтору и читателям извинения за ошибку при публикации

14. Liakh, T., Anureev, I., Rozov, A., Garanina, N. & Zyubin, V., Four-Component Model for Dynamic Verification of Process-Oriented Control Software for Cyber-Physical Systems. // SIBIRCON 2019 — International Multi-Conference on Engineering, Computer and Information Sciences, Proceedings. Institute of Electrical and Electronics Engineers Inc.. — P.. 466-471.
15. Zagorulko, Yu.A., Sidorova, E.A., Akhmadeeva, I.R., Sery, A.S. // Approach to automatic population of ontologies of scientific subject domain using lexico-syntactic patterns Journal of Physics: Conference Seriesthis, — 2021. — 2099(1), 012028.
16. Массель Л.В., Массель А.Г., Пестерев Д.В. Технология управления знаниями с использованием онтологий, когнитивных моделей и продукционных экспертных систем // Известия ЮФУ. Технические науки. №4. – С. 140-152. http://izv-tn.tti.sfedu.ru/index.php/izv_tn/article/view/176/151
17. Купер А. Психбольница в руках пациентов. Алан Купер об интерфейсах. //Питер Спб, Библиотека программиста. — 2018. — 384 с.
18. Weinberg G.M. The Psychology of Computer Programming. - Silver Anniversary eBook Edition Kindle Edition. — 2011.— 288 p.
19. Липаев В.В. Человеческие факторы в программной инженерии. Рекомендации и требования к профессиональной квалификации специалистов. // Учебник.— М.: СИНТЕГ, 2009.— 348с.
20. Андреева Т.А. Возможность автоматизации процесса генерирования тестовых наборов // ж. Universum: технические науки. — М., Изд. «МЦНО», 2017. — №8(41). — с. 5-7.
21. Andreyeva, T.A. Automation of correctness checking in education. // A.P. Ershov Informatics Conference (the PSI

Conference Series, 12th edition) Educational Informatics Workshop proceedings. July 2–3, 2019. — Novosibirsk 2019. — P. 6-15.
22. Андреева Т.А. Сборник задач для предолимпиадной подготовки по программированию. — Новосибирск: Изд-во НГУ, 2009.— 226 с.
23. Андреева Т.А. Программирование на языке Pascal. Сер. Основы информационных технологий / — М., 2016. — 277 с. — ISBN 5-9556-0025-6 (переиздание).
24. Gouy, Isaac. The Computer Language Benchmarks Game. Web. https://benchmarksgame-team.pages.debian.net/benchmarksgame/.
25. Erann Gat Lisp as an Alternative to Java https://flownet.com/gat/papers/lisp-java.pdf.
26. Городняя, Л.В. Парадигма программирования: учебное пособие. — Лань, 2019. — 232 с. — 978-5-8114-3565-4.
27. Городняя Л.В. Парадигма программирования: учебное пособие для вузов // 2-е изд., стер.— Лань, 2021. — 232 с. — ISBN: 978-5-8114-6680-1.
28. Городняя Л.В. Парадигма программирования: курс лекций // Новосиб. Гос. Ун-т.- Новосибирск: РИЦ НГУ, 2015. - 206 с.
29. Городняя Л.В. Парадигмы программирования: анализ и сравнение. Сиб. Отделение Рос. Акад. наук, Ин-т систем информатики им. А.П. Ершова. — Новосибирск: Изд-во СО РАН, 2017. — 232 с.
30. Городняя Л.В. Парадигмы программирования. - М.: Интернет-Университет Информационных технологий.— 2006. — URL: http://www.intuit.ru/studies/courses/.
31. Городняя Л.В. Гуманитарные факторы программирования /Сиб. Отделение Рос. Акад. наук, Ин-т систем информатики им. А.П. Ершова. — Новосибирск: Изд-во СО РАН, 2020. — 163с.
32. Городняя Л.В. Функциональное программирование.

Парадигма, модели, методы. / Сиб. Отделение Рос. Акад. наук, Ин-т систем информатики им. А.П. Ершова. г. Новосибирск: Изд-во СО РАН, 2022. — 482 с.

Обозначения программ, использованных в предварительном эксперименте по измерению вклада программируемых решений в производительность программ приведены в таблице 9. Они используются в приложении 2.

Таблица 9.

Функционально эквивалентные программы на разных ЯП (C++, Go, Clisp, Java, Haskell, Pascal, Python, Clojure)

ЯП	*Функция* [16]	*Обозначение*[17]	*Примечание*

C++	//Fibonacci Recursive
	int fibonacci(int n)
	{
	return (n<=2 ? 1 :
	fibonacci(n-1) + fibonacci(n-2));
	}
		C-fibonacci	Самое очевидное решение на базе условного выражения

Go	//Fibonacci Recursive
	func fibr(n int) int {
	if n < 2 { return 1 }
	return fibr(n-2) + fibr(n-1)
	}
		G-fibr	Столь же очевидное решения на базе условного оператора

	//Fibonacci Iterative
	func fibi(n int) int {
	var a, b int = 1, 1
	for i := 0; i < n; i++ {
	a, b = b, a+b
	}
	return a
	}
		G-fibi	Решение на базе оператора цикла for

Clisp	(defun fibonacci (n) (if (< n 3) 1 (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))	**L**-fibonacci	Самое очевидное решения на базе условного выражения на уровне базовой семантики, на один виток рекурсии меньше, чем при сравнении с 2.
	(defun fib-iter (a b count) (Cond ((= 0 count) b) (T (fib-iter (+ a b) a (1- count)))))	**L**-fib-iter	Решение на базе ветвления уровня абстрактного синтаксиса

16 Источник progopedia.ru

17 Используется в Приложении 2 с протоколами измерений

	;Fibonacci Recursive (Defun fib-r (n) (Cond ((< n 2) 1) (T (+ (fib-r (- n 2)) (fib-r (- n 1))))))	**L**-fib-r	Семантический эквивалент C-fibonacci и G- fibr
	(Defun fib (n) (Cond ((< n 3) 1) (T (+ (fib (- n 1)) (fib (- n 2))))))	**L**-fib	На один виток рекурсии меньше, чем в L-fib-r
	(Defun zipWith (Lf) (cons (+ (car Lf) (cadr Lf)) Lf)) (defun line (n fibs) (cond ((< n 3) fibs) (T (line (- n 1) (zipWith fibs)))))	**L**-line	Выделение вспомогательной функции и учёт лишь последних двух чисел.. На большем, чем 3000, переполнение стека, на меньшем нет метки времени
	; Clisp (defun fib-pr(x y count) (prog (i a b) (setq a x) (setq b y) (setq i count) CIRC (Cond ((= 0 i) (return b))) (setq a (+ a b)) (setq b (- a b)) (setq i (1- i)) (go CIRC)))	**L**- fib-pr	Использование механизма императивных средств внутри функционального ЯП
Haskell	fib :: Integer -> Integer fib n | n < 2 = 1 | (n == 2) = 1 | n > 2 = fib (n-1) + fib (n-2)	**H**-fib	Управление выбором ветви по образцу
	fcas :: Integer -> Integer -> Integer -> Integer fcas x a b = case x < 3 of True -> a False -> fcas (x - 1) (a + b) a	**H**-fcas	Использован переключатель и только два последних числа. Краткий вариант динамического программирования
Java	static int fibonacci(int n) { return (n<=2 ? 1 : fibonacci(n-1)	**J**- fibonacci	Условное выражение как в C++

	+ fibonacci(n-2)); }		

Pascal	function fib(n:integer): integer;
	begin
	if (n <= 2) then
	fib := 1
	else
	fib := fib(n-1) + fib(n-2);
	end;
	
	var
	i:integer;
	begin
	for i := 1 to 16 do
	write(fib(i), ', ');
	writeln('...');
	end.
		P-fibonacci	Выделена вспомогательная функция для одного шага и цикл for

Python	def fibonacci(n):
	if n < 3:
	return 1
	else:
	return fibonacci(n - 1)
	+ fibonacci(n - 2)
		Py- fibonacci	Семантический эквивалент L-fibonacci

Clojure	(defn fibr [n] (if (< n 2) 1 (+ (fibr (- n 2)) (fibr (- n 1)))))	**C**l- fibr	Семантический эквивалент L-fib-r

Протоколы измерений

В начале февраля 2022 года, используя сайт «Courses And Assignments https://www.jdoodle.com», проведены измерения процессорного времени и объёма использованной памяти для программ вычисления чисел Фибоначчи, приведённых в Приложении 1, написанных на ЯП C++, Go, Clisp, Java, Haskell, Pascal, Python, Clojure. Регистрировались следующие данные:

Min - обнаружение ненулевого процессорного времени,

20 — показатели на 20 числах,

30 — показатели на 30 числах,

40 — показатели на 40 числах,

max — самое большое число и показатели на нём,

esc - причина прерывания счёта.

Измерения проведены на бенчмарке https://www.jdoodle.com/**.** Результаты измерений приведены в таблице 10.

В протоколе, расположенном в таблице 10 зарегистрированы следующие характеристики:

Грань — величина наибольшего полученного числа.

Не ноль — число с первым ненулевым значением времени.

Время — показатель процессорного времени по завершении вычисления.

Память — максимальный объём памяти для всей серии измерений.

Esc - причина аварийного завершения вычислений.

Таблица 10. *Числа Фибоначчи на разных ЯП*

ЯП	*Обозначение*	*Результаты измерений (показатели)*
C++	**C**-fibonacci	Min 32 20 – нет числа 32 CPU Time: 0.01 sec(s), Memory: 3404 kilobyte(s) 40 CPU Time: 0.57 sec(s), Memory: 3424 kilobyte(s) max 46 CPU Time: 10.27 sec(s), Memory: 3436 kilobyte(s) **esc Timeout**
Go	**G**-fibr	Min 2 CPU Time: 0.20 sec(s), Memory: 40192 kilobyte(s) 20 CPU Time: 0.16 sec(s), Memory: 40364 kilobyte(s) 30 CPU Time: 0.19 sec(s), Memory: 40144 kilobyte(s) 40 CPU Time: 0.79 sec(s), Memory: 38196 kilobyte(s) max 46 CPU Time: 10.65 sec(s), Memory: 40368 kilobyte(s) esc Timeout
	G-fibi	Min 2 CPU Time: 0.19 sec(s), Memory: 38488 kilobyte(s) 20 CPU Time: 0.20 sec(s), Memory: 40096 kilobyte(s) 30 CPU Time: 0.28 sec(s), Memory: 40044 kilobyte(s) 40 CPU Time: 0.21 sec(s), Memory: 40420 kilobyte(s) 1000 CPU Time: 0.20 sec(s), Memory: 40668 kilobyte(s) 2000 CPU Time: 0.20 sec(s), Memory: 40952 kilobyte(s) 4400 CPU Time: 0.19 sec(s), Memory: 40512 kilobyte(s) 30000 - CPU Time: 0.22 sec(s), Memory: 40304 kilobyte(s) max 4453 CPU Time: 0.20 sec(s), Memory: 40668 kilobyte(s) **esc – переход в отрицательные числа**
Clisp	**L**-fibonacci	14 CPU Time: 0.01 sec(s), Memory: 9684 kilobyte(s) 20 CPU Time: 0.01 sec(s), Memory: 9640 kilobyte(s) 30 CPU Time: 0.76 sec(s), Memory: 9620 kilobyte(s) 35 CPU Time: 8.06 sec(s), Memory: 9720 kilobyte(s) max 36 CPU Time: 13.83 sec(s), Memory: 9704 kilobyte(s) esc Timeout
	L-fib-r	Min 19 CPU Time: 0.01 sec(s), Memory: 9728 kilobyte(s) 20 CPU Time: 0.01 sec(s), Memory: 9780 kilobyte(s) 30 CPU Time: 1.36 sec(s), Memory: 9640 kilobyte(s) 40 - no max 35 CPU Time: 13.06 sec(s), Memory: 9636 kilobyte(s) esc Timeout
	L-fib	4. 14 CPU Time: 0.01 sec(s), Memory: 9684 kilobyte(s) 5. 20 CPU Time: 0.01 sec(s), Memory: 9640 kilobyte(s) 30 CPU Time: 0.76 sec(s), Memory: 9620 kilobyte(s) 35 CPU Time: 8.06 sec(s), Memory: 9720 kilobyte(s) макс 36 CPU Time: 13.83 sec(s), Memory: 9704 kilobyte(s) esc Timeout CPU Time: 0.10 sec(s), Memory: 10524 kilobyte(s)
	L- fib-pr	Min 2 CPU Time: 0.01 sec(s), Memory: 9796 kilobyte(s) 20 CPU Time: 0.01 sec(s), Memory: 9696 kilobyte(s) 30 CPU Time: 0.00 sec(s), Memory: 9764 kilobyte(s) 40 CPU Time: 0.00 sec(s), Memory: 9796 kilobyte(s) 4000 CPU Time: 0.01 sec(s), Memory: 10336 kilobyte(s) 10000 CPU Time: 0.04 sec(s), Memory: 10448 kilobyte(s) 20000 CPU Time: 0.17 sec(s), Memory: 10628 kilobyte(s)

		30000 CPU Time: 0.26 sec(s), Memory: 10428 kilobyte(s) 32000 CPU Time: 0.30 sec(s), Memory: 10460 kilobyte(s) 33000 CPU Time: 0.33 sec(s), Memory: 10528 kilobyte(s) 34000 CPU Time: 0.33 sec(s), Memory: 10592 kilobyte(s) 34050 CPU Time: 0.36 sec(s), Memory: 10484 kilobyte(s) max 34056 CPU Time: 0.35 sec(s), Memory: 10568 kilobyte(s) CPU Time: 0.34 sec(s), Memory: 10584 kilobyte(s) esc output Limit reached. 170060 - хватит
	L-line	Min 15 - CPU Time: 0.01 sec(s), Memory: 9760 kilobyte(s) 20 - CPU Time: 0.00 sec(s), Memory: 9768 kilobyte(s) 30 - CPU Time: 0.01 sec(s), Memory: 9764 kilobyte(s) 40 - CPU Time: 0.00 sec(s), Memory: 9708 kilobyte(s) max – 3384 CPU Time: 0.01 sec(s), Memory: 14764 kilobyte(s) esc stack overflow
	L-fib-loop	20 CPU Time: 0.01 sec(s), Memory: 9752 kilobyte(s) 30 CPU Time: 1.23 sec(s), Memory: 9724 kilobyte(s) 35 CPU Time: 12.40 sec(s), Memory: 9748 kilobyte(s) esc Timeout
Java	**J-** fibonacci	Min 2 CPU Time: 0.07 sec(s), Memory: 32016 kilobyte(s) 20 CPU Time: 0.09 sec(s), Memory: 31500 kilobyte(s) 30 CPU Time: 0.07 sec(s), Memory: 31848 kilobyte(s) 40 CPU Time: 0.37 sec(s), Memory: 31528 kilobyte(s) 47 CPU Time: 8.76 sec(s), Memory: 31408 kilobyte(s) max 48 CPU Time: 14.24 sec(s), Memory: 32096 kilobyte(s) esc Timeout
Haskell	**H**-fib	Min 27 20 - нет 27 CPU Time: 0.01 sec(s), Memory: 5100 kilobyte(s) 30 CPU Time: 0.04 sec(s), Memory: 5196 kilobyte(s) 40 CPU Time: 4.99 sec(s), Memory: 5340 kilobyte(s) max 42 CPU Time: 12.95 sec(s), Memory: 5452 kilobyte(s) esc timeout
	H-fcas	Min 32000 CPU Time: 0.01 sec(s), Memory: 5712 kilobyte(s) 20 - нет 30 - нет max 33832 CPU Time: 0.01 sec(s), Memory: 5776 kilobyte(s) esc output Limit reached.
Pascal	**P**-fibonaci	29 CPU Time: 0.01 sec(s), Memory: 352 kilobyte(s) 30 CPU Time: 0.01 sec(s), Memory: 344 kilobyte(s) 40 CPU Time: 1.65 sec(s), Memory: 352 kilobyte(s) 43 CPU Time: 7.28 sec(s), Memory: 348 kilobyte(s) 44 CPU Time: 11.00 sec(s), Memory: 412 kilobyte(s) esc Timeout
Python	**Py-** fibonacci	Min 3 CPU Time: 0.01 sec(s), Memory: 7884 kilobyte(s) 20 PU Time: 0.01 sec(s), Memory: 7804 kilobyte(s) 30 CPU Time: 0.26 sec(s), Memory: 7724 kilobyte(s) 38 CPU Time: 10.41 sec(s), Memory: 7756 kilobyte(s) 40 - нет max 39 esc timeout

Clojure	**Cl-**fibr	2: CPU Time: 2.96 sec(s), Memory: 124424 kilobyte(s) 20: CPU Time: 2.9*4 sec(s), Memory: 122956 kilobyte(s) 30: CPU Time: 3.02 sec(s), Memory: 125804 kilobyte(s) 40: CPU Time: 4.17 sec(s), Memory: 129068 kilobyte(s) 43: CPU Time: 8.74 sec(s), Memory: 236728 kilobyte(s) 44 CPU Time: 13.64 sec(s), Memory: 235828 kilobyte(s) 2: CPU Time: 3.33 sec(s), Memory: 120464 kilobyte(s) 2 CPU Time: 3.54 sec(s), Memory: 117640 kilobyte(s) esc timeout

Утверждено к печати в электронном виде
Редакционным советом Института систем информатики СО РАН

Т.А. Андреева, Л.В. Городняя

ФУНКЦИОНАЛЬНЫЙ ПОДХОД К ИЗМЕРЕНИЮ ВКЛАДА ПРОГРАММИРУЕМЫХ РЕШЕНИЙ В ПРОИЗВОДИТЕЛЬНОСТЬ ПРОГРАММ

Препринт
187

Редактор Т.М. Бульонкова
Рецензент Д.С. Мигинский