УДК 004.4'422

ТЕОРЕТИЧЕСКИЕ АСПЕКТЫ РАЗРАБОТКИ КОМПИЛЯТОРА ДЛЯ УЧЕБНЫХ ЦЕЛЕЙ

Кузнецов М. А., Хорольский А. В.

ФБГУ ВПО «Волгоградский государственный технический университет», Волгоград, Россия (400131, г. Волгоград, пр. Ленина, 28), e-mail: legion.dart@gmail.com

Статья посвящена принципам, по которым можно разработать интерактивное обучающее средство для курса «Основы трансляции». Сформулирована цель и поставлены задачи разработки. Далее проведен анализ существующих разработок настраиваемых компиляторов и их недостатки, а также приведен пример успешной реализации подобного учебного проекта и причины его несостоятельности. Рассмотрены этапы компиляции, которые необходимо рассмотреть при проектировании и разработке такого учебного программного комплекса, в частности, описание грамматики, лексический анализ, синтаксический разбор, семантический анализ, генерация кода, а также построение выводов для проверки заданного языка. В статье также рассмотрены теоретические аспекты и способы их практической реализации для достижения цели и сделан вывод о том, как подобный комплекс может быть, должен быть реализован для максимальной эффективности учебного процесса.

Ключевые слова:транслятор, компилятор, обучение, грамматика, языки программирования.

THEORETICAL ASPECTS OF DEVELOPING OF COMPILER FOR LEARNING THE BASICS OF TRANSLATION

Kuznetsov M. A., Khorolskiy A. V.

Volgograd State Technical University, Volgograd, Russia (400131, Volgograd, Lenina street, 28), e-mail: legion.dart@gmail.com

This paper is devoted to the principles on which you an interactive learning tool for the course "Fundamentals of translation" can be developed.The objective and the tasks of development have been formulated. Next, an analysis of existing examples of custom compilers and their limitations, as well as an example of successful implementation of this training project and the reasons for its failure. The stages of compilation, that need to be considered when designing and developing such an educational software package, in particular the description of grammar, lexical analysis, parsing, semantic analysis, code generation, and building terminals to check the given language. This article also discussed the theoretical aspects of the methods and their implementation to achieve the objectives and concluded that such complex may be implemented to maximize the efficiency of the learning process.

Key words: translator, compilator, education, grammatics, programming languages.

Введение

Процесс компиляции – один из важнейших процессов, происходящих при создании программного продукта. Понимание того, что при нем происходит – ключ к написанию наиболее хорошо оптимизированных и качественных приложений.

Курс «Основы трансляции» в большинстве вузов построен с помощью разделения процесса компиляции на этапы, которые подробнее будут рассмотрены ниже. В каждом этапе изучаются входные и выходные данные, а также основные алгоритмы, в нем задействованные. Суть алгоритмов практически не раскрывается, так как в этом нет необходимости; важен лишь принцип, по которому они действуют. Для более глубокого понимания смысла происходящих при компиляции процессов полезно личное участие студента в процессе описания параметров языка и компилятора.

Существует несколько программных средств для написания собственных компиляторов. Например, генератор лексических анализаторов LeX, используемый вместе с генератором парсеровYacc, интерактивный парсерGNUBison и лексический анализатор GNUFlex. Результатом является текст программы-компилятора на языке высокого уровня. Функциональные и учебные возможности этих программ ограничены лишь двумя функциями: лексическим и синтаксическим анализом, чего явно недостаточно. Кроме того, работа Bisonи Flex возможна только в POSIX-совместимых системах.

На кафедре ЭВМ и систем ВолгГТУ с 2002 года ведется разработка программного комплекса, представляющего собой настраиваемый пользователем компилятор. Но используемое сейчас программное средство морально устарело и уже не удовлетворяет современным требованиям к приложениям, таким как эргономичность, отказоустойчивость и дизайн, а также не выполняет важнейшую для компилятора функцию – генерацию исполняемого кода. Он реализует следующие этапы описания компилятора:

- задание грамматики в форме Бэкуса-Науэра;
- задание графа-распознавателя лексического анализатора;
- LL(1)-анализ грамматики;
- построение выводов;
- задание синтаксического анализатора (действий);
- проверка компилятора построением матрицы интерпретации.

Для поддержки курса требуются новые программные средства. Целью разработки является создание программного комплекса, который превосходит уже существующий по таким параметрам, как отказоустойчивость и эргономичность дизайна, а также позволит довести процесс компиляции до конца. Основная функция, которую выполняет разработка, – это синтез и проверка компилятора с языка программирования на основе ограниченных правил языков Си или Паскаль.

Рассмотрим основные этапы работы такого приложения подробнее.

Грамматики

Основой языков программирования высокого уровня являются порождающие грамматики. Порождающая грамматика G – это четверка (VT, VN, P, S), где: VT–алфавит терминальных символов (терминалов – базовых символов), VN- алфавит нетерминальных символов (НТ-символов или нетерминалов, определяемых символов), не пересекающийся с VT; P-множество правил вида $\alpha \rightarrow \beta$, где α–нетерминал, aβ–цепочка терминалов и нетерминалов;S - начальный символ (цель) грамматики, являющийся нетерминалом. Пример грамматики: G1 = ({0,1}, {A,S}, P, S), где P состоит из правил: $S \rightarrow 0A1$, $OA \rightarrow 00A1$, $A \rightarrow \lambda$.[1].

Одним из самых распространенных способов задания грамматики является форма Бэкуса-Науэра (далее БНФ), которая предполагает использование в качестве нетерминальных символов комбинаций слов естественного языка, заключённых в угловые скобки, а в качестве разделителя – специального знака, состоящего из двух двоеточий и равенства. Например, если правила L → EL и L → E записаны в символической форме, и символ L соответствует синтаксическому понятию «список», а символ E – «элемент списка», то их можно представить в форме Бэкуса-Наура так:

<список> ::= <элемент списка><список>,

<список> ::= <элемент списка>.

Чтобы сократить описание схемы грамматики, в БНФ разрешается объединять правила с одинаковой левой частью в одно правило, правая часть которого должна включать правые части объединяемых правил, разделённые вертикальной чертой. Используя объединение правил, для рассматриваемого примера получаем:

<список>::=<элемент списка><список>|<элемент списка>.

Именно этот способ задания грамматики используется в разрабатываемом учебном комплексе. Вот пример двух правил в БНФ, использующихся в описании языка Си:

<prog>::=<opisanie> main ( ) { <define_list><operlist> } <opis_fun>

<mas> ::= [const] |

Форма Бэкуса-Науэра требует, чтобы были определены все символы языка, при этом количество правил не ограничено. Но в программном средстве обязательно нужно объединять правила, определяющие один и тот же нетерминальный символ, в одно; таким образом, количество правил должно совпадать с количеством НТ-символов [1].

На первом этапе работы с учебным компилятором студент определяет грамматику по логике работы синтезируемого им компилятора, но в дальнейшем процессе работы БНФ может измениться, для этого в разработке предусмотрена возможность сохранения и загрузки заданной грамматики. После задания грамматики она обязательно подлежит проверке на ошибки построения БНФ, например, неверная последовательность правил, неверное описание символа или присутствие необъявленного нетерминального символа. Ошибки целесообразно выводить на экран, чтобы пользователь мог их исправить.

Описание лексического анализатора

Важными задачами любого транслятора являются определение принадлежности имеющегося текста программы к данному языку (т.е. не содержит ли он ошибок), а также отдельных его частей к определенным классам (лексемам). В отношении исходной программы транслятор выступает как распознаватель (лексический анализатор), а человек, создавший программу на некотором языке, выступает в роли генератора цепочек этого языка.

Распознаватель работает с символами своего алфавита – алфавита распознавателя. Алфавит распознавателя конечен и включает в себя все допустимые символы входных цепочек, а также некоторый дополнительный алфавит символов, которые могут обрабатываться устройством управления и храниться в рабочей памяти распознавателя.

Анализатор – это конечный автомат, поэтому для его описания чаще всего используют граф. Пример такого графа приведен на рис. 1.

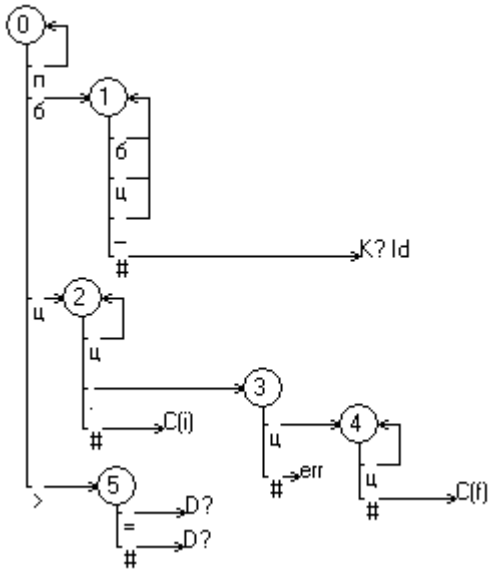

Рис. 1. Пример графа автомата-распознавателя

В таком графе вершины – состояния автомата – бывают трех видов: начальная (нулевая), промежуточные (обозначены кружками с цифрами) и результат (определяет класс, к которому относится обрабатываемая лексема, или сигнализирует об ошибке), а дугами (переходами) являются литеры или классы литер (буква, цифра, разделитель) [2].

Также целесообразно провести проверку лексического анализатора, так как он должен безошибочно распознавать лексемы, относя их к определенному классу. Пользователь может ввести цепочку символов, а программа должна выдать ему класс, к которому относится лексема (ключевое слово, идентификатор или константа).

Построение синтаксического LL(1)-анализатора

Синтаксический анализ (парсинг) – это процесс сопоставления линейной последовательности лексем языка с его формальной грамматикой, результатом которого, как правило, является дерево разбора (синтаксическое дерево). СА бывает нисходящего (от стартового символа до лексемы), восходящего типа (наоборот). В качестве нисходящего парсера наиболее часто используется LL-анализатор, проходящий текст программы слева направо. LL(1)-анализатором называется такой анализатор, который использует предпросмотр на одну лексему при разборе входного потока. Таким образом, LL(1)-

анализаторы просматривают поток только на один символ вперед при принятии решения о том, какое правило грамматики необходимо применить.

Для любого правила в LL(1)-грамматике характерно наличие так называемого направляющего символа. Так как разбор идет слева направо, самый первый в правиле символ является направляющим, и по нему распознается, какое правило должно быть использовано, и собственно, определяется текущий нетерминал, в том числе при выборе из двух альтернатив в одном правиле. Рассмотрим пример:

<A> ::= <B><C> |
<B><D> (1)

Грамматика, в которую входит такое правило, не может быть разобрана LL(1)-анализатором, потому что в обоих вариантах правила слева находится один и тот же символ B. Чтобы привести грамматику к LL(1)-виду, придется ввести дополнительное правило:

<A> ::= <B><E>
<E>::=<C> | <D> (2)

Грамматика(2) уже является LL(1) грамматикой и может быть распознана анализатором. Операцию, описанную выше, должен производить пользователь, но метакомпилятор при переходе к синтаксическому анализатору должен указать ему, что грамматика не приведена к нужному виду [3].

Построение выводов

Для проверки синтаксического анализатора можно использовать левые выводы. Вывод – цепочка, которая может быть получена применением правил грамматики. Построение вывода заключается в поэтапном раскрытии правил путем выбора различных альтернатив в правилах грамматики. При этом теоретически можно построить любой возможный код, подходящий к заданному правилу, в нисходящем порядке раскрывая нетерминальные символы.

Процесс построения останавливается, когда последний в цепочке нетерминальный символ раскрыт полностью и в правиле не осталось других нераскрытых нетерминалов. Таким образом, проверяется конечность грамматики: если построение вывода зацикливается, и невозможно его завершить, то грамматика была построена неверно [5].

Построение семантического анализатора

Семантика программы – внутренняя модель (база данных) множества именованных объектов, с которыми работает программа, с описанием их свойств, характеристик и связей [1].

Описать семантику языка можно, задав действия, которые компилятор должен выполнять при прохождении через текст программы. Например, «Занесение элемента из буфера в стек», «Операция сложения», «Инициализировать цикл». Действия можно отмечать в тексте грамматики, и если они требуют операндов, то учитывая учебную ограниченность и простоту действий, их можно брать прямо из текста программы без дополнительных преобразований. Таким образом, пользователю будет достаточно лишь указать код действия в нужном месте.

Генерация кода

По завершении задания семантического анализатора построение компилятора студентом можно считать завершенным. Далее наступает его проверка, и логичнее всего при этом увидеть результат: исполняемый код. Этот этап не требует иного вовлечения студента в процесс, кроме контроля и наблюдения. Его можно разделить на два: построение матрицы интерпретации по введенной программе и компиляцию исполняемого кода [4].

Матрица интерпретации определяет последовательность элементарных операций, выполняемых программой в процессе работы, и их операндов. Это промежуточный этап перед генерацией объектного кода. Матрица интерпретации может быть однозначно преобразована в соответствующий ассемблерный код, и поэтому удобна для понимания правильности процесса компиляции [4].

Генерацию кода целесообразно выполнить посредством Ассемблера при использовании компиляторов TASMили MASM.

Заключение

Рассмотренный в статье компилятор кардинально отличается от существующих промышленных трансляторов и пакетов для построения компиляторов. От первых основное отличие заключается в ориентированности на управляемый предварительно заданной грамматикой процесс распознавания. Причем грамматика задается в процессе работы системы интерактивно. При этом, конечно, теряется эффективность трансляции, но появляется возможность изменения грамматики языка на лету. От второго класса программ существенное отличие в возможности контролировать и интерактивно управлять любым этапом компиляции. Компилятор является транслятором, поддерживающим обработку некоторого подмножества языков программирования. Поэтому его лучше назвать метакомпилятором. Метакомпилятор позволяет наглядно продемонстрировать для студентов все этапы трансляции. На каждом этапе есть возможность управления основными параметрами алгоритмов трансляции.

Список литературы

1. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. – М.: Мир, 1978. – Т. 1,2.
2. Гордеев А. В. Системное программное обеспечение / А. В. Гордеев, А. Ю. Молчанов. − СПб.: Питер, 2001.
3. Касьянов В. Н. Лекции по теории формальных языков, автоматов и сложности вычислений. – Новосибирск: НГУ, 1995.
4. Креншоу Д. Пишем компилятор [Электронный ресурс]. − Режим доступа: ftp://vt.ustu/pub/discip/spo/2004/books/krenshaw.pdf, свободный. – Яз.рус.
5. Системное программное обеспечение: Основы трансляции: конспект лекций / составители: А. Н. Карпушин, П. С. Макаров. – Ульяновск: УлГТУ, 2007. – 59 с.

Рецензенты:

Лукьянов Виктор Сергеевич, д-р техн. наук, профессор кафедры ЭВМ и систем Волгоградского государственного технического университета, г. Волгоград.

Камаев Валерий Алексеевич, д-р техн. наук, профессор, заведующий кафедрой «САПР и ПК» Волгоградского государственного технического университета, г. Волгоград.