информатика

Сафонов Владимир Олегович

СОВРЕМЕННЫЕ МЕТОДЫ, ИНСТРУМЕНТЫ И РЕСУРСЫ ДЛЯ ОБУЧЕНИЯ КОМПИЛЯТОРАМ

Аннотация

В статье анализируется 17-летний опыт автора в области обучения компиляторам. Подчеркивается фундаментальный характер компиляторов как учебной дисциплины и их важность для полноценного современного университетского образования в области ИТ. Рассмотрены принципы построения современного курса по компиляторам, подход автора к обучению компиляторам и современные инструменты построения компиляторов и обучения компиляторам. Даны ссылки на публикации автора в данной области и на полезные Web-ресурсы.

Ключевые слова: компиляторы, динамические компиляторы, надежные и безопасные компиляторы, лексический анализ, синтаксический анализ, семантический анализ, оптимизация, генерация кода, инструменты построения компиляторов, Java, .NET, Microsoft Phoenix, ANTLR, CoCo/R.

ВВЕДЕНИЕ

Статья продолжает серию статей автора [1–3] о преподавании ИТ. Отметим также весьма интересную серию статей проф. Б.К. Мартыненко об обучении компиляторам, начатую статьей [4] и созвучную данной статье.

Компиляторы, как и языки программирования, существуют уже около 60 лет. Но они отнюдь не утратили своего значения, ни как инструменты системного программирования, ни как научная и учебная дисциплина. Собственно разработчиков компиляторов в мире сравнительно немного, и большинство студентов впоследствии вряд ли встретятся в практике своей работы именно с задачей разработки нового компилятора. Однако в компиляторах используются классические и современные методы и алгоритмы представления и обработки данных, информационного поиска, анализа деревьев и графов, которые носят более общий характер и полезны для всех ИТ-специалистов. Студент, изучивший и практически освоивший методы и инструменты построения компиляторов, получит, таким образом, необходимые для его повседневной работы как системного программиста теоретические знания и практические навыки.

В настоящее время наблюдается бурное развитие компиляторов, возможности которых существенно расширились. Появились принципиально новые виды компиляторов, например, динамические (just-in-time) и графовые (graph) компиляторы. Кроме того, постоянно возникают новые задачи в области компиляции. Свежий пример из области профессиональных интересов моих коллег – методы оптимизации для новых графических мно-

гоядерных процессоров (GPU – Graphics Processor Units) серии Tesla фирмы nVidia.

В связи с этим, по мнению автора, обучение компиляторам студентов в области ИТ в университетах должно носить столь же фундаментальный характер, как обучение теории множеств, математическому анализу, алгебре и математической логике – в области математики.

В феврале 2010 г. в издательстве John Wiley & Sons вышла из печати книга автора по компиляторам [5], которая, наряду с материалами курсов и семинаров автора по компиляторам [6–8], опубликованными в Интернете, может быть использована для обучения компиляторам на современном уровне. Его принципы рассмотрены в данной статье. Информация о моей новой книге и ссылки на учебные ресурсы доступны на моих Web-страницах: http://www.vladimirsafonov.org/trustworthycompilers.

Автор в течение 17 лет читает постоянно обновляемый курс по компиляторам на математико-механическом факультете СПбГУ (в настоящее время объем курса – 64 часа), в течение ряда лет вел семинар по компиляторам на мат-мехе СПбГУ и имеет 33-летний опыт разработки исследовательских и промышленных компиляторов, включая компиляторы для МВК «Эльбрус» [9], компиляторы фирмы Sun [3, 5] и компиляторы на основе инструментария Microsoft Phoenix [5].

1. КЛАССИФИКАЦИЯ И СОВРЕМЕННЫЕ НАПРАВЛЕНИЯ РАЗВИТИЯ КОМПИЛЯТОРОВ

В традиционном понимании, как широко известно, *компилятор* (*compiler*) – это системная программа, транслирующая исходный код, написанный на некотором языке программирования высокого уровня (например C, Java, C#), в объектный (машинный) код конкретной аппаратной платформы, например x86 или SPARC. Исторически первым для обозначения такого рода программных инструментов стал в 1950-х гг. термин *программирующая программа*, затем долгие годы использовался и до сих пор используется термин *транслятор*.

Однако в современном понимании концепция компилятора гораздо шире. Рассмотрим классификацию компиляторов, приведенную в работах [5–7]:

- *Традиционные (классические) компиляторы*, удовлетворяющие приведенному выше определению. Классическими учебниками по традиционным компиляторам являются работы [10, 11]. Отметим, что в книге автора [5] описаны оригинальные эффективные методы компиляции, более быстрые, чем классические.
- *Интерпретаторы* – другой способ реализации языков программирования, основанный на эмуляции (в терминах исходного кода) выполнения программы, представленной на некотором промежуточном языке (например в постфиксной форме), на который она предварительно оттранслирована с языка высокого уровня. Интерпретационные реализации традиционно характерны для языков символьной обработки (LISP, SNOBOL) в связи с их расширенными динамическими возможностями, в частности, с динамической типизацией. С помощью интерпретаторов реализуются также некоторые современные языки разработки *скриптов* и языки *Web-программирования*. Реализация языка Java в первой версии (Java 1.0, 1995 г.) была также интерпретационной.
- *Кросс-компиляторы* – компиляторы, использующие для своей работы некоторую распространенную и удобную платформу, например x86 или x64, и генерирующие объектный код для другой платформы (как правило, для какого-либо встроенного микропроцессора).
- *Пошаговые компиляторы (incremental compilers)* – разновидность компиляторов, популярная в 1960-х–1970-х гг. Пошаговый компилятор позволяет вводить, редактировать, компилировать и отлаживать исходную программу по *шагам*. Шаг – это описание, группа описаний, оператор, группа операторов, заголовок процедуры (метода). Подобная схема использования

полезна для пошаговой разработки и отладки программ и обучения программированию. Однако подобная реализация неэффективна, так как при выполнении программы используются таблица и интерпретатор шагов.

- *Конвертор* – компилятор с одного языка высокого уровня на другой, например с более старого (COBOL) на более новый (Java, C#). Конверторы используются для *реинжиниринга* программ – их переноса на новые языковые платформы. Кроме того, конвертор – это распространенный способ реализации каких-либо исследовательских расширений языка программирования. Например, в нашей системе управления знаниями Knowledge.NET [12] расширение языка C# средствами представления знаний реализовано путем конвертирования в базовый язык C#.
- *Just-in-time (JIT) – компиляторы* (или *динамические* компиляторы) –новый вид компиляторов, широко используемый в современных платформах для разработки программ (Java, .NET) и впервые реализованный в конце 1980-х – начале 1990-х гг. в системе VisualWorks фирмы Xerox PARC (США) – интегрированной среде разработки программ на языке Smalltalk. Кстати, этой же фирмой при реализации разработанного в ней языка Smalltalk (а не в реализации Java, как иногда считают) впервые использован термин *байт-код* в значении «промежуточное бинарное представление программы в виде постфиксной записи». JIT-компилятор вызывается во время выполнения программы, представленной в промежуточном коде (в постфиксной форме – в виде байт-кода в Java или CIL-кода – аналога байт-кода в .NET). При первом вызове метода JIT-компилятор транслирует промежуточный код метода в его платформно-зависимый (native) код используемой аппаратной платформы и обеспечивает в дальнейшем, при последующих вызовах метода, запуск его откомпилированного native-кода, что в большинстве случаев значительно уменьшает суммарное время выполнения программы.
- *Ahead-of-time (AOT) – компиляторы* (или *предкомпиляторы*) – также популярный в современных платформах для разработки программ вид компиляторов, выполняющий *предкомпиляцию* двоичного промежуточного кода единицы компиляции из байт-кода полностью в native-код перед его выполнением.
- *Двоичные компиляторы (binary compilers)* – это компиляторы двоичного кода одной аппаратной платформы в двоичный код другой, как правило, более новой (без анализа и компиляции исходного кода программы на языке высокого уровня). Двоичные компиляторы широко используются для переноса полезного программного обеспечения (включая игры и даже операционные системы) на новые аппаратные платформы, если исходный код программ по каким-либо причинам недоступен.
- *Графовые компиляторы (graph compilers)* – современная разновидность компиляторов, которые транслируют не исходный код, а некоторое графовое (визуальное) представление программы либо в другое ее графовое представление, либо в код на каком-либо языке (например Java). Графовые компиляторы активно используются в электронике, химии и во многих других областях, методы которых основаны на визуальных представлениях каких-либо схем или программ.

Приведенная классификация дает представление о разнообразии видов компиляторов.

В настоящее время наблюдается настоящий бум развития компиляторов, вызванный, на наш взгляд, следующими основными причинами:

- *Развитие компьютерных архитектур* и появление новых видов параллельных процессоров, например *Very Long Instruction Word (VLIW), Explicit Parallelism Instruction Computer (EPIC), многоядерные (multicore) процессоры*. Параллельные архитектуры ставят перед разработчиками компиляторов принципиально новые, более сложные задачи, связанные с необходимостью статического планирования параллельных

вычислений на уровне генерируемого объектного кода программ.

- *Популярность новых платформ для разработки программ – Java и .NET*, основанных, как уже отмечалось, на двухэтапной реализации – компиляция в промежуточную постфиксную форму, с последующей JIT-компиляцией во время выполнения программы. Подобная схема реализации требует решения целого ряда новых задач, например JIT-компиляция и оптимизации при JIT-компиляции с целью повышения производительности программ.
- *Поддержка многоязыкового объектно-ориентированного программирования на платформе .NET* [13]. Программа для .NET может использовать модули, написанные на любых реализованных в ней языках программирования (например C#, Visual Basic, Managed C++ и др.), при условии соблюдения их реализациями требований межъязыковой совместимости (*Common Language Specification – CLS*), причем все эти языки равноправны, а базовые объектно-ориентированные механизмы – наследование, контроль типов, обработка исключений и др. – реализованы единым образом для всех яыков, так что, например, возможны описание класса на одном языке, а его потомка – на другом, либо генерация исключения в модуле на C# и его обработка в модуле на Visual Basic. Подобные возможности .NET вызвали целую волну коммерческих и исследовательских работ по реализации для .NET все новых и новых языков, которых сейчас реализовано более 30, и, соответственно, по развитию инструментов построения компиляторов (лексических и синтаксических анализаторов), облегчающих подобную реализацию.
- *Развитие языков и технологий Web-программирования* (например JavaScript / ECMAScript, Ruby, Python), основанных на динамической типизации – например, возможности динамически ввести в объект новое поле путем присваивания – и, следовательно, требующих развития эффективных методов компиляции и динамической поддержки подобных языков. Достаточно упомянуть, что динамическая типизация языков Web-программирования стимулировала разработку новой расширенной виртуальной машины .NET с поддержкой динамизма типов – *Dynamic Language Runtime (DLR)*.
- *Развитие мобильных устройств и технологий разработки программного обеспечения для них*, стимулировавшее развитие компиляторов и методов компиляции для таких классов устройств.
- *Популярность концепции надежных и безопасных вычислений (trustworthy computing)* [14] и связанное с ней развитие работ по *надежным и безопасным компиляторам (trustworthy compilers)* [5] – новым видам компиляторов, верифицирующих транслируемые ими программы (*verifying compilers*), например *Spec#* и *vcc* фирмы Microsoft Research, – и компиляторам, исходный код которых, в свою очередь, формально верифицирован (*verified compilers*), – например компилятор *CompCert* с промышленного расширения языка Си, разработанный в INRIA (Франция).

2. ПРИНЦИПЫ ПОСТРОЕНИЯ И ПРЕПОДАВАНИЯ СОВРЕМЕННОГО КУРСА ПО КОМПИЛЯТОРАМ

Использование методики ERATO [1]. В многолетней преподавательской практике автора сложился подход к преподаванию компиляторов (как и других дисциплин в области ИТ), основанный на методике *ERATO (Experience, Retrospective, Analysis, Theory, Oncoming perspectives)*; в русском варианте – *ТРОПА (Теория, Ретроспектива, Опыт, Перспективы, Анализ)*. Данная методика подробно описана в предыдущих работах автора [1, 5]. Применительно к компиляторам она означает следующее:

– рассматривается история развития компиляторов и методов компиляции;

– активно используется опыт, научные и практические результаты автора в области компиляторов [5, 9]; обучение компиляторам ведется на основе информации

о реальных, а не «игрушечных» компиляторных проектах; студенты и аспиранты активно участвуют в них под руководством автора;

– анализируются различные подходы к компиляторам и фазам компиляции;

– изучаются и используются классические и современные инструменты построения компиляторов;

– рассматриваются новые направления в области компиляторов, перспективы развития компиляторов и инструментов их построения.

Анализ концепции надежного и безопасного компилятора (trustworthy compiler). При обучении компиляторам постоянно подчеркиваются особенности их архитектуры, методов и алгоритмов, способствующие их надежности и безопасности (trustworthiness [5]), как наиболее важные для их широкого использования:

– генерация надежного и безопасного объектного кода;

– надежность поведения компилятора; его устойчивость к ошибкам в компилируемой программе; высокое качество диагностики и нейтрализации ошибок (error recovery);

– генерация надежного и безопасного кода;

– защита компилятором своих промежуточных данных (например промежуточного кода программы, передаваемого от одной фазы компиляции другой через временный файл) от взлома или непреднамеренной порчи;

– надежность методов проектирования и реализации компилятора;

– проверка компилятором надежности и безопасности компилируемой программы; в идеальном варианте – ее формальная верификация, то есть проверка семантической корректности ее реализации, а не только проверка отсутствия лексических, синтаксических и семантических ошибок. Данная идея – verifying compiler – принадлежит Ч. Хоару [5];

– особо тщательное изучение базовой концепции *типа*: методов типизации, алгоритмов контроля типов, обработки типов во время выполнения программы и т. д.

Примерный план современного курса по компиляторам. Приведем список тем, рассматриваемых в годовом курсе автора «Компиляторы» на математико-механическом факультете СПбГУ:

1. Понятие надежного и безопасного компилятора (trustworthy compiler). Верифицирующие и верифицированные компиляторы. Виды компиляторов. Фазы компиляции.

2. Особенности реализации Java и компиляции для платформы .NET.

3. Принципы построения компиляторов. Front-end и back-end. Метод раскрутки (bootstrapping). Системы построения трансляторов (СПТ).

4. Лексический анализ. Классы лексем. Обработка идентификаторов и ключевых слов. Таблица идентификаторов. Использование хеш-функций.

5. Лексический анализ. Архитектура и принципы реализации. Утилита lex. Спецификация и генерация лексических анализаторов в системе ANTLR.

6. Синтаксический анализ. Метод рекурсивного спуска. Реализация заглядывания вперед (lookahead) для метода рекурсивного спуска.

7. Методы нейтрализации синтаксических ошибок. Нейтрализация ошибок для метода рекурсивного спуска. Стратегии panic mode и intelligent panic mode.

8. Синтаксический анализ снизу вверх (сдвиг-свертка). Классы языков LL(k) и LR(k). SLR-анализ. LR(0)-элементы. Операции closure(I) и goto(I,X). Канонический набор множеств LR(0)-элементов.

9. Языки, не принадлежащие классу SLR. LR(1)-элементы. LR(1)-анализатор. Понятие о LALR(1)-анализе. Нейтрализация ошибок при LR-анализе.

10. Утилита yacc. Современные инструменты построения компиляторов – ANTLR, CoCo/R, JavaCC, SableCC, bison и др.

11. Семантический анализ. Атрибутные грамматики. Синтезируемые и наследуемые атрибуты и их вычисление. L-атрибутные грамматики.

12. Семантические атрибуты типов, переменных и выражений (тип, смещение и др.).

13. Принципы эффективного вычисления семантических атрибутов. Вычисление синтезируемых атрибутов при анализе снизу вверх. Использование списков примененных вхождений. Системы определений и эффективный алгоритм их анализа.

14. Обработка описаний и идентификация. Таблица описаний (NL). Эффективные алгоритмы идентификации (lookup). Ссылка на текущее действующее определение (CED). Особенности идентификации для объектно-ориентированных языков.

15. Тип. Именная и структурная идентичность типов. Совместимость типов. Совместимость типов по присваиванию.

16. Обработка обозначений типов. Таблица типов. Контроль типов.

17. Формы внутреннего представления программы в компиляторах: обратная польская запись, триплеты, PCC-деревья.

18. Оптимизация программ. Обзор основных методов оптимизации. Понятие о смешанных вычислениях.

19. Assignment (SSA). Зависимости по управлению и по данным. Структуры данных, используемые при оптимизации. Оптимизация в Microsoft Phoenix и в компиляторах Sun Studio.

20. Генерация кода. Особенности генерации кода для платформы Microsoft.NET. Общая схема организации компиляторов для платформы SPARC и роль генератора кода.

21. Представление данных и адресация переменных: простые типы, записи (структуры), записи с вариантами (объединения), массивы, указатели (управляемые и неуправляемые).

22. Процедуры, функции, методы, процедурные параметры, их представление. Организация стека для вызовов процедур. Статическая и динамическая цепочки. Дисплей. Адресация локальных данных. Адресные пары. Особенности реализации дисплея для RISC-архитектур.

23. Элементы архитектуры RISC и SPARC. Регистровые окна. Соглашение о связях (ABI) для платформы SPARC. ELF-файлы и их структура (секции). Генерация кода для платформы SPARC (пример).

24. Отладочная информация и ее роль в системе программирования. Генерация отладочной информации для Microsoft.NET и платформы SPARC. Структура STABS – отладочной информации для платформы SPARC.

25. Методы генерации кода для описаний, выражений и операторов.

26. Система поддержки выполнения (runtime) и ее функции. Связь runtime и операционной системы.

27. Обзор инструментального комплекса для разработки компиляторов. Microsoft Phoenix.

28. Just-in-Time (JIT) – компиляторы. Архитектура JIT-компилятора. Оптимизации при JIT-компиляции.

29. Графовые грамматики (graph grammars) и компиляторы графов (graph compilers).

На мой взгляд, такие фундаментальные дисциплины, как оптимизация и преобразования программ, обычно изучаемые в курсе компиляторов, заслуживают отдельных специальных или даже обязательных курсов.

3. СОВРЕМЕННЫЕ ИНСТРУМЕНТЫ ПОСТРОЕНИЯ КОМПИЛЯТОРОВ И ОБУЧЕНИЯ КОМПИЛЯТОРАМ

В настоящее время разработано и доступно большое число современных инструментов для разработки компиляторов, заслуживающих детального изучения и широкого использования [5]. Среди этих инструментов есть такие, которые (что стало уже многолетней традицией) автоматизируют разработку лексических и синтаксических анализаторов, но более современными методами, например ANTLR, CoCo/R, SableCC, *bison*. Последний, например, предлагает *обобщенный синтаксический LR-анализ (generalized LR parsing)* – параллельный LR-разбор вариантов синтак-

сиса конструкции с последующим слиянием «клонов» синтаксического анализатора, находящихся в идентичных состояниях, и удалением «клонов», зашедших в тупик. Существуют также инструментарии, предназначенные для поддержки разработки оптимизирующих и кодогенерирующих частей компиляторов (back-end). Среди подобных инструментов уникальную роль играет Microsoft Phoenix, пользователями которого наша группа является с 2003 г. Phoenix обеспечивает средства анализа двоичного кода нескольких платформ, его перевода в собственное многоуровневое внутреннее представление (Phoenix IR) с последующей его оптимизацией и трансляцией в исполняемый объектный код любой из целевых платформ, набор которых может быть расширен (в настоящее время это x86, x64, AMD64, .NET).

Изучение и использование этих инструментов традиционно вызывает у студентов большой интерес.

Для изучения компиляторов на реальных примерах используется также академическая версия .NET (SSCLI, или Rotor [5]), в которую включены компиляторы с языков C# и JScript с открытым исходным кодом. Однако пока лишь немногим студентам удается, по моему заданию, разобраться по хорошо самодокументированному исходному коду этих компиляторов в их архитектуре и используемых методах и алгоритмах и сделать на основе полученной информации доклад на семинаре или, тем более, расширить доступные в виде исходного кода компиляторы новыми возможностями.

ЗАКЛЮЧЕНИЕ

Поколение программистов, способных «с нуля» (from scratch) полностью спроектировать и разработать компилятор, уходит в прошлое, поскольку принципиально изменились подходы к разработке программного обеспечения в целом. На смену ему приходит другое поколение, которое необходимо научить правильному использованию существующих инструментов для разработки компиляторов. Студентам, кроме того, необходимо объяснять и доказывать на конкретных практических примерах, что компиляторы – это отнюдь не устаревшая и застывшая, а современная и активно развивающаяся область, в развитие которой они, после изучения классических и современных методов разработки компиляторов, смогут внести свой большой вклад, предложив новые, более эффективные методы и алгоритмы, чем описанные в классических работах, и, тем самым, хорошо зарекомендовать себя как высококвалифицированных системных программистов. Студентов необходимо убедить, что знание компиляторов им впоследствии очень пригодится и станет основой для их собственного развития и творческой работы в области программного обеспечения.

Данная статья является лишь началом дискуссии о современных методах разработки компиляторов. Буду благодарен уважаемым коллегам за отзывы, замечания и предложения. Приглашаю присылать их по электронной почте: *vosafonov@gmail.com.*

Литература

1. *Сафонов В.О.* Актуальные проблемы преподавания технологий программирования в России // Компьютерные инструменты в образовании, 2008, № 5.

2. *Сафонов В.О.* «Золотой век» операционных систем и преподавание ОС в университетах России // Компьютерные инструменты в образовании, 2009, № 1.

3. *Сафонов В.О.* Технологии Sun и открытое программное обеспечение: опыт мат-меха // Компьютерные инструменты в образовании, 2009, № 2.

4. *Мартыненко Б.К.* Учебный исследовательский проект реализации алгоритмических языков // Компьютерные инструменты в образовании, 2008, № 5.

5. *Safonov V.O.* Trustworthy Compilers. – Wiley Interscience. John Wiley & Sons, February 2010.

6. *Safonov V.O.* Trustworthy compiler development. University course curriculum / http://www.facultyresourcecenter.com/curriculum/pfv.aspx?ID=7698. February 2010

7. *Safonov V.O.* Compiler Development. University course curriculum / http://www.facultyresourcecenter.com/curriculum/pfv.aspx?ID=5938. December 2005

8. *Safonov V.O.* Compiler Development. Undergraduate seminar curriculum / http://www.facultyresourcecenter.com/curriculum/pfv.aspx?ID=6244. November 2005

9. *Сафонов В.О.* Языки и методы программирования в системе «Эльбрус». – М.Ж Наука, 1989.

10. *Ахо А., Сети Р., Ульман Дж.* Компиляторы: Принципы, технологии, инструменты. М.: Вильямс, 2000.

11. *Грис Д.* Конструирование компиляторов для цифровых вычислительных машин. М.: Мир, 1975.

12. Web-сайт проекта Knowledge.NET // http://www.knowledge-net.ru

13. *Сафонов В.О.* Платформа .NET: принципы, возможности, перспективы // Компьютерные инструменты в образовании, 2004, № 4.

14. *Сафонов В.О.* Современные технологии разработки надежных и безопасных программ (trustworthy computing) // Компьютерные инструменты в образовании, 2008, № 6.

Abstract

The article analyses the author's 17-years experience in compiler teaching. Fundamental nature of compiler development as educational discipline and their importance for modern university education in the IT area are emphasized. Principles of organization of a modern compiler course and the author's approach to compiler teaching, and modern tools for compiler development and compiler teaching are covered. References to author's publications in this area and to useful Web resources are provided.

Keywords: compilers, just-in-time (JIT) compilers, trustworthy compilers, lexical analysis, parsing, semantic analysis, optimization, code generation, compiler development tools, Java, .NET, Microsoft Phoenix, ANTLR, CoCo/R.

Сафонов Владимир Олегович,
доктор технических наук,
профессор кафедры информатики
СПбГУ, руководитель лаборатории
Java-технологии,
v_o_safonov@mail.ru