Лабораторная работа № 3

Построение простейшего дерева вывода

Цель работы

Цель работы: изучение основных понятий теории грамматик простого и операторного предшествования, ознакомление с алгоритмами синтаксического анализа (разбора) для некоторых классов КС-грамматик, получение практических навыков создания простейшего синтаксического анализатора для заданной грамматики операторного предшествования.

Краткие теоретические сведения

Назначение синтаксического анализатора

По иерархии грамматик Хомского выделяют четыре основные группы языков (и описывающих их грамматик) [1, 3, 4, 7]. При этом наибольший интерес представляют регулярные и контекстно-свободные (КС) грамматики и языки. Они используются при описании синтаксиса языков программирования. С помощью регулярных грамматик можно описать лексемы языка – идентификаторы, константы, служебные слова и прочие. На основе КС-грамматик строятся более крупные синтаксические конструкции: описания типов и переменных, арифметические и логические выражения, управляющие операторы и, наконец, полностью вся программа на входном языке.

Входные цепочки регулярных языков распознаются с помощью конечных автоматов (КА). Они лежат в основе сканеров, выполняющих лексический анализ и выделение слов в тексте программы на входном языке. Результатом работы сканера является преобразование исходной программы в список или таблицу лексем. Дальнейшую ее обработку выполняет другая часть компилятора – синтаксический анализатор. Его работа основана на использовании правил КС-грамматики, описывающих конструкции исходного языка.

Синтаксический анализатор (синтаксический разборщик) – это часть компилятора, которая отвечает за выявление и проверку синтаксических конструкций входного языка. В задачу синтаксического анализатора входит:

- найти и выделить синтаксические конструкции в тексте исходной программы;
- установить тип и проверить правильность каждой синтаксической конструкции;
- представить синтаксические конструкции в виде, удобном для дальнейшей генерации текста результирующей программы.

Синтаксический анализатор – это основная часть компилятора на этапе анализа. Без выполнения синтаксического разбора работа компилятора

бессмысленна, в то время как лексический разбор, в принципе, не является обязательной фазой компиляции. Все задачи по проверке синтаксиса входного языка могут быть решены на этапе синтаксического разбора. Лексический анализатор только позволяет избавить сложный по структуре синтаксический анализатор от решения примитивных задач по выявлению и запоминанию лексем исходной программы.

Выходом лексического анализатора является таблица лексем. Эта таблица образует вход синтаксического анализатора, который исследует только один компонент каждой лексемы – ее тип. Остальная информация о лексемах используется на более поздних фазах компиляции при семантическом анализе, подготовке к генерации и генерации кода результирующей программы.

Синтаксический анализатор воспринимает выход лексического анализатора и разбирает его в соответствии с грамматикой входного языка. Однако в грамматике входного языка программирования обычно не уточняется, какие конструкции следует считать лексемами. Примерами конструкций, которые обычно распознаются во время лексического анализа, служат ключевые слова, константы и идентификаторы. Но эти же конструкции могут распознаваться и синтаксическим анализатором. На практике не существует жесткого правила, определяющего, какие конструкции должны распознаваться на лексическом уровне, а какие надо оставлять синтаксическому анализатору. Обычно это определяет разработчик компилятора исходя из технологических аспектов программирования, а также синтаксиса и семантики входного языка. Принципы взаимодействия лексического и синтаксического анализаторов были рассмотрены в лабораторной работе № 2.

В основе синтаксического анализатора лежит распознаватель текста исходной программы, построенный на основе грамматики входного языка. Как правило, синтаксические конструкции языков программирования могут быть описаны с помощью КС-грамматик; реже встречаются языки, которые могут быть описаны с помощью регулярных грамматик.

Главную роль в том, как функционирует синтаксический анализатор и какой алгоритм лежит в его основе, играют принципы построения распознавателей для КС-языков. Без применения этих принципов невозможно выполнить эффективный синтаксический разбор предложений входного языка.

Проблема распознавания цепочек КС-языков

Взаимодействие лексического и синтаксического анализаторов рассматривалось в предыдущей лабораторной работе, здесь же будут рассмотрены алгоритмы, лежащие в основе синтаксического анализа. Перед синтаксическим анализатором стоят две основные задачи: проверить правильность конструкций программы, которая представляется в виде уже выделенных слов входного языка, и преобразовать ее в вид, удобный для дальнейшей семантической (смысловой) обработки и генерации кода. Одним из способов такого представления является дерево синтаксического разбора.

Основой для построения распознавателей КС-языков являются автоматы с магазинной памятью – МП-автоматы – односторонние недетерминированные

распознаватели с линейно-ограниченной магазинной памятью (полная классификация распознавателей приведена в [1, 4, 3, 7]). Поэтому важно рассмотреть, как функционирует МП-автомат и как для КС-языков решается задача разбора – построение распознавателя языка на основе заданной грамматики. Далее рассмотрены технические аспекты, связанные с реализацией синтаксических анализаторов.

МП-автомат в отличие от обычного КА имеет стек (магазин), в который можно помещать специальные «магазинные» символы (обычно это терминальные и нетерминальные символы грамматики языка). Переход МП-автомата из одного состояния в другое зависит не только от входного символа, но и от одного или нескольких верхних символов стека. Таким образом, конфигурация автомата определяется тремя параметрами: состоянием автомата, текущим символом входной цепочки (положением указателя в цепочке) и содержимым стека.

При выполнении перехода МП-автомата из одной конфигурации в другую из стека удаляются верхние символы, соответствующие условию перехода, и добавляется цепочка, соответствующая правилу перехода. Первый символ цепочки становится верхушкой стека. Допускаются переходы, при которых входной символ игнорируется (и тем самым он будет входным символом при следующем переходе). Эти переходы называются ^-переходами. Если при окончании цепочки автомат находится в одном из заданных конечных состояний, а стек пуст, цепочка считается принятой (после окончания цепочки могут быть сделаны Х-переходы). Иначе цепочка символов не принимается.

МП-автомат называется недетерминированным, если при одной и той же его конфигурации возможен более чем один переход. В противном случае (если из любой конфигурации МП-автомата по любому входному символу возможно не более одного перехода в следующую конфигурацию) МП-автомат считается детерминированным (ДМП-автоматом). ДМП-автоматы задают класс детерминированных КС-языков, для которых существуют однозначные КС-грамматики. Именно этот класс языков лежит в основе синтаксических конструкций всех языков программирования, так как любая синтаксическая конструкция языка программирования должна допускать только однозначную трактовку [1–4, 7].

По произвольной КС-грамматике

$$\mathbf{G(VN,VT,P,}S\mathbf{),\ V = VT} \cup \mathbf{VN}$$

всегда можно построить недетерминированный МП-автомат, который допускает цепочки языка, заданного этой грамматикой [1–3, 7]. А на основе этого МП-автомата можно создать распознаватель для заданного языка.

Однако при алгоритмической реализации функционирования такого распознавателя могут возникнуть проблемы. Дело в том, что построенный МП-автомат будет, как правило, недетерминированным, а для МП-автоматов, в отличие от обычных КА, не существует алгоритма, который позволял бы преобразовать произвольный МП-автомат в ДМП-автомат. Поэтому программирование функционирования МП-автомата – нетривиальная задача.

Если моделировать его функционирование по шагам с перебором всех возможных состояний, то может оказаться, что построенный для тривиального МП-автомата алгоритм никогда не завершится на конечной входной цепочке символов при определенных условиях. Примеры таких МП-автоматов можно найти в [1, 3, 7].
Поэтому для построения распознавателя для языка, заданного КС-грамматикой, рекомендуется воспользоваться соответствующим математическим аппаратом и одним из существующих алгоритмов.

Виды распознавателей для КС-языков
Существуют несложные преобразования КС-грамматик, выполнение которых гарантирует, что построенный на основе преобразованной грамматики МП-автомат можно будет промоделировать за конечное время на основе конечных вычислительных ресурсов. Описание сути и алгоритмов этих преобразований можно найти в [1, 3, 7].
Эти преобразования позволяют строить два основных типа простейших распознавателей:
• распознаватель с подбором альтернатив;
• распознаватель на основе алгоритма «сдвиг-свертка».
Работу распознавателя с подбором альтернатив можно неформально описать следующим образом: если на верхушке стека МП-автомата находится нетерминальный символ A, то его можно заменить на цепочку символов a при условии, что в грамматике языка есть правило $A \rightarrow a$, не сдвигая при этом считывающую головку автомата (этот шаг работы называется «подбор альтернативы»); если же на верхушке стека находится терминальный символ a, который совпадает с текущим символом входной цепочки, то этот символ можно выбросить из стека и передвинуть считывающую головку на одну позицию вправо (этот шаг работы называется «выброс»). Данный МП-автомат может быть недетерминированным, поскольку при подборе альтернативы в грамматике языка может оказаться более одного правила вида $A \rightarrow a$, тогда функция $\delta(q,\lambda,A)$ будет содержать более одного следующего состояния – у МП-автомата будет несколько альтернатив.
Решение о том, выполнять ли на каждом шаге работы МП-автомата выброс или подбор альтернативы, принимается однозначно. Моделирующий алгоритм должен обеспечивать выбор одной из возможных альтернатив и хранение информации о том, какие альтернативы на каком шаге уже были выбраны, чтобы иметь возможность вернуться к этому шагу и подобрать другие альтернативы.
Распознаватель с подбором альтернатив является нисходящим распознавателем: он читает входную цепочку символов слева направо и строит левосторонний вывод. Название «нисходящий» дано ему потому, что дерево вывода в этом случае следует строить сверху вниз, от корня к концевым вершинам («листьям»).
Работу распознавателя на основе алгоритма «сдвиг-свертка» можно описать так: если на верхушке стека МП-автомата находится цепочка символов y, то

ее можно заменить на нетерминальный символ A при условии, что в грамматике языка существует правило вида A → y, не сдвигая при этом считывающую головку автомата (этот шаг работы называется «свертка»); с другой стороны, если считывающая головка автомата обозревает некоторый символ входной цепочки a, то его можно поместить в стек, сдвинув при этом головку на одну позицию вправо (этот шаг работы называется «сдвиг» или «перенос»).

Этот распознаватель потенциально имеет больше неоднозначностей, чем рассмотренный выше распознаватель, основанный на алгоритме подбора альтернатив. На каждом шаге работы автомата надо решать следующие вопросы:

• что необходимо выполнять: сдвиг или свертку;

• если выполнять свертку, то какую цепочку y выбрать для поиска правил (цепочка y должна встречаться в правой части правил грамматики);

• какое правило выбрать для свертки, если окажется, что существует несколько правил вида A → γ (несколько правил с одинаковой правой частью).

Для моделирования работы этого расширенного МП-автомата надо на каждом шаге запоминать все предпринятые действия, чтобы иметь возможность вернуться к уже сделанному шагу и выполнить эти же действия по-другому. Этот процесс должен повторяться до тех пор, пока не будут перебраны все возможные варианты.

Распознаватель на основе алгоритма «сдвиг-свертка» является восходящим распознавателем: он читает входную цепочку символов слева направо и строит правосторонний вывод. Название «восходящий» дано ему потому, что дерево вывода в этом случае следует строить снизу вверх, от концевых вершин к корню.

Функционирование обоих рассмотренных распознавателей реализуется достаточно простыми алгоритмами, которые можно найти в [3, 7]. Однако оба они имеют один существенный недостаток – время их функционирования экспоненциально зависит от длины входной цепочки n = |α|, что недопустимо для компиляторов, где длина входных программ составляет от десятков до сотен тысяч символов. Так происходит потому, что оба алгоритма выполняют разбор входной цепочки символов методом простого перебора, подбирая правила грамматики произвольным образом, а в случае неудачи возвращаются к уже прочитанной части входной цепочки и пытаются подобрать другие правила.

Существуют более эффективные табличные распознаватели, построенные на основе алгоритмов Эрли и Кока—Янгера—Касами [1, 3]. Они обеспечивают полиномиальную зависимость времени функционирования от длины входной цепочки (n3 для произвольного МП-автомата и n2 для ДМП-автомата). Это самые эффективные из универсальных распознавателей для КС-языков. Но и полиномиальную зависимость времени разбора от длины входной цепочки нельзя признать удовлетворительной.

Лучших универсальных распознавателей не существует. Однако среди всего типа КС-языков существует множество классов и подклассов языков, для

которых можно построить распознаватели, имеющие линейную зависимость времени функционирования от длины входной цепочки символов. Такие распознаватели называют линейными распознавателями КС-языков.

В настоящее время известно множество линейных распознавателей и соответствующих им классов КС-языков. Каждый из них имеет свой алгоритм функционирования, но все известные алгоритмы являются модификацией двух базовых алгоритмов – алгоритма с подбором альтернатив и алгоритма «сдвиг-свертка», рассмотренных выше. Модификации заключаются в том, что алгоритмы выполняют подбор правил грамматики для разбора входной цепочки символов не произвольным образом, а руководствуясь установленным порядком, который создается заранее на основе заданной КС-грамматики. Такой подход позволяет избежать возвратов к уже прочитанной части цепочки и существенно сокращает время, требуемое на ее разбор.

Среди всего множества можно выделить следующие наиболее часто используемые распознаватели:

• распознаватели на основе рекурсивного спуска (модификация алгоритма с подбором альтернатив);

• распознаватели на основе LL(1) – и LL(k) – грамматик (модификация алгоритма с подбором альтернатив);

• распознаватели на основе LR(0) – и LR(1) – грамматик (модификация алгоритма «сдвиг-свертка»);

• распознаватели на основе SLR(1) – и LALR(1) – грамматик (модификация алгоритма «сдвиг-свертка»);

• распознаватели на основе грамматик предшествования (модификация алгоритма «сдвиг-свертка»).

Алгоритмы функционирования всех перечисленных и ряда других линейных распознавателей описаны в [1–4, 7].

Построение синтаксического анализатора

Синтаксический анализатор должен распознавать весь текст исходной программы. Поэтому, в отличие от лексического анализатора, ему нет необходимости искать границы распознаваемой строки символов. Он должен воспринимать всю информацию, поступающую ему на вход, и либо подтвердить ее принадлежность входному языку, либо сообщить об ошибке в исходной программе.

Но, как и в случае лексического анализа, задача синтаксического анализа не ограничивается только проверкой принадлежности цепочки заданному языку. Необходимо оформить найденные синтаксические конструкции для дальнейшей генерации текста результирующей программы. Синтаксический анализатор должен иметь некий выходной язык, с помощью которого он передает следующим фазам компиляции информацию о найденных и разобранных синтаксических структурах. В таком случае он уже является не разновидностью МП-автомата, а преобразователем с магазинной памятью – МП-преобразователем [1, 2, 7].

Вопросы, связанные с представлением информации, являющейся результатом работы синтаксического анализатора, и с порождением на основе этой информации текста результирующей программы, рассмотрены в лабораторной работе № 4, поэтому здесь на них останавливаться не будем.

Построение синтаксического анализатора – это более творческий процесс, чем построение лексического анализатора. Этот процесс не всегда может быть полностью формализован.

Имея грамматику входного языка, разработчик синтаксического анализатора должен в первую очередь выполнить ряд формальных преобразований над этой грамматикой, облегчающих построение распознавателя. После этого он должен проверить, относится ли полученная грамматика к одному из известных классов КС-языков, для которых существуют линейные распознаватели. Если такой класс найден, можно строить распознаватель (если найдено несколько классов, следует выбрать тот, для которого построение распознавателя проще либо построенный распознаватель будет обладать лучшими характеристиками). Если же такой класс КС-языков найти не удалось, то разработчик должен попытаться выполнить над грамматикой некоторые преобразования, чтобы привести ее к одному из известных классов. Эти преобразования не могут быть описаны формально, и в каждом конкретном случае разработчик должен попытаться найти их сам (иногда преобразования имеет смысл искать даже в том случае, когда грамматика подпадает под один из известных классов КС-языков, с целью найти другой класс, для которого можно построить лучший по характеристикам распознаватель).

Сложностей с построением синтаксических анализаторов не существовало бы, если бы для КС-грамматик были разрешимы проблемы преобразования и эквивалентности. Но поскольку в общем случае это не так, то одним классом КС-грамматик, для которого существуют линейные распознаватели, ограничиться не удается. По этой причине для всех классов КС-грамматик существует принципиально важное ограничение: в общем случае невозможно преобразовать произвольную КС-грамматику к виду, требуемому данным классом КС-грамматик, либо же доказать, что такого преобразования не существует. То, что проблема неразрешима в общем случае, не говорит о том, что она не решается в каждом конкретном частном случае, и зачастую удается найти такие преобразования. И чем шире набор классов КС-грамматик с линейными распознавателями, тем проще их искать.

Только, когда в результате всех этих действий не удалось найти соответствующий класс КС-языков, разработчик вынужден строить универсальный распознаватель. Характеристики такого распознавателя будут существенно хуже, чем у линейного распознавателя: в лучшем случае удается достичь квадратичной зависимости времени работы распознавателя от длины входной цепочки. Такое бывает редко, поэтому все современные компиляторы построены на основе линейных распознавателей (иначе время их работы было бы недопустимо велико).

Часто одна и та же КС-грамматика может быть отнесена не к одному, а сразу к нескольким классам КС-грамматик, допускающих построение линейных распознавателей. Тогда необходимо решить, какой из нескольких возможных распознавателей выбрать для практической реализации.

Ответить на этот вопрос не всегда легко, поскольку могут быть построены два принципиально разных распознавателя, алгоритмы работы которых несопоставимы. В первую очередь речь идет именно о восходящих и нисходящих распознавателях: в основе первых лежит алгоритм подбора альтернатив, в основе вторых – алгоритм «сдвиг-свертка».

На вопрос о том, какой распознаватель – нисходящий или восходящий – выбрать для построения синтаксического анализатора, нет однозначного ответа. Эту проблему необходимо решать, опираясь на некую дополнительную информацию о том, как будут использованы или каким образом будут обработаны результаты работы распознавателя. Более подробно обсуждение этого вопроса можно найти в [1, 7].

Совет.

Следует вспомнить, что синтаксический анализатор– это один из этапов компиляции. И с этой точки зрения результаты работы распознавателя служат исходными данными для следующих этапов компиляции. Поэтому выбор того или иного распознавателя во многом зависит от реализации компилятора, от того, какие принципы положены в его основу.

Желание использовать более простой класс грамматик для построения распознавателя может потребовать каких-то манипуляций с заданной грамматикой, необходимых для ее преобразования к требуемому классу. При этом нередко грамматика становится неестественной и малопонятной, что в дальнейшем затрудняет ее использование для генерации результирующего кода. Поэтому бывает удобным использовать исходную грамматику такой, какая она есть, не стремясь преобразовать ее к более простому классу.

В целом следует отметить, что, с учетом всего сказанного, интерес представляют как левосторонний, так и правосторонний анализ. Конкретный выбор зависит от реализации конкретного компилятора, а также от сложности грамматики входного языка программирования.

В общем виде процесс построения синтаксического анализатора можно описать следующим образом:

1. Выполнить простейшие преобразования над заданной КС-грамматикой.
2. Проверить принадлежность КС-грамматики, получившейся в результате преобразований, к одному из известных классов КС-грамматик, для которых существуют линейные распознаватели.
3. Если соответствующий класс найден, взять за основу для построения распознавателя алгоритм разбора входных цепочек, известный для этого класса, если найдено несколько классов линейных распознавателей – выбрать из них один по своему усмотрению.

4. Иначе, если соответствующий класс по п. 2 не был найден или же найденный класс КС-грамматик не устраивает разработчиков компилятора – попытаться выполнить над грамматикой неформальные преобразования с целью подвести ее под интересующий класс КС-грамматик для линейных распознавателей и вернуться к п. 2.
5. Если же ни в п. 3, ни в п. 4 соответствующий распознаватель найти не удалось (что для современных языков программирования практически невозможно), необходимо использовать один из универсальных распознавателей.
6. Определить, в какой форме синтаксический распознаватель будет передавать результаты своей работы другим фазам компилятора (эта форма называется внутренним представлением программы в компиляторе).
Реализовать выбранный в п. 3 или 5 алгоритм с учетом структур данных, соответствующих п. 6.
В данной лабораторной работе в заданиях предлагаются грамматики, не требующие дополнительных преобразований. Кроме того, гарантировано, что все они относятся к классу КС-грамматик операторного предшествования, для которых существует известный алгоритм линейного распознавателя. Поэтому создание синтаксического распознавателя для выполнения лабораторной работы существенно упрощается.
Для грамматик, предложенных в заданиях, известно, что они относятся также к классам КС-грамматик LR(1) и LALR(1), для которых также существует известный алгоритм линейного распознавателя, но, по мнению автора, этот алгоритм более сложен (его описание можно найти в [1, 2, 7]). Однако желающие могут не согласиться с автором и использовать для выполнения лабораторной работы любой из этих классов.
После несложных преобразований эти же грамматики могут быть приведены к виду, удовлетворяющему требованиям алгоритма рекурсивного спуска (или алгоритма анализа для LL(1) – грамматик). Этот алгоритм тривиально прост, но для его реализации надо выполнить достаточно несложные неформальные преобразования над заданными грамматиками – автор оставляет эти преобразования для желающих попробовать свои силы.
Выполняющие лабораторную работу могут пойти любым из рекомендованных путей или построить иной синтаксический анализатор по своему усмотрению – в этом направлении их ничто не ограничивает.
В качестве основного пути выполнения лабораторной работы автор предлагает распознаватель на основе грамматик операторного предшествования, поэтому именно этот класс КС-грамматик далее рассмотрен более подробно (описания остальных известных классов и подклассов КС-грамматик можно найти в [1–3, 7]).

Грамматики предшествования
КС-языки делятся на классы в соответствии со структурой правил их грамматик. В каждом из классов налагаются дополнительные ограничения на допустимые правила грамматики. Одним из таких классов является класс

грамматик предшествования. Они используются для синтаксического разбора цепочек с помощью модификаций алгоритма «сдвиг-свертка».

Принцип организации распознавателя на основе грамматики предшествования исходит из того, что для каждой упорядоченной пары символов в грамматике устанавливается отношение, называемое отношением предшествования. В процессе разбора МП-автомат сравнивает текущий символ входной цепочки с одним из символов, находящихся на верхушке стека автомата. В процессе сравнения проверяется, какое из возможных отношений предшествования существует между этими двумя символами. В зависимости от найденного отношения выполняется либо сдвиг, либо свертка. При отсутствии отношения предшествования между символами алгоритм сигнализирует об ошибке.

Задача заключается в том, чтобы иметь возможность непротиворечивым образом определить отношения предшествования между символами грамматики. Если это возможно, то грамматика может быть отнесена к одному из классов грамматик предшествования.

Отношения предшествования будем обозначать знаками «=.», «<.» и «.>». Отношение предшествования единственно для каждой упорядоченной пары символов. При этом между какими-либо двумя символами может и не быть отношения предшествования – это значит, что они не могут находиться рядом ни в одном элементе разбора синтаксически правильной цепочки. Отношения предшествования зависят от порядка, в котором стоят символы, и в этом смысле их нельзя путать со знаками математических операций (хотя по внешнему виду они очень похожи) – они не обладают ни свойством коммутативности, ни свойством ассоциативности. Например, если известно, что Bi.> Bj, то не обязательно выполняется Bj <. Bi (поэтому знаки предшествования помечают специальной точкой: «=.», «<.», «.>»).

Метод предшествования основан на том факте, что отношения предшествования между двумя соседними символами распознаваемой строки соответствуют трем следующим вариантам:

• Bi <. Bi+1, если символ Bi+1 – крайний левый символ некоторой основы (это отношение между символами можно назвать «предшествует основе» или просто «предшествует»);

• Bi.> Bi+1, если символ Bi – крайний правый символ некоторой основы (это отношение между символами можно назвать «следует за основой» или просто «следует»);

• Bi =. Bi+1, если символы Bi и Bi+1 принадлежат одной основе (это отношение между символами можно назвать «составляют основу»).

Исходя из этих соотношений выполняется разбор входной строки для грамматик предшествования.

Суть принципа такого разбора поясняет рис. 3.1. На нем изображена входная цепочка символов αγβδ в тот момент, когда выполняется свертка цепочки γ. Символ α является последним символом подцепочки α, а символ b – первым символом подцепочки β. Тогда, если в грамматике удастся установить непротиворечивые отношения предшествования, то в процессе выполнения разбора по алгоритму «сдвиг-свертка» можно всегда выполнять сдвиг до тех

пор, пока между символом на верхушке стека и текущим символом входной цепочки существует отношение <. или =.. А как только между этими символами будет обнаружено отношение.>, сразу надо выполнять свертку. Причем для выполнения свертки из стека надо выбирать все символы, связанные отношением =. Все различные правила в грамматике предшествования должны иметь различные правые части – это гарантирует непротиворечивость выбора правила при выполнении свертки.

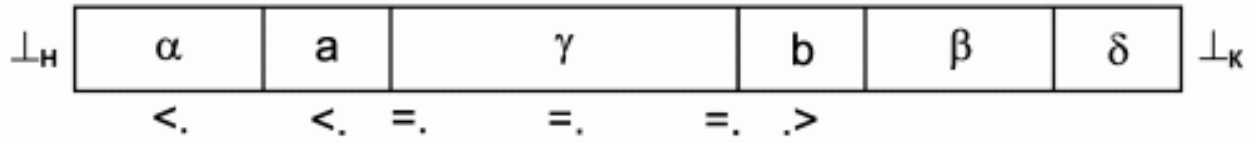


Рис. 3.1. Отношения между символами входной цепочки в грамматике предшествования.

Таким образом, установление непротиворечивых отношений предшествования между символами грамматики в комплексе с несовпадающими правыми частями различных правил дает ответы на все вопросы, которые надо решить для организации работы алгоритма «сдвиг-свертка» без возвратов.

На основании отношений предшествования строят матрицу предшествования грамматики. Строки матрицы предшествования помечаются первыми (левыми) символами, столбцы – вторыми (правыми) символами отношений предшествования. В клетки матрицы на пересечении соответствующих столбца и строки помещаются знаки отношений. При этом пустые клетки матрицы говорят о том, что между данными символами нет ни одного отношения предшествования.

Существует несколько видов грамматик предшествования. Они различаются по тому, какие отношения предшествования в них определены и между какими типами символов (терминальными или нетерминальными) могут быть установлены эти отношения. Кроме того, возможны незначительные модификации функционирования самого алгоритма «сдвиг-свертка» в распознавателях для таких грамматик (в основном на этапе выбора правила для выполнения свертки, когда возможны неоднозначности) [1]*Молчанов А. Ю. Системное программное обеспечение: Учебник для вузов. – СПб.: Питер, 2003. – 396 с.*

.

Выделяют следующие виды грамматик предшествования:

• простого предшествования;
• расширенного предшествования;
• слабого предшествования;
• смешанной стратегии предшествования;
• операторного предшествования.

Далее будут рассмотрены ограничения на структуру правил и алгоритмы разбора для грамматик операторного предшествования.

Матрицу операторного предшествования КС-грамматики можно построить, опираясь непосредственно на определения отношений предшествования [1, 3,

7], но проще и удобнее воспользоваться двумя дополнительными типами множеств – множествами крайних левых и крайних правых символов, а также множествами крайних левых терминальных и крайних правых терминальных символов для всех нетерминальных символов грамматики.

Если имеется КС-грамматика

$G(\mathbf{VT},\mathbf{VN},\mathbf{P},S)$, $\mathbf{V} = \mathbf{VT} \cup \mathbf{VN}$

то множества крайних левых и крайних правых символов определяются следующим образом:

$$\mathbf{L}(U) = \{T \mid \exists\ U \Rightarrow {}^{*}Tz\}$$

– множество крайних левых символов относительно нетерминального символа U;

$$\mathbf{R}(U) = \{T \mid \exists\ U \Rightarrow {}^{*}zT\}$$

– множество крайних правых символов относительно нетерминального символа U,

где U – заданный нетерминальный символ

$(U \in \mathbf{VN})$

T – любой символ грамматики

$(T \in \mathbf{V})$

а z – произвольная цепочка символов (

$z \in \mathbf{V}^{*}$

цепочка z может быть и пустой цепочкой).

Множества крайних левых и крайних правых терминальных символов определяются следующим образом:

$$\mathbf{L}_t(U) = \{t \mid \exists\ U \Rightarrow {}^{*}tz \text{ или } \exists\ U \Rightarrow {}^{*}Ctz\}$$

– множество крайних левых терминальных символов относительно нетерминального символа U;

$$\mathbf{R}_t(U) = \{t \mid \exists\ U \Rightarrow {}^{*}zt \text{ или } \exists\ U \Rightarrow {}^{*}ztC\}$$

– множество крайних правых терминальных символов относительно нетерминального символа U,

где t – терминальный символ

$(t \in \mathbf{VT})$

U и C – нетерминальные символы (U,

$C \in \mathbf{VN}$

а z – произвольная цепочка символов (

$z \in \mathbf{V}^{*}$

цепочка z может быть и пустой цепочкой).

Множества L(U) и R(U) могут быть построены для каждого нетерминального символа

$U \in \mathbf{VN}$

по очень простому алгоритму:

1. Для каждого нетерминального символа U ищем все правила, содержащие U в левой части. Во множество L(U) включаем самый левый символ из правой части правил, а во множество R(U) – самый правый символ из правой части (то есть во множество L(U) записываем все символы, с которых начинаются правила для символа U, а во множество R(U) – символы, которыми эти правила заканчиваются). Если в правой части правила для символа U имеется только один символ, то он должен быть записан в оба множества – L(U) и R(U).
2. Для каждого нетерминального символа U выполняем следующее преобразование: если множество L(U) содержит нетерминальные символы грамматики [U', U', …, то его надо дополнить символами, входящими в соответствующие множества L(U'), L(U')… и не входящими в L(U). Ту же операцию надо выполнить для R(U). Фактически, если какой-то символ U' входит в одно из множеств для символа U, то надо объединить множества для U' и U, а результат записать во множество для символа U.
3. Если на предыдущем шаге хотя бы одно множество L(U) или R(U) для некоторого символа грамматики изменилось, то надо вернуться к шагу 2, иначе – построение закончено.

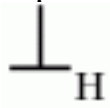
Для нахождения множеств Lt(U) и Rt(U) используется следующий алгоритм:

1. Для каждого нетерминального символа грамматики U строятся множества L(U) и R(U).
2. Для каждого нетерминального символа грамматики U ищутся правила вида U → tz и U → Ctz, где

$t \in \mathbf{VT}, C \in \mathbf{VN}, z \in \mathbf{V}^*$

терминальные символы t включаются во множество Lt(U). Аналогично для множества Rt(U) ищутся правила вида U → zt и U → ztC (то есть во множество Lt(U) записываются все крайние слева терминальные символы из правых частей правил для символа U, а во множество Rt(U) – все крайние справа терминальные символы этих правил). Не исключено, что один и тот же терминальный символ будет записан в оба множества – Lt(U) и Rt(U).
3. Просматривается множество L(U), в которое входят символы U', U' … Множество Lt(U) дополняется терминальными символами, входящими в Lt(U'), Lt(U')… и не входящими в Lt(U). Аналогичная операция выполняется и для множества Rt(U) на основе множества R(U).

Для практического использования матрицу предшествования дополняют терминальными символами



и

(начало и конец цепочки). Для них определены следующие отношения предшествования:

$\perp_{\text{н}} <\cdot\ a$, если $a \in \mathbf{L}_t(S)$;

$\perp_{\text{к}} \cdot> a$, если $a \in \mathbf{R}_t(S)$.

Имея построенные множества Lt(U) и Rt(U), заполнение матрицы операторного предшествования для КС-грамматики G(VT,VN,P,S) можно выполнить по следующему алгоритму:

1. Берем первый символ из множества терминальных символов грамматики VT:

$a_i \in \mathbf{VT},\ i = 1$

Будем считать этот символ текущим терминальным символом.

2. Во всем множестве правил P ищем правила вида C → xaiby или C → xaiUbjy, где ai – текущий терминальный символ, Ьj – произвольный терминальный символ

$b_j \in \mathbf{VT}$

U и C – произвольные нетерминальные символы

$(U, C \in \mathbf{VN})$

a x и y – произвольные цепочки символов, возможно пустые

$(x, y \in \mathbf{V}^*)$

Фактически производится поиск таких правил, в которых в правой части символы ai и Ъj стоят рядом или же между ними есть не более одного нетерминального символа (причем символ ai обязательно стоит слева от Ьj).

3. Для всех символов Ьj, найденных на шаге 2, выполняем следующее: ставим знак «=.» («составляет основу») в клетки матрицы операторного предшествования на пересечении строки, помеченной символом ai, и столбца, помеченного символом bj.

4. Во всем множестве правил P ищем правила вида C → xaiUjy, где ai – текущий терминальный символ, Uj и C– произвольные нетерминальные символы (Uj,

$C \in \mathbf{VN}$

a x и y – произвольные цепочки символов, возможно пустые

$x, y \in \mathbf{V}^*$

Фактически ищем правила, в которых в правой части символ ai стоит слева от нетерминального символа Uj.

5. Для всех символов Uj, найденных на шаге 4, берем множество символов Lt(Uj). Для всех терминальных символов ck, входящих в это множество, выполняем следующее: ставим знак «<.» («предшествует») в клетки матрицы операторного предшествования на пересечении строки, помеченной символом ai, и столбца, помеченного символом ck.

6. Во всем множестве правил P ищем правила вида C → xUjaiy, где ai – текущий терминальный символ, Uj и C – произвольные нетерминальные символы

$$U_j,\ C \in \mathbf{VN}$$

а x и y – произвольные цепочки символов, возможно пустые

$$x, y \in \mathbf{V}^*$$

Фактически ищем правила, в которых в правой части символ a стоит справа от нетерминального символа Uj.

7. Для всех символов Uj, найденных на шаге 6, берем множество символов Rt(Uj). Для всех терминальных символов ck, входящих в это множество, выполняем следующее: ставим знак «.>» («следует») в клетки матрицы операторного предшествования на пересечении строки, помеченной символом ck, и столбца, помеченного символом ai.

8. Если рассмотрены все терминальные символы из множества VT, то переходим к шагу 9, иначе – берем очередной символ

$$a_i \in \mathbf{VT}$$

из множества VT, i:= i + 1, делаем его текущим терминальным символом и возвращаемся к шагу 2.

9. Берем множество Lt(S) для целевого символа грамматики S. Для всех терминальных символов ck, входящих в это множество, выполняем следующее: ставим знак «<.» («предшествует») в клетки матрицы операторного предшествования на пересечении строки, помеченной символом

$$\perp_{\text{Н}}$$

(«начало строки»), и столбца, помеченного символом ck.

10. Берем множество Rt(S) для целевого символа грамматики S. Для всех терминальных символов ck, входящих в это множество, выполняем следующее: ставим знак «.>» («следует») в клетки матрицы операторного предшествования на пересечении строки, помеченной символом ck, и столбца, помеченного символом

$$\perp_{\text{К}}$$

(«конец строки»). Построение матрицы закончено.

Если на всех шагах алгоритма построения матрицы операторного предшествования не возникло противоречий, когда в одну и ту же клетку матрицы надо записать два или три различных символа предшествования, то матрица построена правильно (в каждой клетке такой матрицы присутствует один из символов предшествования – «=.», «<.» или «.>» – или же клетка пуста). Если на каком-то шаге возникло противоречие, значит, исходная КС-грамматика G(VT,VN,P,S) не является грамматикой операторного предшествования. В этом случае можно попробовать преобразовать грамматику так, что она станет удовлетворять требованиям операторного

предшествования (что не всегда возможно), либо необходимо использовать другой тип распознавателя.
Более подробно работа с грамматиками предшествования и другими типами распознавателей описана в [1–4, 7].

Алгоритм «сдвиг-свертка» для грамматик операторного предшествования
Алгоритм «сдвиг-свертка» для грамматики операторного предшествования выполняется МП-автоматом с одним состоянием. Для моделирования его работы необходима входная цепочка символов и стек символов, в котором автомат может обращаться не только к самому верхнему символу, но и к некоторой цепочке символов на вершине стека.
Этот алгоритм для заданной КС-грамматики G(VT,VN,P,S) при наличии построенной матрицы предшествования можно описать следующим образом:
1. Поместить в верхушку стека символ «начало строки», считывающую головку МП-автомата поместить в начало входной цепочки (текущим входным символом становится первый символ входной цепочки). В конец входной цепочки надо дописать символ «конец строки».
2. В стеке ищется самый верхний терминальный символ sj (если на вершине стека лежат нетерминальные символы, они игнорируются и берется первый терминальный символ, находящийся под ними), при этом сам символ sj остается в стеке. Из входной цепочки берется текущий символ ai (справа от считывающей головки МП-автомата).
3. Если символ sj – это символ начала строки, а символ ai – символ конца строки, то алгоритм завершен, входная цепочка символов разобрана.
4. В матрице предшествования ищется клетка на пересечении строки, помеченной символом sj, и столбца, помеченного символом ai (выполняется сравнение текущего входного символа и терминального символа на верхушке стека).
5. Если клетка, найденная на шаге 3, пустая, то значит, входная строка символов не принимается МП-автоматом, алгоритм прерывается и выдает сообщение об ошибке.
6. Если клетка, найденная на шаге 3, содержит символ «=.» («составляет основу») или «<.» («предшествует»), то необходимо выполнить перенос (сдвиг). При выполнении переноса текущий входной символ ai помещается на верхушку стека, считывающая головка МП-автомата во входной цепочке символов сдвигается на одну позицию вправо (после чего текущим входным символом становится следующий символ ai+1, i:= i+ 1). После этого надо вернуться к шагу 2.
7. Если клетка, найденная на шаге 3, содержит символ «.>» («следует»), то необходимо произвести свертку. Для выполнения свертки из стека выбираются все терминальные символы, связанные отношением «=.» («составляет основу»), начиная от вершины стека, а также все нетерминальные символы, лежащие в стеке рядом с ними. Эти символы вынимаются из стека и собираются в цепочку γ (если в стеке нет символов, связанных отношением

«=.», то из него вынимается один самый верхний терминальный символ и лежащие рядом с ним нетерминальные символы).
8. Во всем множестве правил P грамматики G(VT,VN,P,S) ищется правило, у которого правая часть совпадает с цепочкой γ (по условиям грамматик предшествования все правые части правил должны быть различны, поэтому может быть найдено или одно такое правило, или ни одного). Если правило найдено, то в стек помещается нетерминальный символ из левой части правила, иначе, если правило не найдено, это значит, что входная строка символов не принимается МП-автоматом, алгоритм прерывается и выдает сообщение об ошибке. Следует отметить, что при выполнении свертки считывающая головка автомата не сдвигается и текущий входной символ ai остается неизменным. После выполнения свертки необходимо вернуться к шагу 2.
После завершения алгоритма решение о принятии цепочки зависит от содержимого стека. Автомат принимает цепочку, если в результате завершения алгоритма он находится в состоянии, когда в стеке находятся начальный символ грамматики S и символ

Выполнение алгоритма может быть прервано, если на одном из его шагов возникнет ошибка. Тогда входная цепочка не принимается.
Алгоритм «сдвиг-свертка» для грамматики операторного предшествования игнорирует нетерминальные символы. Поэтому имеет смысл преобразовать исходную грамматику таким образом, чтобы оставить в ней только один нетерминальный символ. Это преобразование заключается в том, что все нетерминальные символы в правилах грамматики заменяются на один нетерминальный символ (чаще всего – целевой символ грамматики).
Построенная в результате такого преобразования грамматика называется остовной грамматикой, а само преобразование – остовным преобразованием [1, 7].
Остовное преобразование не ведет к созданию эквивалентной грамматики и выполняется только для упрощения работы алгоритма (который при выборе правил все равно игнорирует нетерминальные символы) после построения матрицы предшествования. Полученная в результате остовного преобразования грамматика может не являться однозначной, но все необходимые данные о порядке применения правил содержатся в матрице предшествования и распознаватель остается детерминированным. Поэтому остовное преобразование может выполняться без потерь информации только после построения матрицы предшествования. При этом также необходимо следить, чтобы в грамматике не возникло неоднозначностей из-за одинаковых правых частей правил, которые могут появиться в остовной грамматике. Вывод, полученный при разборе на основе остовной грамматики, называют результатом остовного разбора, или остовным выводом.

По результатам остовного разбора можно построить соответствующий ему вывод на основе правил исходной грамматики. Однако эта задача не представляет практического интереса, поскольку остовной вывод отличается от вывода на основе исходной грамматики только тем, что в нем отсутствуют шаги, связанные с применением цепных правил, и не учитываются типы нетерминальных символов. Для компиляторов же распознавание цепочек входного языка заключается не в нахождении того или иного вывода, а в выявлении основных синтаксических конструкций исходной программы с целью построения на их основе цепочек языка результирующей программы. В этом смысле типы нетерминальных символов и цепные правила не несут никакой полезной информации, а напротив, только усложняют обработку цепочки вывода. Поэтому для реального компилятора нахождение остовного вывода является даже более полезным, чем нахождение вывода на основе исходной грамматики. Найденный остовной вывод в дальнейших преобразованиях уже не нуждается.

В общем виде последовательность построения распознавателя для КС-грамматики операторного предшествования G(VT,VN,P,S) можно описать следующим образом:

1. На основе множества правил грамматики P построить множества крайних левых и крайних правых символов для всех нетерминальных символов грамматики

$$U \in \mathbf{VN}\text{: } \mathbf{L}(U) \text{ и } \mathbf{R}(U)$$

2. На основе множества правил грамматики P и построенных на шаге 1 множеств L(U) и R(U) построить множества крайних левых и крайних правых терминальных символов для всех нетерминальных символов грамматики

$$U \in VN$$

: Lt(U) и Rt(U).

3. На основе построенных на шаге 2 множеств Lt(U) и Rt(U) для всех терминальных символов грамматики

$$a \in \mathbf{VT}$$

заполняется матрица операторного предшествования.

4. Исходная грамматика G(VT,VN,P,S) преобразуется в остовную грамматику G'(VT,{S},P,S) с одним нетерминальным символом.

5. На основе построенной матрицы предшествования и остовной грамматики строится распознаватель на базе алгоритма «сдвиг-свертка» для грамматик операторного предшествования.

Важно, что алгоритм распознавателя может быть реализован вне зависимости от матрицы предшествования и правил исходной грамматики. Тогда, меняя матрицу и правила, один и тот же алгоритм можно использовать для распознавания входных цепочек любой грамматики операторного предшествования.

Далее в примере выполнения работы проиллюстрирован именно такой подход к построению распознавателя.

Требования к выполнению работы

Порядок выполнения работы

Для выполнения лабораторной работы требуется написать программу, которая выполняет лексический анализ входного текста в соответствии с заданием, порождает таблицу лексем и выполняет синтаксический разбор текста по заданной грамматике с построением дерева разбора. Текст на входном языке задается в виде символьного (текстового) файла. Синтаксис входного языка и перечень допустимых лексем указаны в задании. Допускается исходить из условия, что текст содержит не более одного предложения входного языка.

При наличии во входном файле текста, соответствующего заданному языку, программа должна строить и отображать дерево синтаксического разбора. Если же текст во входном файле содержит ошибки (лексические или синтаксические), программа должна выдавать сообщения о наличии ошибок во входном тексте и корректно завершать свое выполнение.

Рекомендуется разбить программу на три составные части: лексический анализ, построение цепочки вывода и построение дерева вывода. Лексический анализатор должен выделять в тексте лексемы языка и заменять их на терминальный символ грамматики (который в задании обозначен как a). Полученная после лексического анализа цепочка должна рассматриваться во второй части программы в соответствии с алгоритмом разбора. При неудачном завершении алгоритма выдается сообщение об ошибке, при удачном – строится цепочка вывода. После построения цепочки вывода на ее основе строится дерево разбора, в котором символы a последовательно заменяются на лексемы из таблицы лексем.

Для выполнения лексического анализа рекомендуется использовать программные модули, созданные в результате выполнения лабораторной работы № 2.

Длину идентификаторов и строковых констант можно считать ограниченной 32 символами. Программа должна допускать наличие комментариев неограниченной длины во входном файле. Форму организации комментариев предлагается выбрать самостоятельно.

1. Получить вариант задания у преподавателя.
2. Построить множества крайних левых и крайних правых символов, множества крайних правых и крайних левых терминальных символов и матрицу операторного предшествования для заданной грамматики (если для построения синтаксического распознавателя предполагается использовать другой механизм, отличный от грамматик операторного предшествования, то форму его надо предварительно согласовать с преподавателем).
3. Выполнить разбор простейшего примера вручную по правилам заданной грамматики, убедиться, что разбор выполняется корректно.
4. Подготовить и защитить отчет.
5. Написать и отладить программу на ЭВМ.
6. Сдать работающую программу преподавателю.

Требования к оформлению отчета

Отчет должен содержать следующие разделы:

• Задание по лабораторной работе.

• Краткое изложение цели работы.

• Запись заданной грамматики входного языка в форме Бэкуса—Наура (если для построения синтаксического распознавателя используется механизм, требующий преобразования исходной грамматики входного языка, то эти преобразования и полученная в результате их грамматика должны быть отражены в отчете).

• Множества крайних правых и крайних левых символов с указанием шагов построения.

• Множества крайних правых и крайних левых терминальных символов.

• Заполненную матрицу предшествования для грамматики (если для построения синтаксического распознавателя используется другой механизм, отличный от грамматик операторного предшествования, то форму его отображения в отчете надо согласовать с преподавателем).

• Пример выполнения разбора простейшего предложения входного языка.

• Текст программы (оформляется после выполнения программы на ЭВМ).

Основные контрольные вопросы

• Какую роль выполняет синтаксический анализ в процессе компиляции?

• Какие проблемы возникают при построении синтаксического анализатора и как они могут быть решены?

• Какие типы грамматик существуют? Что такое КС-грамматики? Расскажите об их использовании в компиляторе.

• Какие типы распознавателей для КС-грамматик существуют? Расскажите о недостатках и преимуществах различных типов распознавателей.

• Поясните правила построения дерева вывода грамматики.

• Что такое грамматики простого предшествования?

• Как вычисляются отношения предшествования для грамматик простого предшествования?

• Что такое грамматика операторного предшествования?

• Как вычисляются отношения для грамматик операторного предшествования?

• Расскажите о задаче разбора. Что такое распознаватель языка?

• Расскажите об общих принципах работы распознавателя языка.

• Что такое перенос, свертка? Для чего необходим алгоритм «перенос-свертка»?

• Расскажите, как работает алгоритм «перенос-свертка» в общем случае (с возвратами).

• Как работает алгоритм «перенос-свертка» без возвратов (объясните на своем примере)?

Варианты заданий

Варианты исходных грамматик

Далее приведены варианты грамматик. Во всех вариантах символ S является начальным символом грамматики; S, F, T и E обозначают нетерминальные символы.

Терминальные символы выделены жирным шрифтом. Вместо символа a должны подставляться лексемы.

1. S → a:= F;
F → F+T |T
T → T·E | TIE | E
E → (F) | – (F) | a
2. S → a:= F;
F → F or T | F xor T | T
T → T and E | E
E → (F) | not (F) | a
3. S → F;
F → if E then T else F| if E then F| a:= a
T → if E then T else T | a:= a
E → aa | a=a
4. S → F;
F → for (T) do F | a:= a
T → F;E;F |;E;F | F;E; |;E;
E → aa I a=a

Исходные грамматики и типы допустимых лексем

Ниже в табл. 3.1 приведены номера заданий. Для каждого задания указана соответствующая ему грамматика и типы допустимых лексем.

Таблица 3.1. Номера заданий для выполнения лабораторной работы

№	№ варианта грамматики	Допустимые лексемы входного языка
1	1	Идентификаторы, десятичные числа с плавающей точкой
2	2	Идентификаторы, константы true и false
3	3	Идентификаторы, десятичные числа с плавающей точкой
4	4	Идентификаторы, десятичные числа с плавающей точкой
5	1	Идентификаторы, римские числа
6	2	Идентификаторы, константы 0 и 1
7	3	Идентификаторы, римские числа

№	№ варианта грамматики	Допустимые лексемы входного языка
8	4	Идентификаторы, римские числа
9	1	Идентификаторы, шестнадцатеричные числа
10	2	Идентификаторы, шестнадцатеричные числа
11	3	Идентификаторы, шестнадцатеричные числа
12	4	Идентификаторы, шестнадцатеричные числа
13	1	Идентификаторы, символьные константы (в одинарных кавычках)
14	2	Идентификаторы, символьные константы 'T' и 'F'
15	3	Идентификаторы, строковые константы (в двойных кавычках)
16	4	Идентификаторы, строковые константы (в двойных кавычках)

Примечание.

• Римскими числами считать последовательности больших латинских букв X, V и I.

• Шестнадцатеричными числами считать последовательность цифр и символов «a», «Ь», «c», «d», «e» и «f», начинающуюся с цифры (например: 89, 45ac9, 0abc4).

• Для выполнения работы рекомендуется использовать лексический анализатор, построенный в ходе выполнения лабораторной работы № 2.

Пример выполнения работы

Задание для примера

Для выполнения лабораторной работы возьмем тот же самый язык, который был использован для выполнения лабораторной работы № 2.

Этот язык может быть задан, например, с помощью следующей КС-грамматики

G({if,then,else,a,=,or,xor,and,(,),},{S,F,E,D,C},P,S) с правилами P:

S → F;

F → if E then T else F | if E then F | a:= E

T → if E then T else T | a:= E

E → E or D | E xor D | D

D → D and C | C

C → a | (E)

Жирным шрифтом в грамматике и в правилах выделены терминальные символы.

Как было уже сказано ранее, выбранный в качестве примера язык не совпадает ни с одним из предложенных выше вариантов и, кроме этого, служит хорошей иллюстрацией основных особенностей построения синтаксического распознавателя, присущих различным вариантам.

Построение матрицы операторного предшествования

Построение множеств крайних правых и крайних левых символов

Построение множеств крайних левых и крайних правых символов выполним согласно описанному ранее алгоритму.
На первом шаге возьмем все крайние левые и крайние правые символы из правил грамматики G. Получим множества, представленные в табл. 3.2.
Таблица 3.2. Множества крайних левых и крайних правых символов. Шаг 1

Символ U	L(U)	R(U)
S	F	;
F	if, a	F, E
T	if, a	T, E
E	E, D	D
D	D, C	C
C	a, (	a,)

Из табл. 3.2 видно, что множества L(U) для символов S, E, D, а также множества R(U) для символов F, T, E, D содержат другие нетерминальные символы, а потому должны быть дополнены. Например, L(S) должно быть дополнено L(F), так как символ F входит в L(S): F є L(S), а R(F) должно быть дополнено R(E), так как символ E входит в R(F): E є R(F).
Выполним необходимые дополнения и получим множества, представленные в табл. 3.3.
Таблица 3.3. Множества крайних левых и крайних правых символов. Шаг 2

Символ U	L(U)	R(U)
S	F, if, a	;
F	if, a	F, E, D
T	if, a	T, E, D
E	E, D, C	D, C
D	D, C, a, (	C, a,)
C	a, (	a,)

Практически все множества в табл. 3.3 изменились по сравнению с табл. 3.2 (кроме множеств для символа C), а значит, построение не закончено. Продолжим дополнять множества. Получим множества, представленные в табл. 3.4.
В табл. 3.4 по сравнению с табл. 3.3 изменились множества для символов F, Г и E – построение не закончено. Продолжим дополнять множества. Получим множества, представленные в табл. 3.5.
Таблица 3.4. Множества крайних левых и крайних правых символов. Шаг 3

Символ U	L(U)	R(U)
S	F, if, a	;
F	if, a	F, E, D, C
T	if, a	T, E, D, C
E	E, D, C, a, (	D, C, a,)
D	D, C, a, (	C, a,)
C	a, (	a,)

Таблица 3.5. Множества крайних левых и крайних правых символов. Шаг 4 (результат)

Символ U	L(U)	R(U)
S	F, if, a	;
F	if, a	F, E, D, C, a,)
T	if, a	T, E, D, C, a,)
E	E, D, C, a, (	D, C, a,)
D	D, C, a, (	C, a,)
C	a, (	a,)

В табл. 3.5 по сравнению с табл. 3.4 изменились только множества R(U) для символов F иT– построение не закончено. Продолжим дополнять множества. Но если выполнить еще один шаг (шаг 5), то можно убедиться, что множества уже больше не изменятся (чтобы не создавать еще одну лишнюю таблицу, этот шаг здесь выполнять не будем). Таким образом, множества, представленные в табл. 3.5, являются результатом построения множеств крайних левых и крайних правых символов грамматики G.

Построение множеств крайних правых и крайних левых терминальных символов

Построение множеств крайних левых и крайних правых терминальных символов также выполним согласно описанному выше алгоритму.

На первом шаге возьмем все крайние левые и крайние правые терминальные символы из правил грамматики G. Получим множества, представленные в табл. 3.6.

Таблица 3.6. Множества крайних левых и крайних правых терминальных символов. Шаг 1

Символ U	Lt(U)	Rt(U)
S	;	;
F	if, a	else, then, :=
T	if, a	else, :=
E	or, xor	or, xor
D	and	and
C	a, (	a,)

Дополним множества, представленные в табл. 3.6, на основании ранее построенных множеств крайних левых и крайних правых символов, представленных в табл. 3.5. Например, Lt(E) должно быть дополнено Lt(D) и Lt(C), так как символы D и C входят в L(E): D, C e L(E), а Rt(F) должно быть дополнено Rt(E), Rt(D) и Rt(C), так как символы E, D и C входят в R(F): E, D, C e R(F).

Получим итоговые множества крайних левых и крайних правых терминальных символов, которые представлены в табл. 3.7.

Таблица 3.7. Множества крайних левых и крайних правых терминальных символов. Результат

Символ U	Lt(U)	Rt(U)
S	if, a, ;	;
F	if, a	else, then, :=, or, xor, and, a,)
T	if, a	else, := , or, xor, and, a,)
E	or, xor, and, a, (	or, xor, and, a,)
D	and, a, (	and, a,)
C	a, (	a,)

Теперь все готово для заполнения матрицы операторного предшествования.

Заполнение матрицы предшествования

Для заполнения матрицы операторного предшествования необходимы множества крайних левых и крайних правых терминальных символов, представленные в табл. 3.7, и правила исходной грамматики G.

Заполнение таблицы рассмотрим на примере лексем or и (.

Символ or не стоит рядом с другими терминальными символами в правилах грамматики. Поэтому знак «=.» («составляет основу») для него не используется. Символ or стоит слева от нетерминального символа D в правиле E → E or D. В множество Lt(D) входят символы and, a и (. Поэтому в строке матрицы, помеченной символом or, ставим знак «<.» («предшествует») в клетках на пересечении со столбцами, помеченными символами and, a и (.

Кроме того, символ or стоит справа от нетерминального символа E в том же правиле E → E or D. В множество Rt(E) входят символы or, xor, and, a и). Поэтому в столбце матрицы, помеченном символом or, ставим знак «.>» («следует») в клетках на пересечении со строками, помеченными символами or, xor, and, a и).

Больше ни в каких правилах символ or не встречается, поэтому заполнение матрицы для него закончено.

Символ (стоит рядом с терминальным символом) в правиле C → (E) (между ними должно быть не более одного нетерминального символа – в данном случае один символ E). Поэтому в строке матрицы, помеченной символом (, ставим знак «=.» («составляет основу») на пересечении со столбцом, помеченным символом).

Символ (также стоит слева от нетерминального символа E в том же правиле C → (E). В множество Lt(E) входят символы or, xor, and, a и (. Поэтому в строке матрицы, помеченной символом (, ставим знак «<.» («предшествует») в клетках на пересечении со столбцами, помеченными символами or, xor, and, a и (.

Больше ни в каких правилах символ (не встречается, поэтому заполнение матрицы для него закончено.

Повторяя описанные выше действия по заполнению матрицы для всех терминальных символов грамматики G, получим матрицу операторного предшествования. Останется только заполнить строку, соответствующую символу «начало строки», и столбец, соответствующий символу «конец строки».

Начальным символом грамматики G является символ S, поэтому для заполнения строки, помеченной ⊥н, возьмем множество Lt(S). В это множество входят символы if, а и;. Поэтому в строке матрицы, помеченной символом ⊥н, ставим знак «<.» («предшествует») в клетках на пересечении со столбцами, помеченными символами if, а и;.

Аналогично, для заполнения столбца, помеченного ⊥к, возьмем множество R^(S). В это множество входит только один символ —;. Поэтому в столбце матрицы, помеченном символом ⊥к, ставим знак «.>» («следует») в клетке на пересечении со строкой, помеченной символом;.

В итоге получим заполненную матрицу операторного предшествования, которая представлена в табл. 3.8.

Таблица 3.8. Матрица операторного предшествования

Символы	;	if	then	else	A	:=	or	xor	and	(	)	$\perp_к$
;												.>
if			=.		<.		<.	<.	<.	<.		
then	.>	<.		=.	<.							
else	.>	<.		.>	<.							
a	.>		.>	.>		=.	.>	.>	.>		.>	
:=	.>		.>	<.			<.	<.	<.	<.		
or	.>		.>	.>	<.		.>	.>	<.	<.	.>	
xor	.>		.>	.>	<.		.>	.>	<.	<.	.>	
and	.>		.>	.>	<.		.>	.>	.>	<.	.>	
(					<.		<.	<.	<.	<.	=.	
)	.>		.>	.>			.>	.>	.>		.>	
$\perp_н$	<.	<.			<.							

Теперь на основе исходной грамматики G можно построить остовную грамматику G'({if,then,else,a,=,or,xor,and,(,),},{E},P',E) с правилами P':

E → E; – правило 1;

E → if E then E else E | if E then E | a:= E – правила 2, 3 и 4;

E → if E then E else E | a:= E – правила 5 и 6;

E → E or E | E xor E | E – правила 7, 8 и 9;

E → E and E | E – правила 10 и 11;

E → a | (E) – правила 12 и 13.

Жирным шрифтом в грамматике и в правилах выделены терминальные символы.

Всего имеем 13 правил грамматики. Причем правила 2 и 5, а также правила 4 и 6 в остовной грамматике неразличимы, а правила 9 и 11 не имеют смысла (как было уже сказано, цепные правила в остовных грамматиках теряют смысл). То, что две пары правил стали неразличимы, не имеет значения, так как по смыслу (семантике входного языка) эти две пары правил обозначают одно и то же (правила 2 и 5 соответствуют полному условному оператору, а правила 9 и 11 – оператору присваивания). Поэтому в дереве синтаксического разбора нет необходимости их различать. Следовательно, синтаксический распознаватель может пользоваться остовной грамматикой G'.

Примеры выполнения разбора предложений входного языка

Рассмотрим примеры разбора цепочек входного языка в виде последовательности конфигураций МП-автомата, выполняющего разбор. Результат разбора будем представлять в виде последовательности номеров правил грамматики. На основе найденной последовательности правил после выполнения разбора при отсутствии ошибок (когда входная цепочка принята МП-автоматом) можно построить дерево синтаксического разбора.

Рассматриваемый МП-автомат имеет только одно состояние. Тогда для иллюстрации работы МП-автомата будем записывать каждую его конфигурацию в виде трех составляющих {α|β|γ}, где:

• α – непрочитанная часть входной цепочки;
• β – содержимое стека МП-автомата;
• γ – последовательность номеров примененных правил.

В начальном состоянии вся входная цепочка не прочитана, стек автомата содержит только лексему типа «начало строки», последовательность номеров правил пуста.

Для удобства чтения стек МП-автомата будем заполнять в порядке справа налево, тогда находящимся на верхушке стека будет считаться крайний правый символ в цепочке β.

Пример 1

Возьмем входную цепочку «if a or b and c then a:= 1 xor c;».

После выполнения лексического анализа, если все лексемы типа «идентификатор» и «константа» обозначить как «a», получим цепочку: «if a or a and a then a:= a xor a;».

Рассмотрим процесс синтаксического анализа этой входной цепочки. Шаги функционирования МП-автомата будем обозначать символом «÷». Символом «÷п» будем обозначать шаги, на которых выполняется сдвиг (перенос), символом «÷с» – шаги, на которых выполняется свертка.

{if a or a and a then a := a xor a;⊥к|⊥н |л} ч п
{a or a and a then a := a xor a;⊥к|⊥н if|л} ч п
{or a and a then a := a xor a;⊥к|⊥н if a|л} ч с
{or a and a then a := a xor a;⊥к|⊥н if E|12} ч п
{a and a then a := a xor a;⊥к|⊥н if E or|12} ч п
{and a then a := a xor a;⊥к|⊥н if E or a|12} ÷ с
{and a then a := a xor a;⊥к|⊥н if E or E|12 12} ÷ п
{a then a := a xor a;⊥к|⊥н if E or E and|12 12} ÷ п
{then a := a xor a;⊥к|⊥н if E or E and a|12 12} ÷ с
{then a := a xor a;⊥к|⊥н if E or E and E|12 12 12} ÷ с
{then a := a xor a;⊥к|⊥н if E or E|12 12 12 10} ÷ с
{then a := a xor a;⊥к|⊥н if E|12 12 12 10 7} ÷ п
{a := a xor a;⊥к|⊥н if E then|12 12 12 10 7} ч п
{:= a xor a;⊥к|⊥н if E then a|12 12 12 10 7} ч п
{a xor a;⊥к|⊥н if E then a :=|12 12 12 10 7} ч п
{xor a;⊥к|⊥н if E then a := a|12 12 12 10 7} ч с

{xor a;⊥к|⊥н if E then a := E|12 12 12 10 7 12} ч п
{a;⊥к|⊥н if E then a := E xor|12 12 12 10 7 12} ч п
{;⊥к|⊥н if E then a := E xor a|12 12 12 10 7 12} ч с
{;⊥к|⊥н if E then a:= E xor E|12 12 12 10 7 12} ÷ с
{;⊥к|⊥н if E then a := E|12 12 12 10 7 12 12 8} ÷ с
{;⊥к|⊥н if E then E|12 12 12 10 7 12 12 8 4} ÷ с
{;⊥к|⊥н E|12 12 12 10 7 12 12 8 4 3} ÷ п
{⊥к|⊥н E;|12 12 12 10 7 12 12 8 4 3} ÷ с
{⊥к|E⊥н |12 12 12 10 7 12 12 8 4 3 1}– разбор закончен, МП-автомат перешел в конечную конфигурацию, цепочка принята.

В результате получим последовательность правил: 12 12 12 107 12 12843 1. Этой последовательности правил будет соответствовать цепочка вывода на основе остовной грамматики С:

E→1 E; →3 if E then E; →4 if E then a := E; →8 if E then a := E xor E; →12 if E then a := E xor a; →12 if E then a := a xor a; →7 if E or E then a:= a xor a; →10 if E or E and E then a := a xor a; →12 if E or E and a then a := a xor a; →12 if E or a and a then a := a xor a; →12 if a or a and a then a := a xor a;

Стоит обратить внимание, что, так как данный МП-автомат строит правосторонний вывод, в цепочке вывода на каждом шаге правило всегда применяется к крайнему правому нетерминальному символу в цепочке.

Дерево синтаксического разбора, соответствующее данной входной цепочке, приведено на рис. 3.2.

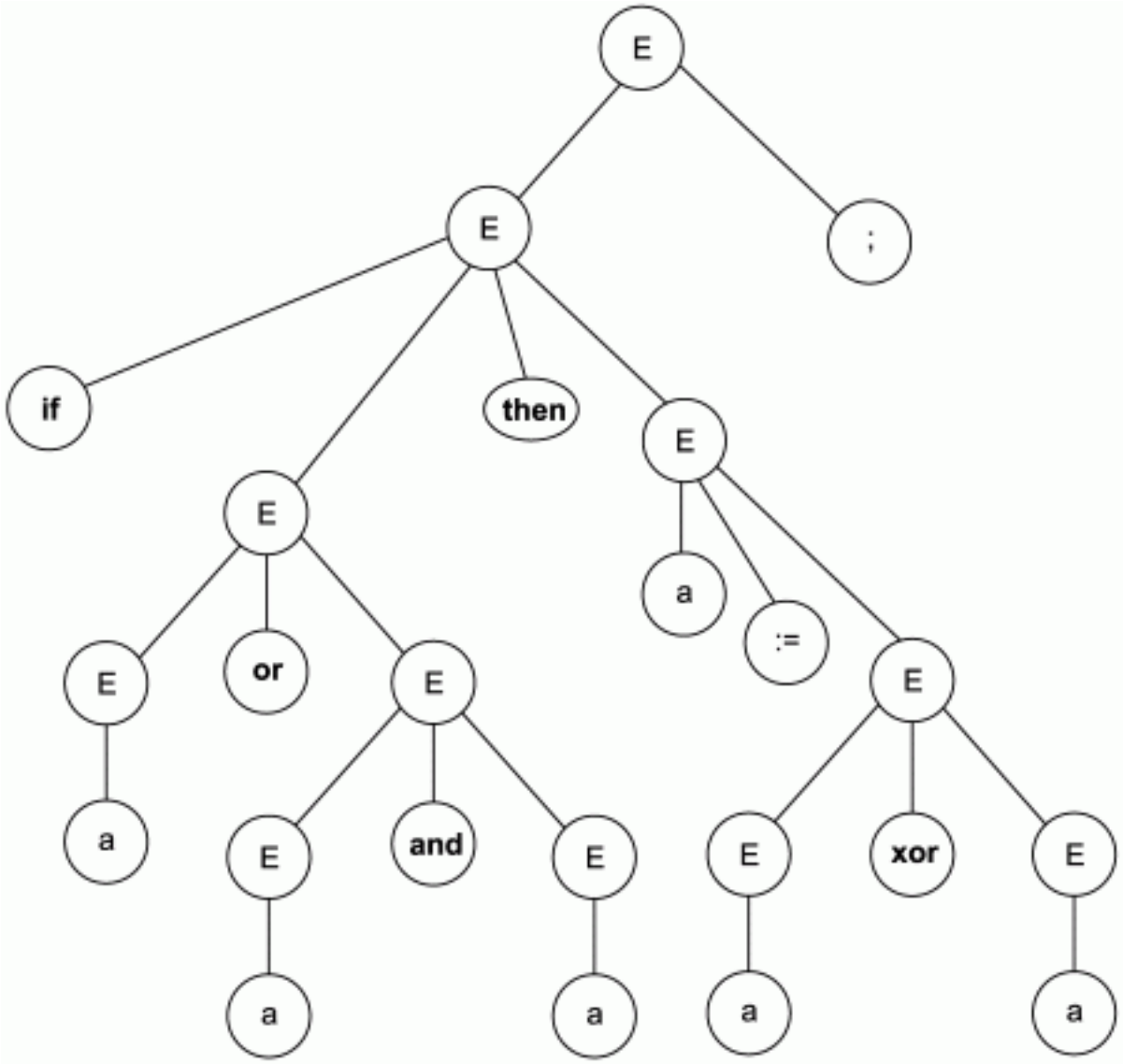


Рис. 3.2. Дерево синтаксического разбора входной цепочки «if a or a and a then a:= a xor a;».

Пример 2

Возьмем входную цепочку «if {a or b then a:= 25;».

После выполнения лексического анализа, если все лексемы типа «идентификатор» и «константа» обозначить как «а», получим цепочку: «if (a or a then a:= a».

Рассмотрим процесс синтаксического анализа этой входной цепочки:

{if (a or a then a := a;⊥к|⊥н |λ} ÷ п
{(a or a then a := a;⊥к|⊥н if|λ} ÷ п
{a or a then a := a;⊥к|⊥н if(|λ} ÷ п
{or a then a := a;⊥к|⊥н if(a|λ} ÷ с
{or a then a := a;⊥к|⊥н if(E|12} ÷ п
{a then a := a;⊥к|⊥н if(E or|12} ÷ п
{then a := a;⊥к|⊥н if(E or a|12} ÷ с
{then a := a;⊥к|⊥н if(E or E|12 12} ÷ с

{then a := a;⊥к|⊥н if(E|12 12 7} – нет отношения предшествования между лексемами «(» и «then», разбор закончен, МП-автомат не перешел в конечную конфигурацию, цепочка не принята (выдается сообщение об ошибке).

Реализация синтаксического распознавателя

Разбиение на модули

В лабораторной работе № 3, так же, как и в лабораторной работе № 2, модули, реализующие синтаксический анализатор разделены на две группы:

• модули, программный код которых не зависит от входного языка;

• модули, программный код которых зависит от входного языка.

В первую группу входят модули:

• SyntSymb – описывает структуры данных для синтаксического анализа и реализует алгоритм «сдвиг-свертка» для грамматик операторного предшествования;

• FormLab3 – описывает интерфейс с пользователем.

Во вторую группу входит один модуль:

• SyntRule – содержит описания матрицы операторного предшествования и правил исходной грамматики.

Такое разбиение на модули позволяет использовать те же самые структуры данных для организации синтаксического распознавателя при изменении входного языка.

Кроме этих модулей для реализации лабораторной работы № 3 используются программные модули TblElem и FncTree, позволяющие работать с комбинированной таблицей идентификаторов, которые были созданы при выполнении лабораторной работы № 1, а также модули LexType, LexElem, и LexAuto, которые обеспечивают работу лексического распознавателя (эти модули были созданы при выполнении лабораторной работы № 2).

Кратко опишем содержание программных модулей, используемых для организации синтаксического анализатора.

Модуль описания матрицы предшествования и правил грамматики

Модуль SyntRule содержит структуры данных, которые описывают матрицу операторного предшествования и правила остовной грамматики.

Матрица операторного предшествования (GramMatrix) описана как двумерный массив, каждой строке и каждому столбцу которого соответствует лексема (тип TLexType). Важно, чтобы данные в строках и столбцах матрицы были заполнены в том же порядке, в каком перечислены типы лексем в описании TLexType в модуле LexType. В каждой клетке матрицы находится символ, обозначающий тип отношения предшествования:

• < – для отношения «<.» («предшествует»);

• > – для отношения «.>» («следует»);

• = – для отношения «=.» («составляет основу»);

• – для пустых клеток матрицы (когда отношение операторного предшествования между двумя символами отсутствует).

Кроме матрицы операторного предшествования и правил грамматики в модуле SyntRule описана функция корректировки отношений

предшествования CorrectRul e, которая позволяет расширять возможности грамматики операторного предшествования. В данной лабораторной работе эта функция не используется (о технике ее использования можно узнать далее из описания примера выполнения курсовой работы).
В целом описанная в модуле SyntRule матрица операторного предшествования GramMatrix полностью соответствует построенной матрице операторного предшествования (см. табл. 3.8). Отличие заключается в том, что, поскольку терминальному символу a в грамматике G могут соответствовать два типа лексем входного языка (переменные и константы), в матрице GramMatrix строка и столбец, соответствующие символу a в табл. 3.8, продублированы.
Таким образом, построенный на основе матрицы предшествования из табл. 3.8 синтаксический анализатор не различает константы и переменные. Это соответствует синтаксису заданного входного языка. Для этого языка проводить различие между переменными и константами необходимо только в одном случае: при анализе оператора присваивания (присваивать значение константе нельзя). Для того чтобы компилятор находил такого рода ошибки, возможны два варианта:
1. Изменить синтаксис входного языка (грамматику G) так, чтобы константы и переменные различались в правилах грамматики, и перестроить синтаксический анализатор.
2. Обрабатывать присваивание значений константам на этапе семантического анализа.
В данном случае выбран второй вариант, который реализован в лабораторной работе № 4 (где рассматриваются генерация кода и подготовка к генерации кода). Позже, при разработке компилятора для выполнения курсовой работы, рассмотрен первый вариант (см. главу, посвященную выполнению курсовой работы). Каждый из рассмотренных вариантов имеет свои преимущества и недостатки. В общем случае выбор того, на каком этапе компиляции будет обнаружена та или иная ошибка, зависит от разработчика компилятора.
Правила остовной грамматики G' описаны в виде массива строк GramRules. Каждому правилу в этом массиве соответствует строка, по написанию совпадающая с правой частью правила (пробелы игнорируются). Правила пронумерованы в порядке слева направо и сверху вниз – так, как они были пронумерованы в остовной грамматике G. Для поиска подходящего правила используется метод простого перебора – так как правил мало (всего 13), в данном случае этот метод вполне удовлетворителен.
Кроме двух упомянутых структур данных (GramMatrix и GramRules) в модуле SyntRule описана также функция MakeSymbolStr, возвращающая наименование нетерминального символа в правилах остовной грамматики. В грамматике G во всех правилах символ обозначен E, поэтому функция MakeSymbolStr всегда возвращает 'E' как результат своего выполнения. Но тем не менее эта функция не бессмысленна, так как могут быть другие варианты остовных грамматик.
Модуль структур данных для синтаксического анализа и реализации алгоритма «сдвиг-свертка»

Модуль SyntSymb содержит реализацию алгоритма «сдвиг-свертка» и описания всех структур данных, необходимых для этой реализации. Поскольку сам алгоритм «сдвиг-свертка» не зависит от входного языка, реализующий его модуль также не зависит от входного языка и правил исходной грамматики (они специально вынесены в отдельный модуль).
Основу модуля составляют следующие структуры данных:
• TSymbInfo – описание двух типов символов грамматики: терминальных и нетерминальных;
• TSymbol – описание всех данных, связанных с понятием «символ грамматики»;
• TSymbStack – описание синтаксического стека.
Структура TSymbInfo содержит информацию о типе символа грамматики – поле SymbType, которое может принимать два значения: SYMBLEX (терминальный символ) или SYMBSYNT (нетерминальный символ), и дополнительные данные:
• ссылку на лексему (LexOne) – для терминального символа;
• перечень всех составляющих (LexList) – для нетерминального символа.
Перечень всех составляющих нетерминального символа LexList построен на основе динамического массива (тип TList из библиотеки VCL системы программирования Delphi 5). В него вносятся ссылки на символы, на основании которых создан данный символ, в том порядке, в котором они следуют в правиле грамматики.
Структура TSymbol содержит информацию о символе (поле SymbInfo типа TSymbInfo), а также номер правила грамматики, на основании которого создан символ (поле данных iRuleNum). Для терминальных символов номер правила равен 0, для нетерминальных символов он может быть от 1 до 13.
Кроме этих данных структура содержит методы, необходимые для работы с символами грамматики:
• конструктор CreateLex для создания терминального символа на основе лексемы;
• конструктор CreateSymb для создания нетерминального символа на основе правила грамматики и массива исходных символов;
• деструктор Destroy для освобождения занятой памяти при удалении символа (при удалении нетерминального символа удаляются все ссылки на его составляющие и динамический массив для их хранения);
• функции, процедуры и свойства для работы с информацией, хранящейся в структуре данных.
Поскольку в поле данных SymbInfo структуры TSymbol хранятся все ссылки на составляющие символы, внутри которых, в свою очередь, могут храниться ссылки на их составляющие и т. д., то на основе структуры TSymbol можно построить полное синтаксическое дерево разбора.
Третья структура данных TSymbStack построена на основе динамического массива типа TList из библиотеки VCL системы программирования Delphi 5. Она предназначена для того, чтобы моделировать синтаксический стек МП-автомата. В этой структуре нет никаких данных (используются только данные,

унаследованные от класса TList), но с ней связаны методы, необходимые для работы синтаксического стека:
• функция очистки стека (Clear) и деструктор для освобождения памяти при удалении стека (Destroy);
• функция доступа к символам в стеке начиная от его вершины (GetSymbol);
• функция для помещения в стек очередной входящей лексемы (Push), при этом лексема преобразуется в терминальный символ;
• функция, возвращающая самую верхнюю лексему в стеке (TopLexem), при этом нетерминальные символы игнорируются;
• функция, выполняющая свертку (MakeTopSymb); новый символ, полученный в результате свертки, помещается на вершину стека.
Кроме трех перечисленных ранее структур данных в модуле SyntSymb описана также функция Bui 1 dSyntList, моделирующая работу алгоритма «сдвиг-свертка» для грамматик операторного предшествования. Входными данными для функции являются список лексем (1 istLex), который должен быть заполнен в результате лексического анализа, и синтаксический стек (symbStack), который в начале выполнения функции должен быть пуст. Результатом функции является:
• нетерминальный символ (ссылающийся на корень синтаксического дерева), если разбор был выполнен успешно;
• терминальный символ, ссылающийся на лексему, где была обнаружена ошибка, если разбор выполнен с ошибками.
Функция BuildSyntList моделирует алгоритм «сдвиг-свертка» для грамматик операторного предшествования так, как он был описан в разделе «Краткие теоретические сведения».

Текст программы распознавателя
Кроме перечисленных выше модулей необходим еще модуль, обеспечивающий интерфейс с пользователем. Этот модуль (FormLab3) реализует графическое окно TLab3Form на основе класса TForm библиотеки VCL и включает в себя две составляющие:
• файл программного кода (файл FormLab3.pas);
• файл описания ресурсов пользовательского интерфейса (файл FormLab3.dfm).
Модуль FormLab3 построен на основе модуля FormLab2, который использовался для реализации интерфейса с пользователем в лабораторной работе № 2. Он содержит все данные, управляющие и интерфейсные элементы, которые были использованы в лабораторной работе № 2, поскольку первым этапом лабораторной работы № 3 является лексический анализ, который выполняется модулями, созданными для лабораторной работы № 2.
Кроме данных, используемых для выполнения лексического анализа так, как это было описано в лабораторной работе № 2, модуль содержит поле symbStack, которое представляет собой синтаксический стек, используемый для выполнения синтаксического анализа. Этот стек инициализируется при создании интерфейсной формы и уничтожается при ее закрытии. Он также

очищается всякий раз, когда запускаются процедуры лексического и синтаксического анализа.
Кроме органов управления, использованных в лабораторной работе № 2, интерфейсная форма, описанная в модуле FormLab3, содержит органы управления для синтаксического анализатора лабораторной работы № 3:
• в многостраничной вкладке (PageControl1) появилась новая закладка (SheetSynt) под названием «Синтаксис»;
• на закладке SheetSynt расположен интерфейсный элемент для просмотра иерархических структур (TreeSynt типа TTreeView).
Внешний вид новой закладки интерфейсной формы TLab3Form приведен на рис. 3.3.
Чтение содержимого входного файла организовано точно так же, как в лабораторной работе № 2.
После чтения файла выполняется лексический анализ, как это было описано в лабораторной работе № 2.
Если лексический анализ выполнен успешно, то в список лексем listLex добавляется информационная лексема, обозначающая конец строки, после чего вызывается функция выполнения синтаксического анализа BuildSyntList, на вход которой подаются список лексем (listLex) и синтаксический стек (symbStack). Результат выполнения функции запоминается во временной переменной symbRes.
Если переменная symbRes содержит ссылку на лексему, это значит, что синтаксический анализ выполнен с ошибками и эта лексема как раз указывает на то место, где была обнаружена ошибка. Тогда список строк входного файла позиционируется на указанное место ошибки, а пользователю выдается сообщение об ошибке.
Иначе, если ошибок не обнаружено, переменная symbRes указывает на корень построенного синтаксического дерева. Тогда в интерфейсный элемент TreeSynt записывается ссылка на корень синтаксического дерева, после чего все дерево отображается на экране с помощью функции MakeTree.
Функция MakeTree обеспечивает рекурсивное отображение синтаксического дерева в интерфейсном элементе типа TTreeView. Элемент типа TTreeView является стандартным интерфейсным элементом в ОС типа Windows для отображения иерархических структур (например он используется для отображения файловой структуры).

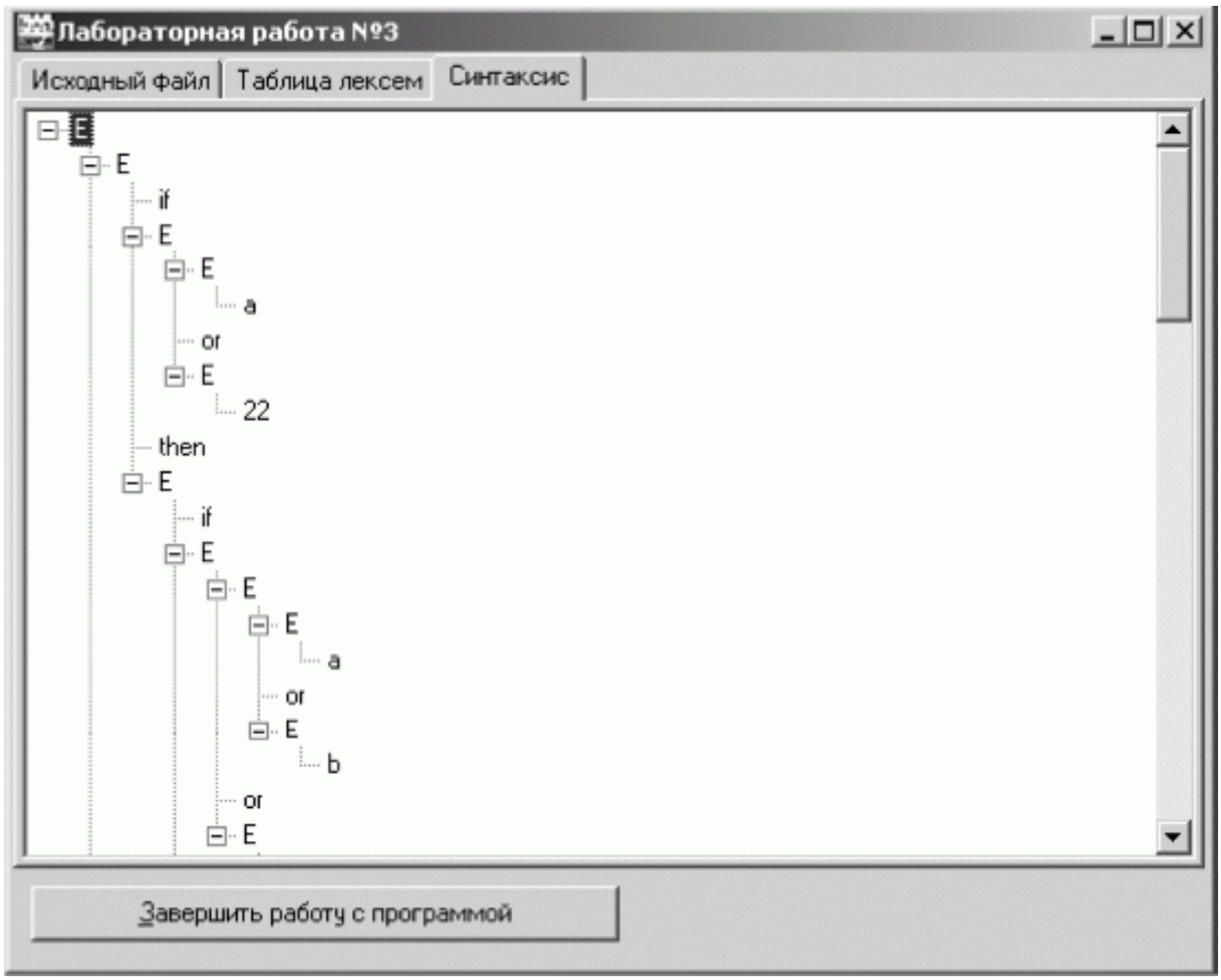


Рис. 3.3. Внешний вид третьей закладки интерфейсной формы для лабораторной работы № 3.

Полный текст программного кода модуля интерфейса с пользователем и описание ресурсов пользовательского интерфейса находятся в архиве, находящемся на веб-сайте издательства, в файлах FormLab3.pas и FormLab3.dfm соответственно.

Полный текст всех программных модулей, реализующих рассмотренный пример для лабораторной работы № 3, можно найти в архиве, находящемся на веб-сайте издательства, в подкаталогах LABS и COMMON (в подкаталог COMMON вынесены те программные модули, исходный текст которых не зависит от входного языка и задания по лабораторной работе). Главным файлом проекта является файл LAB3.DPR в подкаталоге LABS. Кроме того, текст модуля SyntSymb приведен в листинге П3.7 в приложении 3.

Выводы по проделанной работе

В результате лабораторной работы № 3 построен синтаксический анализатор на основе грамматики операторного предшествования. Синтаксический анализ позволяет проверять соответствие структуры исходного текста заданной грамматике входного языка. Синтаксический анализ позволяет обнаруживать любые синтаксические ошибки во входной программе. При наличии одной ошибки пользователю выдается сообщение с указанием местоположения ошибки в исходном тексте. Анализ типа обнаруженной

ошибки не производится. При наличии нескольких ошибок в исходном тексте обнаруживается только первая из них, после чего дальнейший анализ не выполняется.

Результатом работы синтаксического анализатора является структура данных, представляющая синтаксическое дерево. В комплексе с лексическим анализатором, созданным при выполнении лабораторной работы № 2, построенный синтаксический анализатор позволяет выполнять подготовку данных, необходимых для выполнения следующей лабораторной работы, связанной с генерацией кода.