Содержание

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте *www.dmkpress.com*, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу *dmkpress@gmail.com*; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу *http://dmkpress.com/authors/publish_book/* или напишите в издательство по адресу *dmkpress@gmail.com*.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу *dmkpress@gmail.com*. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты *dmkpress@gmail.com*.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Вступительное слово Никлауса Вирта[1]

Мы не говорим на языках программирования – это формальные системы. Программы, которые транслируют тексты, написанные на языках программирования, в последовательность машинных команд, называются компиляторами. Это сложные программы. На заре развития компиляторов использовались языки Фортран (1957) и Алгол (1960). Над проектированием и реализацией компиляторов для них годами трудились большие коллективы программистов. Однако применение систематического подхода к конструированию компиляторов, описанного в этой книге, позволило сократить сроки до нескольких месяцев работы одного человека. Это был гигантский шаг вперед.

Принципиально новые универсальные языки программирования – редкость в наши дни, как и принципиально новые компьютерные архитектуры. Поэтому конструирование компиляторов кажется специализированной областью, в которой подвизаются немногие специалисты в крупных компаниях. Тогда зачем эта книга?

Причина проста. Любая программа, любое приложение строится по правилам, описывающим, как должны выглядеть предложения и объявления данных. Формализация этих правил делает приложение яснее и понятнее. Ключ ко всему – формализация, т. е. определение синтаксиса входных данных. Она позволяет производить синтаксический анализ, который лежит в основе конструирования компиляторов. Синтаксический анализ и другие методы, описанные в этой книге, полезен не только для обработки языков программирования, его можно применить и ко многим другим задачам, включающим систематическую обработку наборов данных или текстов, и к постоянно расширяющемуся множеству других структурированных входов. Эти методы вносят немалый вклад в правильность и понимание подобных приложений.

Пусть эта книга станет вашим помощником в этом многообещающем начинании!

Цюрих, Швейцария
Декабрь 2023

Заслуженный профессор, д-р Никлаус Вирт

[1] Профессор Никлаус Вирт скоропостижно скончался спустя неделю после того, как написал это вступительное слово, немного не дожив до своего девяностолетия.

Предисловие

Написать компилятор? Да ведь это под силу только таким крупным компаниям, как Microsoft, Google или Oracle, разве не так? Так-то оно так, но чуть ли не каждый специалист по информатике рано или поздно испытывает искушение спроектировать свой собственный язык программирования, пусть даже это всего лишь предметно-ориентированный язык или язык команд очень узкого назначения. И конечно, для этого языка нужен компилятор! Но эта мечта часто остается нереализованной из-за отсутствия знаний о конструировании компиляторов или же потому, что методы, описанные в учебниках, слишком сложны и зачастую касаются в основном таких продвинутых тем, как оптимизация, распределение регистров или детали генерирования кода.

По счастью, основы конструирования компиляторов просты – всякий может изучить их и включить в свой арсенал умений и навыков. Да, компиляторы для таких языков программирования, как Java или C/C++, действительно разрабатываются крупными компаниями, но есть много задач (даже выходящих за рамки собственно конструирования компиляторов), которые можно решить просто и элегантно, пользуясь лишь элементарными методами. В принципе, эти методы применимы всюду, где есть структурированные входные данные, которые можно описать грамматикой. Примерами могут служить простые языки команд, обработка конфигурационных файлов, журналов, списков деталей или рядов данных измерений.

В этой книге показано, как это делается. В ней рассматриваются практические основы конструирования компиляторов – от лексического и синтаксического анализа до семантической обработки и генерирования кода. Из других тем отметим описание процессов трансляции с помощью атрибутных грамматик и использование генератора компиляторов для автоматического генерирования основных частей компилятора. Эти две темы особенно важны на практике, хотя в большинстве книг по компиляторам не освещаются.

Для синтаксического анализа в этой книге используется идея рекурсивного спуска – простая техника нисходящего разбора, которую можно реализовать вручную (т. е. без применения инструментов). Но для полноты картины в конце книги представлен также восходящий синтаксический анализ – метод, более мощный, чем рекурсивный спуск, но и более сложный.

Любой метод можно полностью понять только в контексте конкретного примера, поэтому в качестве сквозного примера мы разработаем небольшой компилятор языка *MicroJava* – простого похожего на Java языка программирования, который транслируется в исполняемый байт-код, напоминающий байт-код Java. Полный исходный код этого компилятора можно скачать с со-

проводительного сайта [Download]. Сам компилятор, а также все примеры в этой книге написаны на Java.

Целевой машиной для компилятора является упрощенная *виртуальная Java-машина* (JVM), точнее *MicroJava Virtual Machine* (μJVM), не перегруженная излишними деталями, но в то же время достаточно реалистичная для иллюстрации и изучения методов генерирования кода. В роли системы команд μJVM выступает байт-код, основанный на байт-коде JVM. Для этого байт-кода предоставляется интерпретатор, позволяющий исполнять программы на компьютере. Изучив процесс генерирования кода, вы заодно узнаете, как работает компьютер, – еще одна причина потратить время на конструирование компилятора.

В конце каждой главы имеются упражнения. Решения некоторых из них можно найти на указанном ниже сайте [Download]. Рекомендуется проработать упражнения самостоятельно, так вы лучше усвоите описанные методы. Но даже если вы не сможете найти время для решения всех 70 с лишним упражнений, хотя бы ознакомьтесь с выборочными решениями, они послужат вам дополнительными примерами.

В основу книги положен курс по конструированию компиляторов, который я уже много лет читаю в университете им. Иоганна Кеплера в Линце и в Оксфордском университете Брукса в Англии. Ее можно использовать как учебник по начальному курсу конструирования компиляторов, за которым может последовать дополнительный курс, охватывающий такие темы, как оптимизация и распределение регистров. На сайте книги имеются также слайды, подготовленные для курса. Но книгу можно использовать и для самообразования, поскольку она ни от чего не зависит и в ней описываются все методы, необходимые для практического построения инструментов, напоминающих компилятор.

Хотел бы воспользоваться возможностью и отдать дань моему бывшему учителю и коллеге профессору Никлаусу Вирту (из Швейцарской высшей технической школы Цюриха), который написал вступительное слово к этой книге. Он был мастером конструирования компиляторов, у которого я научился многим методам. Также выражаю признательность издательству Springer-Verlag за поддержку этого проекта и д-ру Брайану Кирку, оказавшему огромную помощь в переводе книги с немецкого. Искреннее спасибо Дэвиду Лайтфуту и Полу Риду за помощь в уточнении перевода.

Линц, Австрия — Ханспетер МёссенБёк
Декабрь 2024

Сопроводительный сайт *https://ssw.jku.at/CompilerBook/*:

- слайды к лекциям;
- исходный код компилятора MicroJava;
- решения избранных упражнений;
- дополнительные материалы.

Глава 1

Общие сведения

Компилятор – это инструмент, который транслирует *исходную программу* (например, на языке Java, Pascal или C) в *целевую программу*. Языком целевой программы обычно является машинный код, например команды процессора или байт-код Java. Однако целью трансляции может быть и другой исходный язык. В таком случае компилятор называется *транспилятором*.

Помимо компиляторов в строгом смысле слова, имеются компиляторо-подобные инструменты, которые транслируют любой синтаксически структурированный вход в какую-то другую форму. Например, можно транслировать файл журнала в электронную таблицу, которая содержит фактически ту же информацию, но в форме, более удобной для последующего использования.

Вам, наверное, интересно, для чего могут пригодиться навыки построения компиляторов. Ведь настоящие компиляторы разрабатывают только крупные компании, такие как Microsoft, Google или Oracle. Для изучения компиляторов есть несколько причин:

- компиляторы – одни из самых часто используемых инструментов разработки программного обеспечения. Поэтому хорошо бы понимать, как они устроены и работают;
- изучая компиляторы, мы заодно узнаем, как работают компьютеры на машинном уровне. Мы должны разобраться с системой команд, регистрами, режимами адресации, а также с областями данных, например стеком вызовов и кучей. Это мост между оборудованием и программным обеспечением;
- зная, в какие команды транслируются те или иные языковые конструкции, вы начинаете лучше разбираться в вопросах эффективности программы;
- при конструировании компиляторов мы по необходимости имеем дело с грамматиками. Как и программирование, умение описывать структурированные данные с помощью грамматик является важным орудием в арсенале любого инженера-программиста.

Навыки конструирования компиляторов могут пригодиться и для программной инженерии вообще. И хотя лишь немногие занимаются созда-

нием компиляторов в строгом смысле слова, большинство программистов в какой-то момент сталкиваются с необходимостью разработать нечто похожее на компилятор. Это может быть разбор параметров командной строки, обработка списков деталей, языки описания документов (типа PDF или postscript) или средство статического анализа программ, которое вычисляет такие метрики, как сложность исходного кода. Во всех этих случаях умение конструировать компиляторы оказывается полезным.

В этой книге речь пойдет о конструировании компиляторов в строгом смысле, хотя рассмотренные методы применимы и к программной инженерии вообще. Описана полная реализация компилятора простого Java-подобного языка программирования (*MicroJava*), который генерирует исполняемый байт-код (похожий на байт-код Java). Книга рассчитана на читателей, проявляющих конкретный интерес к этому предмету. Формальная теория рассматривается только в той мере, которая необходима для понимания используемых методов. В книге сознательно опущены тонкие детали построения компиляторов, а включено лишь то, что требуется на практике. В частности, методы оптимизации едва затрагиваются, поскольку они легко могли бы составить полноценную книгу, а интересны лишь людям, пишущим компиляторы промышленного качества. Кроме того, рассматриваются только методы конструирования компиляторов императивных языков. Для функциональных и логических языков программирования необходимы другие методы, по крайней мере отчасти, но в практической программной инженерии они используются не так часто.

Существует много книг, посвященных продвинутому конструированию компиляторов (см., например, [ALSU06, Appe02, Coop22, FCL09, Much97]). В них в основном рассматриваются методы статического и динамического анализа кода, обширная тема оптимизации компиляторов и тонкости генерирования кода и распределения регистров. Эти книги интересны тем, кто хочет разрабатывать полноценные компиляторы для таких языков, как Java или C++. А для начинающих они зачастую слишком сложны.

В этой книге основное внимание уделено основам. Она написана на базе многолетнего курса по конструированию компиляторов и может использоваться в качестве учебника по этому предмету. Однако она также адресована программистам-практикам, которые хотят лучше понять, как работает компилятор, и включить соответствующие навыки в свой багаж знаний.

1.1. Краткая история компиляторов

Первые компиляторы появились в конце 1950-х годов. Тогда конструирование компиляторов было окутано тайной, а уровень ажиотажа можно сравнить с сегодняшним интересом к искусственному интеллекту. В то время лишь немногие что-то знали об этом, и широко известных методов компиляции языков программирования не существовало вовсе. Разработка пер-

вых компиляторов занимала много человеко-лет. Ныне конструирование компиляторов – один из наиболее изученных разделов информатики. Имеются хорошо отработанные методы синтаксического анализа, оптимизации и генерирования кода. В результате студенты могут реализовать (простой) компилятор в течение одного семестра.

В этой главе приводится краткий (и по необходимости неполный) очерк истории конструирования компиляторов, совпадающей с историей развития идей языков программирования. Изложение ведется на примере языков, которые считаются важными вехами и дают представление о развитии методов построения компиляторов в соответствующий период.

Fortran (1957). Одним из первых языков программирования высокого уровня стал Fortran [Back56], название которого является аббревиатурой от FORmula TRANslation. *Джон Бэкус*, работавший тогда в IBM, возглавил команду, которая поставила себе целью заменить язык ассемблера, превалировавший в то время, языком более высокого уровня и реализовать компилятор, который порождал бы машинный код, сравнимый по эффективности с написанными вручную ассемблерными программами. Многие основополагающие методы конструирования компиляторов были разработаны именно в то время. Например, в самом начале было далеко не очевидно, как транслировать арифметическое выражение (скажем, a + b * c) в машинные команды с сохранением стандартного предшествования (приоритета) арифметических действий. Кроме того, предстояло придумать методы трансляции условных предложений и циклов, а также механизмы вызова процедур и доступа к массивам.

1960: Algol. Algol60 [Naur64] стал знаменательным этапом в истории языков программирования. Он был разработан консорциумом специалистов из Европы и США, часть которых работала в университетах, а другая часть – в коммерческих компаниях. Была поставлена цель научиться выражать алгоритмы в ясной, краткой форме, допускающей машинное исполнение. Важными новшествами стали блочная структура с локальными и глобальными переменными, а также идея рекурсии, благодаря которой процедуры могли сохранять значения переменных в процессе рекурсивных вызовов. Algol60 был замечателен еще и тем, что стал первым языком программирования, синтаксис которого был формально определен грамматикой. *Джон Бэкус* и *Питер Наур* разработали для этой цели форму Бэкуса–Наура (БНФ), которая теперь входит в арсенал любого специалиста по компиляторам. БНФ сама по себе является языком и определяется грамматикой.

1970: Pascal. Преемником Algol60 стал Pascal [JW75], разработанный *Никлаусом Виртом* и *Тони Хоаром*. Хотя Pascal проектировался как простой язык для образовательных целей, он вобрал в себя важные новшества, которые нашли отражение и в конструкции его компилятора. Если в предыдущих языках использовались только предопределенные типы данных, такие как `integer` или `real`, то Pascal позволял программисту определять собственные типы данных, которые затем можно было использовать в объявлениях переменных. В дополнение к массивам – таблицам однородных данных появи-

лись записи (называемые также *структурами*), позволявшие группировать данные разных типов в новый тип. Хотя идея записей уже встречалась в языке Cobol, записи как тип данных появились только в Pascal. Для доступа к элементам таких структурных типов данных понадобилось разработать новые методы компиляции. Кроме того, первые компиляторы Pascal генерировали не машинный код, а так называемый «P-код», напоминающий современный байт-код Java. P-код был гораздо компактнее машинного кода, а это означало, что программы на Pascal можно было исполнять на тогдашних микрокомпьютерах с очень ограниченным объемом памяти. Эта идея способствовала распространению Pascal, но у нее был недостаток – P-код не мог быть выполнен непосредственно процессором, а должен был интерпретироваться, что замедляло работу программы.

1985: C++. C++ [Stro85] стал преемником языка C, который возник примерно в то же время, что и Pascal, и тоже воспринял идеи Algol60. В C++ соединились многие концепции языков программирования и построения компиляторов. Среди прочего были включены такие концепции, как *объектная ориентированность*, *обработка исключений* и *обобщенные* (т. е. параметризуемые) *типы данных*, которые уже существовали в других языках программирования, но популярность обрели только с появлением C++. Объектная ориентированность потребовала новых методов конструирования компиляторов, в частности представления иерархий наследования и трансляции вызовов виртуальных методов.

1995: Java. Java [AGH00] тоже был не совершенно новым языком, а конгломератом концепций существующих языков. Однако новым стал принцип *своевременной компиляции* (JIT-компиляции), благодаря которому Java-программы были переносимыми. Java-программа транслируется в команды байт-кода (похожего на Pascal'евский P-код), которые могут быть выполнены на любом компьютере, для которого имеется интерпретатор байт-кода. Однако наиболее часто выполняемые части программы транслируются в машинный код соответствующего компьютера во время выполнения (т. е. своевременно).

2005: Scala. Scala [Oder08] – язык, производный от Java, в качестве среды его выполнения выступает платформа Java (т. е. интерпретатор байт-кода и JIT-компилятор). Как и многие современные языки, Scala соединяет идеи функциональных языков с идеями императивного программирования. Функциональные языки включают *лямбда-выражения* (функции, которые рассматриваются как данные и могут передаваться другим функциям в качестве параметров), *ленивое выполнение* (откладывание вычисления выражения до того момента, когда в нем возникает необходимость) и *сопоставление с образцом* (распознавание образцов в последовательностях или древовидных структурах). Разумеется, для этих механизмов понадобились специальные методы конструирования компиляторов.

История языков программирования и компиляторов далека от завершения. Даже сегодня изобретаются новые концепции, которые ведут к новым методам компиляции. Но они выходят далеко за рамки этой книги, цель

которой – познакомить с фундаментальными методами конструирования компиляторов. Читатели, интересующиеся историей, могут найти информацию в трудах конференций по *истории языков программирования* ([HOPL-I, HOPL-II, HOPL-III]).

1.2. Динамическая структура компилятора

В качестве первого знакомства с принципами работы компиляторов рассмотрим их динамическую структуру, т. е. этапы компиляции (рис. 1.1).

Лексический анализ. Лексический анализ – первый этап компилятора. На нем поток литер исходной программы преобразуется в поток лексем. При этом отбрасываются незначащие литеры (например, пробелы), а остальные литеры объединяются в символы (например, имена, числа или операторы), которые называются *лексемами*. Каждому виду лексем сопоставляется уникальный код (например, 1 – идентификаторы, 2 – числа и т. д.). Для некоторых лексем имеется только код (например, для идентификации оператора `"*"` вполне достаточно кода лексемы. Для других лексем одним кодом не обойтись (например, существуют разные идентификаторы, скажем `"val"` и `"i"`). Поэтому в дополнение к коду (например, 1 = идентификатор) нам нужно еще и значение лексемы (например, `"val"`). Каждая лексема является объектом, который характеризуется кодом и факультативным значением. Поток лексем – абстракция потока литер, она упрощает последующие этапы компилятора. Часть компилятора, отвечающая за лексический анализ, называется *лексическим анализатором*, или *сканером*.

Синтаксический анализ. Следующий этап компилятора называется синтаксическим анализом. На нем поток лексем анализируется с точки зрения грамматики исходного языка и строится синтаксическое дерево, отражающее синтаксическую структуру исходной программы. Если все прошло без ошибок, то программа синтаксически правильна, и компилятор может переходить к следующему этапу. В противном случае имеется синтаксическая ошибка, компилятор сообщает о ней, а программист должен ее исправить и перекомпилировать программу. Часть компилятора, отвечающая за синтаксический анализ, называется *синтаксическим анализатором*, или *парсером*.

Семантический анализ. На этапе синтаксического анализа проверяется только синтаксическая правильность программы. Другие условия, например что все идентификаторы должны быть объявлены до использования или что типы данных операндов в операциях присваивания и в выражениях должны быть совместимы, проверяются на этапе семантического анализа. На этом этапе также строится *таблица символов*, в которой хранятся все объявленные имена и их свойства. Синтаксическое дерево и таблица символов образуют

следующую абстракцию исходной программы – промежуточное представление программы внутри компилятора.

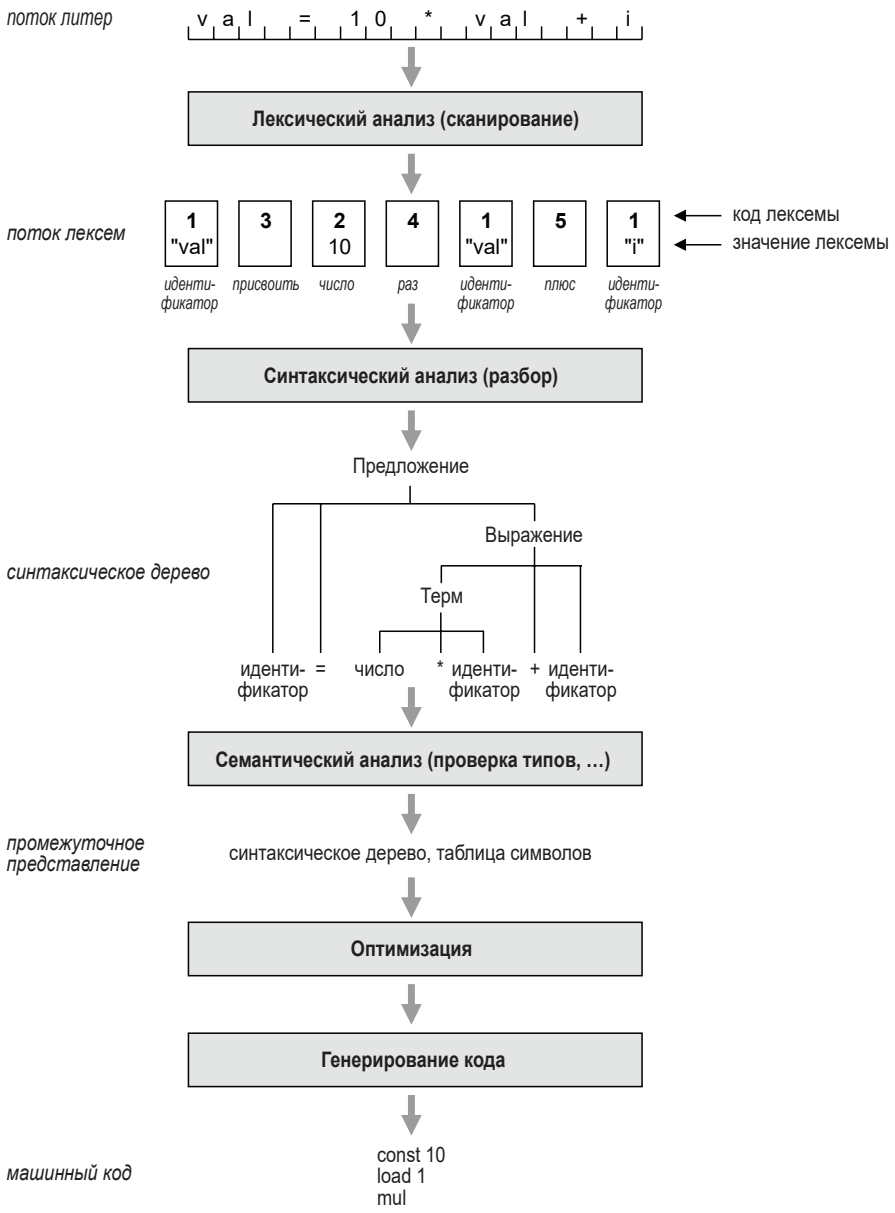

Рис. 1.1 ❖ Динамическая структура компилятора

Оптимизация. За семантическим анализом обычно следует этап оптимизации, на котором программа преобразуется в более быструю или в более компактную. Оптимизация – обширная тема, которой посвящены целые

книги, она важна для компиляторов промышленного качества. В этой книге оптимизация сознательно не рассматривается. Нас интересуют только базовые методы построения простых компиляторов или подобных им инструментов.

Генерирование кода. Последний этап компилятора – генерирование кода, когда по промежуточному представлению программы порождается код для целевой платформы.

1.2.1. Однопроходные и многопроходные компиляторы

Этапы компилятора можно объединять или выполнять строго последовательно. В первом случае компилятор называется однопроходным, а во втором – многопроходным.

В *однопроходном компиляторе* (рис. 1.2) сканер отдает следующую лексему, а парсер проверяет, допускает ли грамматика ее нахождение в текущей позиции. Затем лексему обрабатывает семантический анализатор, например проверяет типы, после чего кодогенератор порождает для нее машинный код. Далее цикл начинается сначала – и так, пока не будет откомпилирована вся программа.

Рис. 1.2 ❖ Однопроходный компилятор

В *многопроходном компиляторе* (рис. 1.3) этапы следуют друг за другом. Сканер читает исходный файл и генерирует поток лексем, парсер проверяет синтаксическую правильность этого потока и порождает синтаксическое дерево, семантический анализатор выполняет дальнейшие проверки и строит таблицу символов, после чего кодогенератор порождает код для целевой платформы.

В прошлом многопроходные компиляторы часто были необходимы, потому что объем компьютерной памяти был невелик или потому что язык был настолько сложен, что казалось обязательным разбивать компилятор