История развития предметно-ориентированных языков программирования

Власовских А. С.

Кафедра автоматики и вычислительной техники
Санкт-Петербургский государственный политехнический университет
vlasovskikh@aivt.ftk.spbstu.ru

Эта статья была зачтена как реферат по истории науки в рамках кандидатского экзамена по истории философии науки и техники на кафедре философии Санкт-Петербургского государственного политехнического университета.

Введение

Предметно-ориентированные языки программирования (Domain Specific Languages, DSL) — языки программирования, предоставляющие при помощи специальной нотации и механизмов абстракции выразительную силу, сконцентрированную на определённой предметной области и обычно ограниченную этой областью. Языки DSL широко используются сообществом программистов. На данный момент они считаются общепризнанным направлением языков программирования. За несколько десятилетий развития этих языков сформировались методы и способы их разработки, стали известны многие варианты использования их на практике. Тем не менее, работы по истории языков программирования не рассматривают языки DSL как единое направление, делая акцент на более крупные направления языков общего назначения. Существуют обобщающие работы по DSL, но их предметом являются вопросы использования DSL в конкретных областях или вопросы технологии разработки DSL.

Целью данного реферата является формирование исторической картины развития DSL с указанием происхождения и авторов нововведений, анализом влияния отдельных групп исследователей и специалистов на развитие языков этого направления, становления основных методов разработки DSL.

Субъективная мотивация выбора темы заключается в полезности результатов исследования истории развития языков DSL для курса «Языки программирования», читаемого автором реферата на кафедре автоматики и вычислительной техники. Языкам DSL в курсе отведена заметная роль для пояснения метода металингвистической абстракции и принципов проектирования языков. Подача студентам материала в исторической перспективе с указанием научных направлений, групп, конкретных имён и работ имеет несомненные достоинства. Ещё одним стимулом к выбору темы послужила её связь с областью диссертационного исследования автора, связанного с методами проектирования ПО на основе формализации элементов архитектуры ПО.

Реферат имеет следующую структуру. Он состоит из трёх разделов, описывающих отдельные этапы развития DSL, частично пересекавшиеся по временным рамкам. В *разд. 1* описывается рассматривается история развития

языка Lisp и его диалектов на протяжении 1960-1970-х гг. *Разд. 2* посвящён так называемым «маленьким языкам», получившим развитие в 1970-1980-х гг. в рамках операционной системы Unix. Языки сценариев, получившие активное развитие в 1980-1990-е гг., рассматриваются в *разд. 3*.

Несколько слов о самом реферате. Оригинал этого документа был создан с использованием нескольких языков DSL обработки текстов и программных инструментов для работы с ними. *Язык XHTML* — это язык структурной разметки гипертекста. *XML* — это подмножество обобщённого языка разметки SGML, позволяющее определять новые языки разметки в рамках стандартного синтаксиса. *DocBook* — язык структурной разметки технической документации на основе XML. *XSLT* — язык преобразований документов XML при помощи функциональных операторов. *XPath* — язык для выбора подмножеств элементов XML. *CSS* — язык каскадных таблиц стилей, позволяющий задавать правила отображения элементов XML на различных типах устройств.

1. Lisp и его диалекты

Начальный этап развития предметно-ориентированных языков программирования (Domain Specific Language, DSL) тесно связан с языком *Lisp*, одним из первых языков программирования. Он был разработан в 1958 г. Джоном Маккарти (John MacCarthy) для решения задач в области искусственного интеллекта (ИИ). На Дартмутской научной конференции 1956 г. Маккарти и другие исследователи обсудили аспекты этой новой в то время области [22]. Исследователи пришли к выводу, что многие задачи ИИ, такие как обработка естественных языков, логический вывод, самообучение, можно будет решить при помощи символьных вычислений. Маккарти начал разработку языка Lisp для проведения вычислений такого типа при помощи обработки списков. Будучи языком общего назначения, Lisp стал основой для целого семейства диалектов и специализированных языков, многие из которых могут быть отнесены к категории DSL. Не менее важно и то, что благодаря Lisp, развились важные концепции программирования: функциональное программирование и металингвистическая абстракция.

Рассмотрим особенности языка Lisp в исторической перспективе, акцентируя внимание на их значении для DSL. Lisp значительно отличался от большинства языков программирования 1950-1970 гг. В статье о ранних годах Lisp [23] Маккарти описывает факторы успеха языка. В основу Lisp была положена модель вычислений на основе *λ-исчисления*, формальной системы определения и применения рекурсивных функций. Такие языки, использующие понятие «функция» в математической форме, называются *функциональными*. В отличие от Lisp, другие широко используемые языки того времени (например, Fortran и Algol) были тесно связаны с аппаратной архитектурой компьютеров. Долгое время единственным видом абстракции в этих языках, именуемых *процедурными*, была весьма простая процедурная абстракция инструкций архитектуры фон Неймана. Оглядываясь назад, эксперт по Lisp Дик Гэбриэл (Dick Gabriel) в интервью, данном в 2007 г. радиопередаче «Software Engineering Radio» [12], говорит, что функциональная природа Lisp позволила программистам пойти по пути процедурной абстракции намного дальше за счёт активного использования приёмов функционального программирования. К ним

относятся, например, *функции высших порядков*, принимающие в качестве аргументов другие функции. Такие функции позволили добиться значительной выразительной силы и универсальности списковых операций и во многом определили успех Lisp. Не менее важным было и умелое совмещение функционального и процедурного подходов, необходимое при решении практических задач.

В своей тьюринговской лекции 1978 г. [7] Джон Бэкус (John Backus) рассуждает о функциональном стиле программирования в целом и отмечает следующие его преимущества, позволившие ему закрепиться в концептуально сложных областях. Свойство *функциональной чистоты*, т. е. отсутствие побочных эффектов при вычислении, позволило относительно просто доказывать свойства фрагментов программ, в то время как для вывода свойств процедурных языков потребовалось создание специальных методов описания семантики программ. Процедурные языки имеют естественное ограничение в способе выражения вычислений из-за цикла фон Неймана выборки инструкций и данных из памяти. Функциональные языки не заставляют программистов концентрироваться на ячейках памяти и пересылках, позволяя им сосредоточиться на концептуальной стороне решаемых задач.

Наибольшее влияние на развитие языков на основе Lisp оказали исследования, проводимые в 1960-1970-е гг. в лаборатории Project MAC Массачусетского технологического института (MIT). В конце 1950-х гг. Маккарти участвовал в научной деятельности в Дартмутского колледжа, Принстонского и Стэнфордского университетов и MIT. В 1963 г. он помог Марвину Минскому (Marvin Minsky) организовать в MIT лабораторию Project MAC для исследований ИИ. Вскоре Маккарти оставил лабораторию Минского и стал профессором в Стэнфорде, где организовал свою лабораторию Stanford AI Lab. В рамках лаборатории Project MAC Минский и его коллеги проводили исследования в областях ИИ (автоматическое доказательство теорем, обработка естественных языков), языков программирования и операционных систем.

Исследователями MIT было продолжено развитие языков на основе Lisp. В 1966 г. был создан язык MacLisp для машин DEC PDP-6, собственный диалект Lisp лаборатории Project MAC, один из наиболее известных диалектов Lisp того времени. В 1967 г. в рамках работы над диссертацией [15] Карлом Хьюитом (Carl

Hewitt) под руководством Минского был создан специализированный язык Planner для автоматического доказательства теорем, расширяющий Lisp. В основе такого расширения, помимо функционального стиля Lisp, лежал ряд особенностей, под воздействием которых начал формироваться способ металингвистической абстракции. Одной такой особенностью Lisp являлась его необычная синтаксическая структура. Lisp имеет очень минималистичный синтаксис: примитивные типы данных, пары для комбинирования данных, вызовы процедур и несколько особых форм для конструкций управления вычислениями. При этом все составные единицы образуются при помощи списков. Благодаря особому синтаксису Lisp, определённые программистом процедуры неотличимы от встроенных его интерпретатор, что делает делает возможным встраивание синтаксиса нового языка.

Ещё одной особенностью Lisp является то, что программа на нём представляет собой абстрактное синтаксическое дерево (для других языков его строят из исходного кода алгоритмами синтаксического разбора). Это дерево составлено из списков, представляющих как процедуры, так и данные. Создатель динамического объектно-ориентированного языка Smalltalk Алан Кэй (Alan Kay) в [18] охарактеризовал Lisp как гомоиконный язык. В *гомоиконном языке* код и данные имеют одинаковое представление. При этом код может трактоваться как данные, а даные — как код. В Lisp код и данные представляются как объединение атомов в списки, код обрабатывается как данные при помощи процедуры закавычивания `quote`, данные трактуются как код при помощи процедуры вычисления `eval`. Такое свойство использовалось в программах ИИ лаборатории Project MAC для написания программ, которые сами пишут программы, т. е. генерируют новый программный код. В рамках диссертации [33] Терри Винограда (Terry Winograd), защищённой в 1971 г. также под руководством Минского, была разработана программа ИИ SHRDLU, получившая известность. В этой программе, написанной на языке MicroPlanner (подмножестве языка Planner) использовались приёмы логического программирования Planner для разбора диалога с пользователем на естественном языке и приёмы генерации кода.

Несмотря на большую гибкость синтаксиса Lisp, использование применения только функций как единственного средства абстракции процедур налагало

определённые ограничения на выразительность программ. Так, использование в Lisp аппликативного порядка вычисления аргументов функций делает невозможной реализацию условных форм типа `cond` и `if` на основе функций. Использование нестрогих стратегий вычислений доступно в Lisp благодаря *макросам*. Многие подходы по использованию макросов Lisp обобщены в книге [14] Пола Грэма (Paul Graham). Макросы Lisp, в отличие от широко известных макросов языка C, функционируют на уровне синтаксической структуры языка. Многие расширения Lisp, а также возможности, включённые в более новые диалекты Lisp, опираются на механизм макросов. Например, система CLOS для языка Common Lisp представляет собой встраиваемый в Lisp объектно-ориентированный язык. Для добавления объектно-ориентированных возможностей в тот же C потребовалась разработка нового компилятора и, фактически, несовместимого языка C++, в то время как для Lisp подобные системы разрабатываются в виде библиотек, способных выполняться в любом интерпретаторе. Минималистичная и выразительная синтаксическая структура и макросы открывают дополнительные возможности эволюции языка [29, 31]. Эволюционный подход к развитию языка имеет большие преимущества по сравнению с созданием полной спецификации языка до начала его применения. Создатели диалектов Lisp показали преимущества эволюционного подхода, широко признаваемые теперь компьютерным сообществом.

В середине 1970-х гг. сотрудники лаборатории MIT AI Lab (образованной в ходе реструктуризации Project MAC) Джеральд Сассман (Gerald J. Sussman) и Гай Стил (Guy L. Steele Jr.) занимались исследованиями языков и интерпретаторов семейства Lisp. Они разработали язык Scheme, диалект Lisp, отличавшийся б*о*льшим соответствием теории λ-исчисления за счёт использования лексического обзора имён переменных и замыканий как значений функций [32]. К концу 1970-х гг. Сассман совместно с Харольдом Абельсоном (Harold Abelson) стали преподавать Scheme студентам младших курсов в качестве базового языка программирования. Они систематизировали накопленный к тому времени опыт применения Lisp и построили свой курс на новой методической основе. В качестве базы освоения языка ими были выделены примитивы, средства композиции и средства абстракции. В качестве способов абстракции в их курсе фигурировали абстракция процедур, абстракция

данных и металингвистическая абстракция. По методу *металингвистической абстракции*, для выделения значимых аспектов задачи и ухода от несущественных деталей нужно создать язык, на котором задача выражается естественным образом. Такой язык выражает не одну конкретную задачу, а класс схожих задач в определённой предметной области в принятых в ней терминах. Этот метод подытожил идеи, применявшиеся в специализированных языках 1960-1970-х гг. и стал основой для направления предметно-ориентированных языков DSL.

В 1985 г. по этому ставшему очень популярным курсу Абельсон и Сассман написали знаменитую книгу «Структура и интерпретация компьютерных программ» [1], переизданную впоследствие на многих языках. В курсе Абельсона и Сассмана присутствовало много материала по металингвистической абстракции. Рассматривались вопросы создания новых специализированных языков, как встраиваемых в базовый язык, так и интерпретируемых. Так, например, рассматривались язык для символьного дифференцирования, язык изображений Хендерсона, упрощённый вариант языка Prolog, метациклический интерпретатор Scheme. В этих примерах успешного применения металингвистической абстракции авторы придавали большое значение изучению студентами рассмотренных выше свойств Lisp, делающих его основой для построения специализированных языков. Курс по этой книге в настоящее время читается во многих вузах разных стран мира. Подход к преподаванию функционального языка как базового считается одним из основных подходов в «Учебном плане компьютерных наук» ACM [9], в котором отмечается роль Абельсона и Сассмана в становлении этого направления преподавания языков.

К середине 1980-х гг. язык Lisp (его диалекты) считался одним из самых перспективных языков. Как вспоминает Гэбриэл [12], этому содействовала популярность таких приложений ИИ как *экспертные системы* в деловой среде. Многие исследователи ИИ в то время основали или устроились на работу в коммерческие компании. Lisp постепенно коммерциализировался, статьи о нём несколько раз появлялись в деловом журнале «Forbes». Однако к концу 1980-х г. началась так называемая «зима ИИ». Гэбриэл говорит, что компании ИИ тех лет были хороши, рассуждая о том, что они *могут* сделать для бизнеса. Но как дело начало касаться *предъявления* конкретных результатов, компании оказывались

несостоятельными. Оказавшись в такой ситуации многие компании заявляли, что всё оттого, что они используют Lisp. Он, по их словам, требовал слишком много памяти, был очень медленным и т. д. Так что доверие к этому языку начало падать. В 1990-е гг. основными языками стали развивавшиеся на протяжении 1970-1980-х C (позже C++) и Java. Очень многие исследователи также сместили свой акцент: к примеру, Стил после разработки Scheme и Common Lisp перешёл в фирму Sun Microsystems и занялся развитием языка Java. С конца 1970-х гг. всё более заметную роль в развитии DSL начинают играть инженеры и исследователи из сообщества, образовавшегося вокруг операционной системы Unix.

Развитие языка Lisp способствовало формированию метода металингвистической абстракции и популяризовало приёмы функционального программирования. Специализированные языки на основе Lisp стали первыми примерами встраиваемых DSL, использовавшими особый синтаксис Lisp, гомоиконность, расширение языка через макросы. Они продемонстрировали такие их преимущества, как выражение проблемы и возможность валидации на уровне абстракции предметной области, фиксация в языке знаний о предметной области. Тем не менее, развитие Lisp происходило в основном в академической среде и не затрагивало широкий круг инженеров. Далее будут рассмотрены т. н. маленькие языки, благодаря которым использование и разработка языков DSL стали повсеместными.

2. Маленькие языки Unix

Языки DSL получили признание среди масс пользователей и разработчиков благодаря программным инструментам операционной системы (ОС) Unix. Эта ОС начала разрабатываться в начале 1970-х гг. в лабораториях Bell Labs корпорации AT&T. На данный момент существует большое количество вариантов Unix, включая такие популярные ОС, как GNU/Linux, Solaris, Mac OS X, FreeBSD. Основные разработчики Unix Кен Томпсон (Ken Tompson) и Деннис Ритчи (Dennis Ritchie) являются лауреатами премии Тьюринга, а их ОС во многом сформировала образ современных информационных технологий.

Описывая историю Unix [4], Эрик Рэймонд (Eric Raymond) приводит несколько неформальных правил её разработчиков, сложившихся в ходе развития ОС. Согласно этим правилам, следует, например:

- писать программы, делающие одну вещь и делающие её хорошо;
- писать программы, которые могут быть соединены с другими через простые интерфейсы;
- выражать знания в данных, а не в программном коде;
- экономить время программиста, а не машинное время.

Одним из наиболее ярких примеров, показавших успешность этих правил при разработке ПО, являются маленькие языки Unix. *Маленькие языки* — языки небольших программных утилит, создаваемые для решения узкого класса задач обработки данных, и часто применяемые совместно для решения более широкой задачи. Этот термин стал общеупотребительным названием языков этой группы после одноимённой статьи [8] Джона Бентли (Jon Bentley) 1986 г., в которой он обобщил практику использования и разработки таких языков. Помимо упомянутых выше правил, которым следует большинство маленьких языков, подчёркивают также простоту поддержки и изменений, вносимых в код на таких языках [10], а также независимость внешних маленьких языков от платформы реализации.

Начало этому направлению языков положила разработка группы программ для т. н. «рабочего места составителя документов». В начале 1970-х гг. одной из важных прикладных задач, решаемых на ОС Unix, была подготовка документации. В основе системы подготовки документов лежала программа

форматирования troff. Однако довольно скоро было обнаружено, что есть много видов информации, которую сложно размечать на языке troff. Разработчики Unix создали ряд языков и соответствующих утилит, призванных упростить и снизить избыточность описания такой информации. Одними из ранних языков для troff были eqn для оформления математических формул, tbl для таблиц и pic для диаграмм. Среди других стоит упомянуть grn для графики, refer и bib для форматирования библиографии, chem для химических формул. Остановимся подробнее на eqn как характерном представителе этой группы.

Язык *eqn* [19] был создан Брайаном Керниганом (Brian Kernighan) и представлен в 1975 г. Несмотря на то, что язык развивался на протяжении нескольких лет, Керниган замечает, что основную работу по созданию языка и его транслятора можно оценить в один человеко-месяц. Язык был создан для набора математических формул как математиками, так и не особо опытными пользователями типа секретарей и наборщиков. Он имеет простую текстовую структуру и включается в разметку troff при помощи ограничителей в виде управляющих слов troff. Такой подход позволяет реализовать транслятор eqn как препроцессор troff и использовать его в конвейере команд Unix. *Конвейер* как принцип разделения языков (и их областей) между трансляторами стал одним из шаблонов проектирования DSL. Простой универсальный интерфейс конвейера команд с единственным методом приёма и передачи и текстовым форматом данных способствовал развитию направления простых маленьких приложений в ОС Unix.

Транслятор eqn, одноимённая утилита, была разработана Керниганом при помощи генератора синтаксических анализаторов. Такой подход часто использовался и для других маленьких языков. Основа для создания анализаторов при помощи генерации кода была заложена работами по алгоритмам лексического и синтаксического анализа [2] Альфреда Ахо (Alfred Aho), Джеффри Ульмана (Jeffrey Ullman) и др. в 1960-1970-х гг. Благодаря генераторам анализаторов, удалось упростить разработку новых языков, что значительно снизило барьер сложности в реализации трансляторов языков. Наиболее популярными генераторами лексических и синтаксических анализаторов с 1970-х гг. по настоящее время являются, соответственно, варианты программ Lex и Yacc. Эти программы использовали собственные DSL

для задания лексических и грамматических конструкций.

Lex [21] был разработан Майком Леском (Mike Lesk) и Эриком Шмидтом (Eric Schmidt) в 1975 г., *Yacc* [17] — в том же году Стефеном Джонсоном (Stephen Johnson). Эти утилиты расширяют язык C при помощи DSL описания лексики и синтаксиса. Реализация DSL *поверх языка*, является ещё одним вариантом создания DSL, в котором удаётся использовать синтаксическую основу несущего языка, не испытывая при этом почти никаких ограничений в реализации DSL. В случае с макросами Lisp это не так, поскольку DSL ограничен синтаксисом списковых выражений Lisp. Lex основывается на языке регулярных выражений, при помощи комбинации которых выражаются правила разбиения потока символов на токены. Yacc позволяет задать грамматику языка при помощи нотации, близкой к форме Бэкуса-Наура. И регулярные выражения, и форма Бэкуса-Наура являются знакомыми средствами для создателей формальных языков с 1960-х гг. Их использование в Lex и Yacc стало очередным примером удобства DSL, приближенных к языкам специалистов предметной области.

Другими примерами языков для автоматизации разработки ПО являются утилиты make и awk. Программа *make* была создана Стюартом Фелдманом (Stuart Feldman) в 1977 г. [13] для автоматизации сборки программ. Язык системы make является декларативным, основанным на правилах. Граф зависимостей между исходными файлами и целями выводится программой автоматически, исходя их фактов, описанных на языке make. Смещение акцента с алгоритмов обработки данных на структуры данных в сторону *повышения декларативности* облегчает создание и восприятие программ человеком. Так, один из разработчиков ОС Plan 9 Роберт Пайк (Robert Pike) в [26] отмечает: «Данные доминируют. Если вы выбрали правильные структуры данных и организовали всё правильно, то алгоритмы почти всегда будут очевидными. Структуры данных, а не алгоритмы, являются более значимыми для программирования».

Язык *awk* [6] был создан Альфредом Ахо, Питером Вейнбергером (Peter Weinberger) и Брайаном Керниганом в 1977 г. и развивался на протяжении 1980-х гг. Этот язык был предназначен для обработки текстовых файлов при помощи сопоставления по образцу. В качестве примеров решаемых с его помощью задач

можно назвать генерацию отчётов по файлам протоколов и создание кросс-ссылок в текстах. Ещё одно применение awk заключалось в использовании его как средства для реализации трансляторов весьма простых языков наподобие конфигурационных файлов. Подобные языки часто планировались таким образом, чтобы для их разбора требовалась только *лексическая обработка* без построения синтаксического дерева. Для сопоставления по образцу в языке активно использовались регулярные выражения. В отличие от make, язык awk является императивным и достаточно общим. Полнота по Тьюрингу, являющаяся для многих маленьких языков скорее дополнительной особенностью или уловкой, активно использовалась в awk для выражения вычислений. В языке awk присутствовали классические конструкции управления потоком вычислений: условия и циклы.

Несмотря на то, что в течение 1980-х гг. awk приобрёл большую популярность, в 1990-е гг. его вытеснили т. н. *языки сценариев*, речь о которых пойдёт ниже. В частности, язык Perl, фактически, заместил awk. Рэймонд в [4] говорит об awk как о негативном примере проектирования DSL. awk создавался как небольшой специализированный язык для генерации отчётов. Однако, его подмножество для описания действия весьма некомпактно для маленького языка, а общая структура на основе сопоставления по образцу, закреплённая в языке, делает его неприменимым в общем случае. Тем не менее, awk продолжает использоваться до сих пор и является элементом культурной традиции Unix.

В 1990-е гг. маленькие языки стали предметом обобщающих исследований. Такой характер имеют, например, упоминавшиеся выше работы [8, 10, 4]. Заметными работами также стали аннотированная библиография [11] и статья [30], в которой описывается ряд шаблонов проектирования DSL. В конце 1990-х гг. состоялись и первые специализированные конференции по DSL. Они были проведены в 1997 и 1999 гг. ассоциацией USENIX. На них были представлены тезисы как по маленьким языкам, так и по многим DSL в специальных областях типа синтеза цифровых схем, а также доклады по языкам сценариев, рассматриваемым ниже. Маленькие языки оказали большое влияние на развитие области DSL. Они стали значительным этапом развития направления внешних DSL. Благодаря этим языкам развились такие приёмы и средства, как конвейер языковых процессоров, реализация DSL поверх несущего языка, лексическая

обработка DSL, а также регулярные выражения и генераторы синтаксических анализаторов. Помимо этого, следует подчеркнуть значительное влияние маленьких языков как на среду Unix-подобных ОС, так и на компьютерную среду в общем. Несмотря на то, что и в 1990-е гг. продолжали создаваться языки DSL, подобные оригинальным маленьким языкам, их развитие происходило, в основном, на основе уже накопленного опыта. В свою очередь, новым активно развивающимся направлением стали языки сценариев.

3. Языки сценариев

Сообщество вокруг ОС Unix стало средой развития сразу нескольких направлений программирования. Помимо уже упоминавшихся маленьких языков, такими направлениями были системное программирование (Unix теснейшим образом связана с языком C) и командные оболочки ОС. Из командных оболочек под влиянием некоторых маленьких языков в 1980-х гг. возникло направление языков сценариев, речь о которых пойдёт в данном разделе. *Языки сценариев* предназначены для «склеивания» приложений. Они подразумевают наличие достаточно мощных компонентов, написанных на других языках, позволяя соединять эти компоненты для решения комплексных задач. Языки сценариев в конце 1980-х гг. заняли нишу между специализированными и далеко не всегда обладающими свойствами Тьюринг-полноты маленькими языками и языками программирования общего назначения. К концу 1990-х гг. языки этой группы стали постепенно отдаляться от DSL, а некоторые из них стали языками общего назначения.

Командные оболочки использовались в Unix не только для интерактивного ввода пользователем команд, но и для исполнения наборов команд, известных как *сценарии*. Одной из первых командных оболочек Unix, включавшей конструкции управления вычислениями, стала оболочка Bourne Shell, разработанная в 1977 г. Стивеном Борном (Stephen Bourne). Создатель одной из последующих оболочек ksh Дэвид Корн (David Korn) в работе [20] отмечает, что оболочка Борна наряду с оболочкой Мэши были первыми оболочками, в которых конструкции условий и переходов были встроены в саму оболочку, а не реализовывались как внешние программы. Эти оболочки были сравнимы по возможностям, но при решении о включении одной из них в ОС Unix Version 7 выбор комитета разработчиков пал на Bourne Shell. Корн пишет, что благодаря включению управляющих конструкций в оболочку стало возможным создание достаточно эффективных командных файлов, не требующих длительных операций порождения внешних процессов при каждой проверке условий. Для программистов оболочка Bourne Shell и другие подобные оболочки начиная с конца 1970-х гг. стала превращаться в интерпретатор языка программирования. В своей книге [3] 1984 г. Керниган и Пайк впервые использовали в учебном

руководстве Bourne Shell как язык программирования.

Основной причиной возникновения интереса к программированию на языках оболочек стала возможность связывать в вычислениях уже готовые программы, используя при этом намного меньшее число строк кода по сравнению с программированием на языках, подобных C или Pascal. Из сценариев оболочки программисты могли вызывать любые программы за одну строку кода. При этом входные и выходные данные могли быть обработаны в сценарии как текстовые строки, достаточно универсальный «нетипизированный» формат данных, удобный для обработки маленькими языками вроде awk или grep для регулярных выражений или bc для вычисления арифметических выражений. В то же время, в программе на C для этого потребовалось бы несколько системных вызовов для создания внешнего процесса программы, выделения динамической памяти, несколько переменных для хранения промежуточных значений, вызовы процедур разбора строк и т. п.

Несмотря на то, что языки сценариев позволяли создавать более короткие программы по сравнению с языками общего назначения, программы на последних были в сотни раз более производительными в связи с тем, что давали программистам больший контроль над машиной и компилировались в машинные инструкции. Языки сценариев же являются интерпретируемыми, что позволяет не тратить время на частую перекомпиляцию при разработке программ, но значительно замедляет их выполнение. Тем не менее, благодаря тенденции утроения производительности каждые два года (актуальной до сих пор и получившей название закона Мура) к концу 1980-х гг. компьютеры были уже достаточно быстрыми, а их вычислительный ресурс — достаточно дешёвым, чтобы ценить время работы программистов намного больше, чем машинное время [4]. Программирование на очень высоком уровне стало набирать популярность. Однако использование языка командной оболочки и совокупности маленьких языков для написания сценариев имело свои отрицательные стороны, что привело к созданию в конце 1980-х — начале 1990-х гг. сразу нескольких «настоящих» языков сценариев, уже не связанных с командной оболочкой.

Первым таким языком стал *Perl*, разработанный Ларри Уоллом (Larry Wall) в 1987 г. [5]. Этот язык предназначался для обработки текстовых файлов и генерации отчётов, что часто требуется системным администраторам. На

комбинации языка программной оболочки, awk и sed такие задачи решались не особо эффективно. Несмотря на прогресс вычислительной техники, сценарии оболочки всё ещё требовали очень много времени на выполнение, что особо сказывалось при больших объёмах обрабатываемых данных. Также зачастую требовались нетривиальные алгоритмы обработки, которые было сложно выразить в awk или оболочке. В языке Perl содержались встроенные средства работы с регулярными выражениями, традиционные конструкции типа ветвлений, циклов, процедур, большое количество процедур обработки строк и процедуры вызова внешних программ. Всё это (наряду со свободным доступом к исходному коду) сделало Perl очень популярным языком в области системного администрирования. Perl демонстрирует несколько характерных черт языков сценариев. Он является интерпретируемым, динамически типизируемым языком, что экономит время программиста при разработке. Язык позволяет обращаться ко внешним программам, анализировать текстовый вывод, объединять программы для выполнения высокоуровневых задач. Как отмечается в [4], Perl является очень практичным языком: эффективным, простым в использовании, в противовес элегантности и минималистичности. В связи с этим он известен как один из наиболее некрасивых языков, но его практичность позволяет оставаться одним из самых популярных среди всех языков программирования.

Ещё одним языком, созданным в то время, стал *Tcl* Джона Остерхота (John Ousterhout) [24]. Tcl (Tool Command Language) был создан для решения проблемы встраивания командного языка для управления и программирования задач в сложных программах. Дополнение программ командным языком является мощным средством расширения функций программ. К тому времени подобный подход был известен, но применялся достаточно редко из-за технических сложностей. Известными примерами таких DSL были собственно командные оболочки ОС Unix и язык Emacs Lisp для текстового редактора Emacs. Остерхот отмечает, что создание командного языка DSL для программы — задача, требующая навыков разработки языков, разработки интерпретатора и проработки аспектов встраивания языка в программу (в случае внешнего DSL). Помимо этого, изучение большого количества новых языков затруднительно для пользователей программ. Tcl задумывался им как универсальный язык для встраивания в программы на языке C самого разного плана, позволяющий

избежать описанного процесса разработки и облегчающий использование таких приложений за счёт необходимости изучения только одного языка. В то время всё большее распространение приобретали персональные компьютеры с относительно большими мониторами, мышами и оконными системами. Если в 1980-х гг. файлы сценариев позволяли эффективно управлять вычислениями, то к 1990-м гг. возникла потребность в большом количестве интерактивных графических программ, сценарии для которых тоже должны были быть интерактивными. Tcl представляет собой простой язык, ориентированный на обработку строк. В нём имеется единственный тип данных — строка. И даже структуры управления вычислениями и блоки кода рассматриваются интерпретатором как строки, что даёт основание причислить его гомоиконным языкам. Остерхотом также была создана графическая библиотека Tk, позволяющая легко разрабатывать графические приложения. Эта библиотека стала очень популярной и до сих пор используется для создания простых графических интерфейсов не только на языке Tcl, но и в других языках.

В статье 1998 г. [25] Остерхот назвал языки сценариев языками программирования более высокого уровня для двадцать первого века. В ней он проанализировал причины популярности и эффективности этих языков. Остерхот условно разделяет языки на системные и языки сценариев. Основное различие между ними — в различном выборе уже упоминавшегося выше компромисса между скоростью разработки программ и скоростью их выполнения. Остерхот высказывает предположение, что в будущем число задач, требующих «склеивания» уже написанных компонентов будет только возрастать. Среди особенностей языков сценариев, которые он выделяет, — обработка строк как универсального формата данных, интерпретация, динамическая и слабая типизация. Следует остановиться на последней особенности. Безтиповые или слабо типизированные языки, по мнению Остерхота, намного больше подходят для решения задач в области интеграции компонентов, поскольку не ограничивают программиста в способах трактовки и использования данных. Это справедливо для таких языков как Bourne Shell или Tcl. Однако языки сценариев, получившие распространение к концу 1990-х — началу 2000-х гг., имели уже строгую, хотя и динамическую типизацию. Возможная причина состоит в том [27], что строгие типы позволяют предотвратить ряд ошибок, а в объектно-

ориентированных языках с одиночной диспетчеризацией — контролировать набор методов, применимых к объекту, что упрощает изучение и использование кода. К тому же хорошо продуманные контейнерные типы с удобными конструкторами позволяют создавать требуемые структуры данных на лету во время анализа данных.

На конференции USENIX по языкам DSL 1997 г. Пол Худак (Paul Hudak) выступи с сообщением [16], в котором рассмотрел особенности языков сценариев как вида DSL. Предметной областью для языков сценариев являются компоненты компьютерной системы: элементы графического интерфейса, объекты компонентных технологий типа COM или CORBA, другие программы. Разработка языков сценариев имеет те же особенности, что и разработка маленьких языков. Имеются возможности по созданию встраиваемых или внешних DSL, по выбору способа реализации транслятора (препроцессор, генерируемый синтаксический анализатор с интерпретатором, макросы или процедуры при встраивании и т. д.). Отличает же языки сценариев то, что они предназначаются для решения достаточно сложных задач, а следовательно должны быть полными по Тьюрингу и предоставлять достаточно развитые средства абстракции. То есть должны быть в определённой степени языками общего назначения.

Новое решение этого противоречия в языках сценариев между необходимостью соответствия предметной области (специализацией) и требованиями полноты и выразительной языков (обобщением) получило развитие в начале 2000 гг. Этим решением стали *динамические языки*, такие как Python, JavaScript, Ruby. В середине 1990-х гг. их ещё причисляли к языкам сценариев, но по мере появления новых программ на этих языках стало ясно, что они достаточно мощные, чтобы решать на них задачи, типичные для языков общего назначения, в том числе и для системных языков. С 2000-х гг. название «язык сценариев» используется по отношению к некоторым из этих языков лишь по традиции. Эти языки характеризуются строгой динамической типизацией и поддержкой сразу нескольких парадигм программирования: процедурного, объектно-ориентированного, функционального. Характерным примером таких языков является *Python*, созданный Гвидо ван Россумом (Guido van Rossum) в 1991 г. [27]. Этот язык создавался как язык сценариев, код на котором будет

очень читаемым и простым (в отличие от Perl). На протяжении 1990-х гг. в язык были добавлены элементы для поддержки функционального программирования, которое, как было отмечено выше, позволяет добиться большой выразительности программ за счёт развитых средств абстракции. В версии 2.0, вышедшей в 2000 г. язык уже был мощным мультипарадигменным языком общего назначения. Тем не менее, он остаётся простым в использовании и считается одним из самых легко читаемых языков. Так, например, существовал проект [28] по обучению программированию на Python как студентов-неспециалистов в вузах, так и детей в школах. Хотя этот проект не получил необходимого финансирования и был закрыт, некоторые вузы начали применять язык как первый при обучении программистов. Будучи языком общего назначения, Python сохраняет основные особенности языков сценариев: он является интерпретируемым, динамически типизируемым, имеющим развитые средства разбора строкового вывода.

Языки сценариев позволили уменьшить разрыв между языками общего назначения, позволяющими писать эффективные программы, но сложными в применении и доступными лишь профессиональным программистам, и языками DSL, специфичными для узких предметных областей, простыми в освоении даже для незнакомых с программированием, но неэффективными и не приспособленными для решения задач за рамками узкой ниши. Благодаря этой группе языков, стало доступным создание командного интерфейса к системам, «склеивание» программ для решения комплексных задач на очень высоком уровне. Также разработчики языков сценариев нашли возможности эффективного сочетания нескольких парадигм программирования в одном языке, что позволило современным динамическим языкам начать вытеснять языки общего назначения вроде C и Java из областей создания больших программ и системного программирования.

Заключение

В этой работе был рассмотрен примерно тридцатилетний период развития языков DSL. Эти языки развивались вместе с языками общего назначения, часто заимствуя у них перспективные идеи. В 1960-е гг. вокруг языка Lisp сформировалось сообщество исследователей, развивших методы функционального программирования и создавших первые языки DSL. 1970-е гг. стали началом эпохи современных ОС, уходящих корнями в ОС Unix. В среде разработчиков Unix возникли многочисленные маленькие специализированные языки, прекрасно дополнившие системные языки и средства ОС. В 1980-е гг. благодаря развитию маленьких языков и созданию языков сценариев удалось заметно повысить отношение числа полезных машинных инструкций к числу строк исходного кода. Это способствовало росту производительности программистов, сокращению сроков разработки и повышению сложности функций ПО. К 1990-м гг. стало заметно и обратное влияние DSL на языки общего назначения, выразившееся в развитии и популяризации языков сценариев как динамических языков. Создание специализированных языков стало общепризнанным мощным методом абстракции. Этот подход позволил выражать решения задач в терминах предметной области, фиксировать в языке знания об области способом, который допускает повторное использование и валидацию и оптимизацию на более высоком уровне, упрощает использование языка людьми, не являющимися специалистами в области информатики и вычислительной техники.

Подходы к созданию DSL продолжают применяться в новых предметных областях. Одной из наиболее заметных является область веб-программирования и языки на основе XML, расширяемого языка разметки текста. За последние десять лет было создано большое количество таких языков для разметки и обработки текста, математических формул, хранения и обмена данными между приложениями. Во многом эти языки повторяют качества маленьких языков Unix, но добавляют использование универсального синтаксиса и пространств имён, что определяет их высокую интероперабельность. Ещё одной заметной областью является применение DSL для решения задач программирования крупного масштаба. В области разработки ПО на основе моделей получили развитие DSL с

графической нотацией, приёмы трансформации кода, а также DSL описания элементов программной архитектуры. Относительное новое направление архитектуры ПО начинает приобретать формальную основу, в том числе и благодаря применению DSL. Наконец, динамические языки породили новую волну интереса к языкам программирования в целом и к DSL в частности. В отличие от 1990-х гг., когда распространённых языков общего назначения было менее пяти, на данный момент широко используются около десяти языков, а современные проекты всё чаще ведутся с использованием целого набора языков, применяемых для эффективного решения задач в предметных областях проекта.

Список использованных источников

1. Абельсон Х., Сассман Д. *Структура и интерпретация компьютерных программ*. — Добросвет, 2006. — urn:isbn:5-8459-0192-8
2. Ахо А., Сети Р., Ульман Д. *Компиляторы: принципы, технологии и инструменты*. — Вильямс, 2003. — urn:isbn:5-8459-0189-8
3. Керниган Б., Пайк Р. *Unix. Программное окружение*. — Символ-Плюс, 2003. — urn:isbn:5-9328-6029-4
4. Реймонд Э. *Искусство программирования Unix*. — Вильямс, 2005. — urn:isbn:5-8459-0791-8
5. Уолл Л., Кристиансен Т., Орвант Д. *Программирование на Perl (3-е издание)*. — Символ-Плюс, 2008. — urn:isbn:5-9328-6020-0
6. Aho A., Kernighan B., Weinberger P. *Awk — A Pattern Scanning and Processing Language* // Software — Practice and Experience. — 1979. — Vol. 9, No. 4. — Pp. 267-280. — http://www.cs.buap.mx/~dpinto/ri/awk.pdf
7. Backus J. *Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs* // Communications of the ACM. — ACM Press New York, 1978. — Vol. 21, No. 8. — Pp. 613-641. — http://www.stanford.edu/class/cs242/readings/backus.pdf
8. Bentley J. *Programming Pearls: Little Languages* // Communications of the ACM. — ACM Press New York, 1986. — Vol. 29, No. 8. — Pp. 711-721. — http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.16.4634
9. *Computing Curricula 2001. Computer Science*: Final Report. — Association of Computer Machinery, 2001. — http://www.sigcse.org/cc2001/cc2001.pdf
10. Van Deursen A., Klint P. *Little Languages: Little Maintenance?* // Journal of Software Maintenance. — 1998. — Vol. 10, No. 2. — Pp. 75-92. — http://www.st.ewi.tudelft.nl/~arie/papers/domain.pdf
11. Van Deursen A., Klint P., Visser J. *Domain-Specific Languages: An Annotated Bibliography* // ACM SIGPLAN Notices. — 2000. — Vol. 35, No. 6. — Pp. 26-36. — http://www.st.ewi.tudelft.nl/~arie/papers/dslbib.pdf
12. *Dick Gabriel on Lisp* [Electronic Resource] / M. Völter // Software Engineering Radio. — 2008. — No. 84. — http://www.se-radio.net/podcast/2008-01/episode-84-dick-gabriel-lisp

13. Feldman S. *Make — A Program for Maintaining Computer Programs* // Software — Practice and Experience. — 1979. — Vol. 9, No. 4. — Pp. 255-265. — http://www.memi.umss.edu.bo/~softcons/feldman79make.pdf
14. Graham P. *On Lisp*. — Prentice Hall, 1994. — urn:isbn:0-1303-0552-9
15. Hewitt C. *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot*: Ph. D. dissertation. — Artificial Intelligence Laboratory, MIT, 1971. — http://handle.dtic.mil/100.2/AD744620
16. Hudak P. *The Promise of Domain Specific Languages*: Keynote Address // USENIX Conference on Domain-Specific Languages 1997. — 1997. — http://cs-www.cs.yale.edu/homes/hudak/hudak-dir/dsl/index.htm
17. Johnson S. *Yacc — Yet Another Compiler Compiler*: Computing Science Technical Report 32. — AT&T Bell Laboratories, 1975. — http://www.cs.rpi.edu/courses/fall01/modcomp/yacc.pdf
18. Kay A. *The Reactive Engine*: Ph. D. dissertation. — University of Utah, 1969. — http://portal.acm.org/citation.cfm?id=905541
19. Kernighan B., Cherry L. *A System for Typesetting Mathematics* // Communications of the ACM. — 1975. — Vol. 18, No. 3. — Pp. 151-157. — http://doi.acm.org/10.1145/360680.360684
20. Korn D. *ksh — An Extensible High Level Language* // Proceedings of the USENIX 1994 Very High Level Languages Symposium. — 2004. — http://www.cs.princeton.edu/~jlk/kornshell/doc/vhll.ps.gz
21. Lesk M., Schmidt E. *Lex — A Lexical Analyzer Generator*: Computing Science Technical Report 39. — AT&T Bell Laboratories, 1975. — http://wolfram.schneider.org/bsd/7thEdManVol2/lex/lex.pdf
22. McCarthy J. et al. *A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence*. — 1955. — http://www-formal.stanford.edu/jmc/history/dartmouth.html
23. McCarthy J. *History of Lisp* // History of Programming Languages. — ACM Press New York, 1978. — Vol. I. — Pp. 173-185. — http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.16.4634
24. Ousterhout J. *Tcl: An Embeddable Command Language* // Proceedings of the 1990 Winter USENIX Conference. — 1990. — Pp. 133-146. —

http://www.eecs.berkeley.edu/Pubs/TechRpts/1989/CSD-89-541.pdf
25. Ousterhout J. *Scripting: Higher Level Programming for the 21st Century* // IEEE Computer. — 1998. — Vol. 31, No. 3. — Pp. 23-30. — http://doi.ieeecomputersociety.org/10.1109/2.660187
26. Pike R. *Notes on Programming in C* [Electronic Resource]. — 1989. — http://www.lysator.liu.se/c/pikestyle.html
27. Pilgrim M. *Dive Into Python*. — Apress, 2004. — urn:isbn:1-5905-9356-1
28. Van Rossum G. *Computer Programming for Everybody*: CNRI Proposal #90120-1a [Electronic Resource]. — 1999. — http://www.python.org /doc/essays/cp4e/
29. Seibel P. *Practical Common Lisp* [Electronic Resource] // Google Tech Talks. — 2006. — http://video.google.com/videoplay?docid=448441135356213813
30. Spinellis D. *Notable Design Patterns for Domain-Specific Languages* // The Journal of Systems and Software. — 2001. — Vol. 56, No. 1. — Pp. 91-99. — http://www.spinellis.gr/pubs/jrnl/2000-JSS-DSLPatterns/html/dslpat.pdf
31. Steele G. *Growing a Language* [Electronic Resource] // OOPSLA'98. — 1998. — http://video.google.com/videoplay?docid=-8860158196198824415
32. Sussman G., Steele G. *Scheme: An Interpreter for Extended Lambda Calculus*: AI Lab Memo AIM-349. — Artificial Intelligence Laboratory, MIT, 1975. — http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-349.pdf
33. Winograd T. *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language*: MAC-TR-84. — MIT Project MAC, 1971. — http://hci.stanford.edu/~winograd/shrdlu/AITR-235.pdf