## Общие понятия

**Транслятор** — программа или техническое средство, выполняющее *трансляцию программы*.

**Трансляция программы** — преобразование программы, представленной на одном из языков программирования, в программу на другом языке и, в определённом смысле, равносильную первой.

Транслятор обычно выполняет также диагностику ошибок, формирует словари идентификаторов, выдаёт для печати текст программы и т. д.

Язык, на котором представлена входная программа, называется *исходным языком*, а сама программа — *исходным кодом*. Выходной язык называется *целевым языком, а выходная (результирующая) программа —* *объектным кодом*.

В общем случае, понятие трансляции относится не только к языкам программирования, но и к другим языкам — как формальным компьютерным (вроде языков разметки типа HTML), так и естественным (русскому, английскому и т. п.)

# Виды трансляторов

Существует несколько видов трансляторов:

* *Диалоговый* транслятор — транслятор, обеспечивающий использование языка программирования в режиме разделения времени.
* *Синтаксически-ориентированный* (*синтаксически-управляемый)* транслятор — транслятор, получающий на вход описание синтаксиса и семантики языка, текст на описанном языке и выполняющий трансляцию в соответствии с заданным описанием.
* *Однопроходной* транслятор — транслятор, создающий объектный модуль при однократном последовательном чтении исходного кода (за один проход).
* *Многопроходной* транслятор — транслятор, создающий объектный модуль после нескольких чтений исходного кода (за несколько проходов).
* *Оптимизирующий* транслятор — транслятор, выполняющий оптимизацию создаваемого кода перед записью в объектный файл.
* *Тестовый* транслятор — транслятор, получающий на вход исходный код и выдающий на выходе изменённый исходный код. Запускается перед основным транслятором для добавления в исходный код отладочных процедур. Например, транслятор с языка ассемблера может выполнять замену макрокоманд на код.
* *Обратный* транслятор — транслятор, выполняющий преобразование машинного кода в текст на каком-либо языке программирования.

# Реализации

Цель трансляции — преобразование текста с одного языка на язык, понятный адресату. При трансляции компьютерной программы адресатом может быть:

* устройство — процессор (трансляция называется компиляцией);
* программа — интерпретатор (трансляция называется интерпретацией).

Виды трансляции:

* компиляция;
* интерпретация;
* динамическая компиляция.

# Компилятор

Компьютерная программа, выполняющая *компиляцию*.

**Компиляция** — трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера). Входной информацией для компилятора (исходный код) является описание алгоритма или программа на объектно-ориентированном языке, а на выходе компилятора — эквивалентное описание алгоритма на машинно-ориентированном языке (объектный код).

**Компилировать** — проводить трансляцию машинной программы с объектно-ориентированного языка] на машинно-ориентированный язык.

## Виды компиляторов

* *Векторизующий*. Транслирует исходный код в машинный код компьютеров, оснащённых векторным процессором.
* *Гибкий*. Сконструирован по модульному принципу, управляется таблицами и запрограммирован на языке высокого уровня или реализован с помощью компилятора компиляторов.
* *Диалоговый*.
* *Инкрементальный*. Повторно транслирует фрагменты программы и дополнения к ней без перекомпиляции всей программы.
* *Интерпретирующий (пошаговый)*. Последовательно выполняет независимую компиляцию каждого отдельного оператора (команды) исходной программы.
* *Компилятор компиляторов*. Транслятор, воспринимающий формальное описание языка программирования и генерирующий компилятор для этого языка.

Синтаксис выражается в виде Форма Бэкуса — Наура (формальная система описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории. БНФ используется для описания контекстно-свободных формальных грамматик, Пример: от пример БНФ-конструкции, описывающей правильные скобочные последовательности: <правпосл>::=<пусто> | (<правпосл>) | <правпосл><правпосл>) или её производной и должен удовлетворять правилам того метода синтаксического анализа, который будет использоваться в генерируемом компиляторе.

Семантика языка обычно описывается путём ассоциирования процедуры генерации кода с каждой синтаксической конструкцией, причём необходимая процедура вызывается всякий раз, когда соответствующая конструкция распознаётся программой синтаксического анализа. Таким образом, пользователю компилятора компиляторов в любом случае нужно разработать исполняющие структуры и выбрать способ преобразования каждой входной синтаксической конструкции в операции выходного языка или в машинные операции, после чего нужно написать собственно процедуры генерации кода. Следовательно, компилятор компиляторов — это полезное средство, помогающее писать компиляторы, но не более того.

Строго говоря, *генератор программы синтаксического анализа* представляет собой часть компилятора компиляторов, однако оба термина часто используются как синонимы.

*Компилятор компиляторов Bison*

Bison – это GNU-выпуск известной программы YACC, предназначенной для порождения компиляторов по описанной пользователем КС-грамматике.

Одним из следствий расширения возможностей и доступности использования вычислительной техники стало производство прикладных программных систем со всё более разнообразным и сложным поведением. При этом краеугольный принцип кибернетики -  принцип необходимого и достаточного разнообразия, сформулированный Н.Виннером и Р.Эшби ещё до , гласит, что степень разнообразия управления подобной сущностью должна соответствовать степени разнообразия её поведения.

* *Отладочный*. Устраняет отдельные виды синтаксических ошибок.
* *Резидентный*. Постоянно находится в оперативной памяти и доступен для повторного использования многими задачами.
* *Самокомпилируемый*. Написан на том же языке, с которого осуществляется трансляция.

Метод создания транслятора для некоторого языка программирования, при котором транслятор пишется на том же языке программирования; создание транслятором исполняемых файлов из исходного кода самого транслятора. Используется для переноса трансляторов на новые платформы. Появился в середине 1950-х годов. Позволяет создать транслятор, который генерирует сам себя. Применялся для создания трансляторов многих языков программирования, включая языки BASIC, Алгол, Си, Паскаль, ПЛ/1, Factor, Haskell, Modula-2, Oberon, OCaml, Common Lisp, Scheme, Java, Python, Scala, Nemerle и другие.

* *Универсальный*. Основан на формальном описании синтаксиса и семантики входного языка. Составными частями такого компилятора являются: ядро, синтаксический и семантический загрузчики

## Виды компиляции

* *Пакетная*. Компиляция нескольких исходных модулей в одном пункте задания.
* *Построчная*. То же, что и интерпретация.
* *Условная*. Компиляция, при которой транслируемый текст зависит от условий, заданных в исходной программе директивами компилятора. Так, в зависимости от значения некоторой константы, можно включать или выключать трансляцию части текста программы.

## Процесс компиляции

Процесс компиляции состоит из следующих этапов:

* Лексический анализ. На этом этапе последовательность символов исходного файла преобразуется в последовательность лексем.
* Синтаксический (грамматический) анализ. Последовательность лексем преобразуется в дерево разбора.
* Семантический анализ. Дерево разбора обрабатывается с целью установления его семантики (смысла) — например, привязка идентификаторов к их декларациям, типам, проверка совместимости, определение типов выражений и т. д. Результат обычно называется «промежуточным представлением/кодом», и может быть дополненным деревом разбора, новым деревом, абстрактным набором команд или чем-то ещё, удобным для дальнейшей обработки.
* Оптимизация. Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла. Оптимизация может быть на разных уровнях и этапах — например, над промежуточным кодом или над конечным машинным кодом.
* Генерация кода. Из промежуточного представления порождается код на целевом языке.

В конкретных реализациях компиляторов эти этапы могут быть разделены или, наоборот, совмещены в том или ином виде.

## Генерация кода

### Генерация машинного кода

Большинство компиляторов переводит программу с некоторого высокоуровневого языка программирования в машинный код, который может быть непосредственно выполнен процессором. Как правило, этот код также ориентирован на исполнение в среде конкретной операционной системы, поскольку использует предоставляемые ею возможности (системные вызовы, библиотеки функций). Архитектура (набор программно-аппаратных средств), для которой производится компиляция, называется *целевой машиной*.

Результат компиляции — исполнимый модуль — обладает максимальной возможной производительностью, однако привязан к определённой операционной системе и процессору (и не будет работать на других).

Для каждой целевой машины (IBM, Apple, Sun и т. д.) и каждой операционной системы или семейства операционных систем, работающих на целевой машине, требуется написание своего компилятора. Существуют также так называемые *кросс-компиляторы*, позволяющие на одной машине и в среде одной ОС генерировать код, предназначенный для выполнения на другой целевой машине и/или в среде другой ОС. Кроме того, компиляторы могут оптимизировать код под разные модели из одного семейства процессоров (путём поддержки специфичных для этих моделей особенностей или расширений наборов инструкций). Например, код, скомпилированный под процессоры семейства Pentium, может учитывать особенности распараллеливания инструкций и использовать их специфичные расширения — MMX, SSE и т. п.

Некоторые компиляторы переводят программу с языка высокого уровня не прямо в машинный код, а на язык ассемблера (примером может служить PureBasic, транслирующий бейсик-код в ассемблер FASM). Это делается для упрощения части компилятора, отвечающей за кодогенерацию, и повышения его переносимости (задача окончательной генерации кода и привязки его к требуемой целевой платформе перекладывается на ассемблер), либо для возможности контроля и исправления результата компиляции программистом.

### Генерация байт-кода

Результатом работы компилятора может быть программа на специально созданном низкоуровневом языке, подлежащем интерпретации *виртуальной машиной*. Такой язык называется псевдокодом или байт-кодом. Как правило, он не является машинным кодом какого-либо компьютера и программы на нём могут исполняться на различных архитектурах, где имеется соответствующая виртуальная машина, но в некоторых случаях создаются аппаратные платформы, напрямую поддерживающие псевдокод какого-либо языка. Например, псевдокод языка Java называется байт-кодом Java и выполняется в Java Virtual Machine, для его прямого исполнения была создана спецификация процессора picoJava. Для платформы .NET Framework псевдокод называется Common Intermediate Language (CIL), а среда исполнения — Common Language Runtime (CLR).

Некоторые реализации интерпретируемых языков высокого уровня (например, Perl) используют байт-код для оптимизации исполнения: затратные этапы синтаксического анализа и преобразование текста программы в байт-код выполняются один раз при загрузке, затем соответствующий код может многократно использоваться без промежуточных этапов.

### Динамическая компиляция

Из-за необходимости интерпретации байт-код выполняется значительно медленнее машинного кода сравнимой функциональности, однако он более переносим (не зависит от операционной системы и модели процессора). Чтобы ускорить выполнение байт-кода, используется *динамическая компиляция*, когда виртуальная машина транслирует псевдокод в машинный код непосредственно перед его первым исполнением (и при повторных обращениях к коду исполняется уже скомпилированный вариант).

CIL-код также компилируется в код целевой машины JIT-компилятором, а библиотеки .NET Framework компилируются заранее.

### Декомпиляция

Существуют программы, которые решают обратную задачу — перевод программы с низкоуровневого языка на высокоуровневый. Этот процесс называют декомпиляцией, а такие программы — декомпиляторами. Но поскольку компиляция — это процесс с потерями, точно восстановить исходный код, скажем, на C++, в общем случае невозможно. Более эффективно декомпилируются программы в байт-кодах — например, существует довольно надёжный декомпилятор для Flash. Разновидностью декомпилирования является дизассемблирование машинного кода в код на языке ассемблера, который почти всегда выполняется успешно (при этом сложность может представлять самомодифицирующийся код или код, в котором собственно код и данные не разделены). Связано это с тем, что между кодами машинных команд и командами ассемблера имеется практически взаимно-однозначное соответствие.

Раздельная компиляция

Трансляция частей программы по отдельности с последующим объединением их компоновщиком в единый загрузочный модуль.

Исторически особенностью компилятора, отражённой в его названии (англ. *compile* — собирать вместе, составлять), являлось то, что он производил как трансляцию, так и компоновку, при этом компилятор мог порождать сразу абсолютный код. Однако позже, с ростом сложности и размера программ (и увеличением времени, затрачиваемого на перекомпиляцию), возникла необходимость разделять программы на части и выделять библиотеки, которые можно компилировать независимо друг от друга. При трансляции каждой части программы компилятор порождает объектный модуль, содержащий дополнительную информацию, которая потом, при компоновке частей в исполнимый модуль, используется для связывания и разрешения ссылок между частями.

Появление раздельной компиляции и выделение компоновки как отдельной стадии произошло значительно позже создания компиляторов. В связи с этим вместо термина «компилятор» иногда используют термин «транслятор» как его синоним: либо в старой литературе, либо когда хотят подчеркнуть его способность переводить программу в машинный код (и наоборот, используют термин «компилятор» для подчёркивания способности собирать из многих файлов один).

# Интерпретатор

**Интерпретатор** — программа (разновидность транслятора), выполняющая *интерпретацию*.

**Интерпретация** — пооператорный (покомандный, построчный) анализ, обработка и тут же выполнение исходной программы или запроса (в отличие от компиляции, при которой программа транслируется без её выполнения).

## Типы интерпретаторов

* **Простой интерпретатор** анализирует и тут же выполняет (собственно интерпретация) программу покомандно (или построчно), по мере поступления её исходного кода на вход интерпретатора. Достоинством такого подхода является мгновенная реакция. Недостаток — такой интерпретатор обнаруживает ошибки в тексте программы только при попытке выполнения команды (или строки) с ошибкой.
* **Интерпретатор компилирующего типа** — это система из компилятора, переводящего исходный код программы в промежуточное представление, например, в байт-код или p-код, и собственно интерпретатора, который выполняет полученный промежуточный код (так называемая виртуальная машина). Достоинством таких систем является большее быстродействие выполнения программ (за счёт выноса анализа исходного кода в отдельный, разовый проход, и минимизации этого анализа в интерпретаторе). Недостатки — большее требование к ресурсам и требование на корректность исходного кода. Применяется в таких языках, как Java, PHP, Tcl, Perl, REXX (сохраняется результат парсинга исходного кода[5]), а также в различных СУБД.

В случае разделения интерпретатора компилирующего типа на компоненты получаются компилятор языка и простой интерпретатор с минимизированным анализом исходного кода. Причём исходный код для такого интерпретатора не обязательно должен иметь текстовый формат или быть байт-кодом, который понимает только данный интерпретатор, это может быть машинный код какой-то существующей аппаратной платформы. К примеру, виртуальные машины вроде QEMU, Bochs, VMware включают в себя интерпретаторы машинного кода процессоров семейства x86.

Некоторые интерпретаторы (например, для языков Лисп, Scheme, Python, Бейсик и других) могут работать в режиме диалога или так называемого цикла чтения-вычисления-печати (англ. *read-eval-print loop, REPL*). В таком режиме интерпретатор считывает законченную конструкцию языка (например, s-expression в языке Лисп), выполняет её, печатает результаты, после чего переходит к ожиданию ввода пользователем следующей конструкции.

Уникальным является язык Forth, который способен работать как в режиме интерпретации, так и компиляции входных данных, позволяя переключаться между этими режимами в произвольный момент, как во время трансляции исходного кода, так и во время работы программ.

Следует также отметить, что режимы интерпретации можно найти не только в программном, но и аппаратном обеспечении. Так, многие микропроцессоры интерпретируют машинный код с помощью встроенных микропрограмм, а процессоры семейства x86, начиная с Pentium (например, на архитектуре Intel P6), во время исполнения машинного кода предварительно транслируют его во внутренний формат (в последовательность микроопераций).

## Алгоритм работы простого интерпретатора

1. прочитать инструкцию;
2. проанализировать инструкцию и определить соответствующие действия;
3. выполнить соответствующие действия;
4. если не достигнуто условие завершения программы, прочитать следующую инструкцию и перейти к пункту 2.

## Достоинства и недостатки

### Достоинства

* Бо́льшая переносимость интерпретируемых программ — программа будет работать на любой платформе, на которой есть соответствующий интерпретатор.
* Как правило, более совершенные и наглядные средства диагностики ошибок в исходных кодах.
* Меньшие размеры кода по сравнению с машинным кодом, полученным после обычных компиляторов.

### Недостатки

* Интерпретируемая программа не может выполняться отдельно без программы-интерпретатора. Сам интерпретатор при этом может быть очень компактным.
* Интерпретируемая программа выполняется медленнее, поскольку промежуточный анализ исходного кода и планирование его выполнения требуют дополнительного времени в сравнении с непосредственным исполнением машинного кода, в который мог бы быть скомпилирован исходный код.
* Практически отсутствует оптимизация кода, что приводит к дополнительным потерям в скорости работы интерпретируемых программ.

# Динамический компилятор

**JIT-компиляция** (англ. *Just-in-time compilation*, компиляция «на лету»), **динамическая компиляция** (англ. *dynamic translation*) — технология увеличения производительности программных систем, использующих байт-код, путём компиляции байт-кода в машинный код или в другой формат непосредственно во время работы программы. Таким образом достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом (сравнимая с компилируемыми языками) за счёт увеличения потребления памяти (для хранения результатов компиляции) и затрат времени на компиляцию. JIT базируется на двух более ранних идеях, касающихся среды исполнения: *компиляции байт-кода* и *динамической компиляции*.

Так как JIT-компиляция является, по сути, одной из форм динамической компиляции, она позволяет применять такие технологии, как адаптивная оптимизация и динамическая рекомпиляция. Из-за этого JIT-компиляция может показывать лучшие результаты в плане производительности, чем статическая компиляция. Интерпретация и JIT-компиляция особенно хорошо подходят для динамических языков программирования, при этом среда исполнения справляется с поздним связыванием типов и гарантирует безопасность исполнения.

Проекты LLVM, GNU Lightning, libJIT (часть проекта DotGNU) и RPython (часть проекта PyPy) могут быть использованы для создания JIT интерпретаторов любого скриптового языка.

## Особенности реализации

JIT-компиляция может быть применена как ко всей программе, так и к её отдельным частям. Например, текстовый редактор может на лету компилировать регулярные выражения для более быстрого поиска по тексту. С AOT-компиляции (компиляция перед исполнением) такое сделать не представляется возможным, так как данные предоставляются во время исполнения программы, а не во время компиляции. JIT используется в реализациях Java, JavaScript, .NET Framework, в одной из реализаций Python — PyPy.[3] Существующие наиболее распространённые интерпретаторы языков PHP, Ruby, Perl, Python и им подобных, которые имеют ограниченные или неполные JIT.

Большинство реализаций JIT имеют последовательную структуру: сначала (AOT) приложение компилируется в байт-код виртуальной машины среды исполнения, а потом JIT компилирует байт-код непосредственно в машинный код. В итоге приложение подтормаживает при запуске, что, впоследствии, компенсируется более быстрой его работой.

## Описание

В языках, таких как Java, PHP, C#, Lua, Perl, GNU CLISP, исходный код транслируется в одно из промежуточных представлений, называемое байт-кодом. Байт-код не является машинным кодом какого-либо конкретного компьютера и может переноситься на различные компьютерные архитектуры и исполнятся точно так же. Байт-код интерпретируется (исполняется) *виртуальной машиной*. JIT читает байткод из некоторых секторов (редко сразу из всех) и компилирует их в машинный код. Этим сектором может быть файл, функция или любой фрагмент кода. Однажды скомпилированный код может кэшироваться и в дальнейшем повторно использоваться без перекомпиляции.

Динамически компилируемая среда — это среда, в которой компилятор может вызываться приложением во время выполнения. Например, большинство реализаций Common Lisp содержат функцию compile, которая может создать функцию во время выполнения; в Python это функция eval. Это удобно для программиста, так как он может контролировать, какие части кода действительно подлежат компиляции. Также с помощью этого приёма можно компилировать динамически сгенерированный код, что в некоторых случаях приводит даже к лучшей производительности, чем реализация в статически скомпилированном коде. Однако стоит помнить, что подобные функции могут быть опасны, особенно когда данные передаются из недоверенных источников.

Основная цель использования JIT — достичь и превзойти производительность статической компиляции, сохраняя при этом преимущества динамической компиляции:

* Большинство тяжеловесных операций, таких как парсинг исходного кода и выполнение базовых оптимизаций происходит во время компиляции (до развёртывания), в то время как компиляция в машинный код из байт-кода происходит быстрее, чем из исходного кода
* Байт-код более переносим (в отличие от машинного кода)
* Среда может контролировать выполнение байт-кода после компиляции, поэтому приложение может быть запущено в песочнице (стоит отметить, что для нативных программ такая возможность тоже существует, но реализация данной технологии сложнее)
* Компиляторы из байт-кода в машинный код легче в реализации, так как большинство работы по оптимизации уже было проделано компилятором

JIT, как правило, эффективней, чем интерпретация кода. К тому же в некоторых случаях JIT может показывать большую производительность по сравнению со статической компиляцией за счёт оптимизаций, возможных только во время исполнения:

1. Компиляция может осуществляться непосредственно для целевого CPU и операционной системы, на которой запущено приложение. Например, JIT может использовать векторные SSE2 расширения процессора, если он обнаружит их поддержку. Однако, до сих пор нет основных реализаций JIT, где этот подход бы использовался, ведь что-бы обеспечить подобный уровень оптимизации, сравнимый со статическими компиляторами, потребовалось бы либо поддерживать бинарный файл под каждую платформу, либо включать в одну библиотеку оптимизаторы под каждую платформу.
2. Среда может собирать статистику о работающей программе и производить оптимизации с учётом этой информации. Некоторые статические компиляторы также могут принимать на вход информацию о предыдущих запусках приложения.
3. Среда может делать глобальные оптимизации кода (например, встраивание библиотечных функций в код) без потери преимуществ динамической компиляции и без накладных расходов, присущим статических компиляторам и линкерам.
4. Более простое перестраивание кода для лучшего использования кэша

## Задержка при запуске, средства борьбы с ней

Типичная причина задержки при запуске JIT-компилятора — расходы на загрузку среды и компиляцию приложения в байт-код. В общем случае, чем лучше и чем больше оптимизаций выполняет JIT, тем дольше получается задержка. Поэтому разработчикам JIT приходится искать компромисс между качеством генерируемого кода и временем запуска. Однако, часто оказывается так, что узким местом в процессе компиляции оказывается не сам процесс компиляции, а задержки системы ввода вывода (так, например, *rt.jar* в Java Virtual Machine (JVM) имеет размер 40MB, и поиск метаданных в нём занимает достаточно большое количество времени).

Ещё одно средство оптимизации — компилировать только те участки приложения, которые используются чаще всего. Этот подход реализован в PyPy и Sun’s HotSpot Java Virtual Machine.

В качестве эвристики может использоваться счётчик запусков участков приложения, размер байт-кода или детектор циклов.

Порой достаточно сложно найти правильный компромисс. Так, например, Sun’s Java Virtual Machine имеет два режима работы — клиент и сервер. В режиме клиента количество компиляций и оптимизаций минимально для более быстрого запуска, в то время как в режиме сервера достигается максимальная производительность, но из-за этого увеличивается время запуска.

Ещё одна техника, называемая pre-JIT, компилирует код до запуска. Преимуществом данной техники является ускоренное время запуска, в то время недостатком является плохое качество скомпилированного кода по сравнению с runtime JIT.

## История

Самую первую реализацию JIT можно отнести к LISP, написанную McCarthy in 1960[5]. В его книге *Recursive functions of symbolic expressions and their computation by machine, Part I,* он упоминает функции, компилируемые во время выполнения, тем самым избавив от надобности вывода работы компилятора на перфокарты.

Другой ранний пример упоминания JIT можно отнести к Кену Томпсону, который в 1968 году впервые применил регулярные выражения для поиска подстрок в текстовом редакторе QED. Для ускорения алгоритма Томпсон реализовал компиляцию регулярных выражений в машинный код IBM 7094.

Важный метод получения скомпилированного кода был предложен Митчелом в 1970 году, когда он реализовал экспериментальный язык *LC2*.[6][7]

Smalltalk (1983) был пионером в области JIT-технологий. Трансляция в машинный код выполнялась по требованию и кэшировалась для дальнейшего использования. Когда память кончалась, система могла удалить некоторую часть закэшированного кода из оперативной памяти и восстановить его, когда он снова потребуется. Язык программирования Self некоторое время был самой быстрой реализацией Smalltalk-а и работал всего-лишь в два раза медленней C, будучи полностью объектно-ориентированным.

Self был заброшен Sun, но исследования продолжились в рамках языка Java. Термин «Just-in-time компиляция» был заимствован из производственного термина «Точно в срок» и популяризован Джеймсом Гослингом, использовавшим этот термин в 1993.[8] В данный момент JIT используется почти во всех реализациях Java Virtual Machine.

Также большой интерес представляет диссертация, защищённая в 1994 году в Университете ETH (Швейцария, Цюрих) Михаэлем Францем «Динамическая кодогенерация — ключ к переносимому программному обеспечению»[9] и реализованная им система Juice[10] динамической кодогенерации из переносимого семантического дерева для языка Оберон. Система Juice предлагалась как плагин для интернет-браузеров.

## Безопасность

Так как JIT составляет исполняемый код из данных, возникает вопрос безопасности и возможных уязвимостей.

JIT компиляция включает в себя компиляцию исходного кода или байт-кода в машинный код и его выполнение. Как правило, результат записывается в память и исполняется сразу же, не используя диск и не вызывая код как отдельную программу. В современных архитектурах для повышения безопасности произвольные участки памяти не могут быть исполнены. Для корректной работы память должна быть помечена, как исполняемая (NX bit); для большей безопасности флаг должен быть поставлен *после* загрузки кода в память, а эта память должна быть помечена как доступная только для чтения, так как перезаписываемая и исполняемая память есть ничто иное, как дыра в безопасности.

# Смешение понятий трансляции и интерпретации

Понятия «трансляция» и «интерпретация» отличаются. Во время трансляция выполняется преобразование кода программы с одного языка на другой. Во время интерпретации программа исполняется.

Так как целью трансляции является, обычно, подготовка к интерпретации, эти процессы рассматриваются вместе. Например, языки программирования часто характеризуются как «компилируемые» или «интерпретируемые», в зависимости от того, что преобладает при использовании языка: компиляция или интерпретация. Причём, практически все языки низкого уровня и третьего поколения, вроде ассемблера, Си или Модулы-2, являются компилируемыми, а более высокоуровневые языки, вроде Python или SQL — интерпретируемыми.

С другой стороны, существует взаимопроникновение процессов трансляции и интерпретации: интерпретаторы могут быть компилирующими (в том числе с динамической компиляцией), а в трансляторах может требоваться интерпретация для реализации метапрограммирования (например, для макросов в языке ассемблера, условной компиляции в Си или шаблонов в C++).

Более того, один и тот же язык программирования может и транслироваться, и интерпретироваться, и в обоих случаях должны присутствовать общие этапы анализа и распознавания конструкций и директив исходного языка. Это относится и к программным реализациям, и к аппаратным — так, процессоры семейства x86 перед исполнением инструкций машинного языка выполняют их декодирование, выделяя в опкодах поля операндов (указание регистров, адресов в памяти, констант), разрядности и т. п., а в процессорах Pentium с архитектурой NetBurst тот же самый машинный код перед сохранением во внутреннем кэше дополнительно транслируется в последовательность микроопераций.