Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

Е. В. Шостак, И. М. Марина, Д. Е. Оношко

ОСНОВЫ ПОСТРОЕНИЯ ТРАНСЛЯТОРОВ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве учебно-методического пособия для специальности 1-40 01 01 «Программное обеспечение информационных технологий»

Минск БГУИР 2019

УДК [004.4'4+004.423.2](076)
ББК 32.973.2я73
Ш79

Р е ц е н з е н т ы:

кафедра программной инженерии учреждения образования
«Белорусский государственный технологический университет»
(протокол №8 от 19.04.2018);

доцент кафедры информатики и веб-дизайна учреждения образования
«Белорусский государственный технологический университет»
кандидат технических наук, доцент А. А. Дятко;

доцент кафедры многопроцессорных систем и сетей
Белорусского государственного университета
кандидат физико-математических наук, доцент Е. Д. Рафеенко

Шостак, Е. В.

Ш79 Основы построения трансляторов языков программирования : учеб.-метод. пособие / Е. В. Шостак, И. М. Марина, Д. Е. Оношко. – Минск : БГУИР, 2019. – 66 с. : ил.

ISBN 978-985-543-470-3.

Содержит теоретический материал и задания для лабораторных работ по курсу «Языки программирования» в рамках действующей программы данного курса.

Издание можно использовать для самостоятельной работы студентов указанной специальности как очной, так и дистанционной форм обучения. Будет полезно студентам других специальностей.

УДК [004.4'4+004.423.2](076)
ББК 32.973.2я73

ISBN 978-985-543-470-3

Содержание

Введение

Несмотря на более чем полувековую историю вычислительной техники, формально годом рождения теории трансляторов можно считать 1957 год, когда появился первый компилятор языка Fortran, созданный Джоном Бэкусом и генерирующий достаточно эффективный объектный код. До этого времени создание компиляторов было весьма «творческим» процессом, и лишь появление теории формальных языков с сопутствующими математическими моделями сделало возможным переход от «творчества» к научному подходу и созданию сотен новых языков программирования. Более того, формальная теория компиляторов дала новый стимул развитию математической лингвистики и методам искусственного интеллекта, связанным с естественными и искусственными языками.

Основу теории трансляторов составляет *теория формальных языков*, являющаяся по своей сути методикой объяснения языка машине, – весьма сложный, насыщенный терминами, математическими моделями и прочими формализмами раздел математики.

Принципы и технологии разработки трансляторов находят применение не только в разработке компиляторов и интерпретаторов для компьютерных языков, но и при решении ряда других задач, связанных с обработкой текстовой информации. В настоящее время их применение стало особенно актуальным с ростом популярности такого направления, как разработка DSL (domain-specific languages) языков, которые (в отличие от языков программирования общего назначения) нацелены на решение относительно узкого круга задач, например, организацию обмена данными между составными частями программной системы или предоставление пользователю расширенных возможностей по автоматизации работы с программным средством.

Основы теории трансляторов

В общем случае под *транслятором* понимают программное средство (или модуль), выполняющее преобразование информации, записанной на некотором исходном языке, в эквивалентную запись этой информации на целевом языке. В роли такой информации чаще всего выступает программа, а исходным и целевым языками при этом могут быть языки программирования, объектный код, графическое представление и т. п. Другими примерами трансляторов являются, например, модуль веб-браузера, преобразующий описание веб-страницы на языке HTML (зачастую в комбинации с другими языками) в ее визуальное представление, или модуль системы управления базами данных, преобразующий запрос на языке SQL в конкретный набор операций, выполняемых над данными.

Исторически получили наибольшее распространение два вида трансляторов – компиляторы и интерпретаторы. Схематически отличия между ними проиллюстрированы на рис. 1.

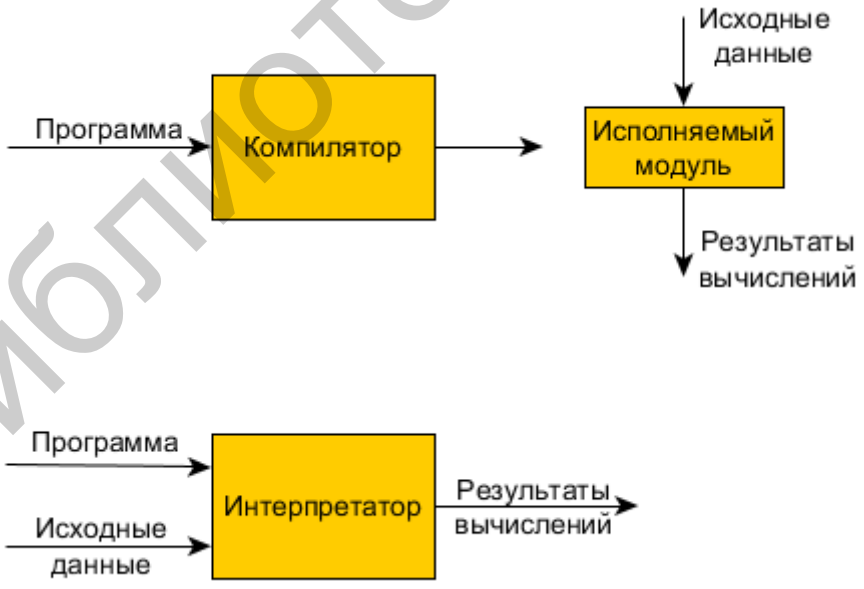


Рис. 1. Компиляторы и интерпретаторы

Компилятор – транслятор, для которого целевым является машинный язык. Другими словами, результатом его работы является модуль, содержащий объектный код, готовый к выполнению.

Интерпретатор – транслятор, который непосредственно выполняет операции, указанные в исходной программе, над исходными данными, предоставленными пользователем, вместо генерации эквивалентной программы.

Независимо от конкретного назначения все трансляторы, для которых исходным языком являются какие-либо текстовые данные, имеют схожую структуру и включают в себя пять основных этапов:

– лексический анализ;

– синтаксический анализ;

– семантический анализ;

– оптимизация;

– генерация кода (целевого представления).

Приведенные этапы принято также группировать в фазы анализа и синтеза. В англоязычной литературе также применяются соответственно понятия front-end и back-end. К анализу относят этапы лексического, синтаксического и семантического анализа, к синтезу – оптимизацию и генерацию кода.

Анализ предполагает разбиение исходной программы на составные части, наложение на них грамматической структуры, а также обнаружение лексических, синтаксических и семантических ошибок. В результате анализа формируется промежуточное представление исходной программы – чаще всего в виде древовидной структуры данных. Кроме того, в ходе анализа производится сбор дополнительной информации об элементах исходной программы и сохранение этой информации в специальной структуре данных – *таблице символов*.

В случае если обрабатываемая программа корректна с точки зрения правил исходного языка (что проверяется на этапах анализа), транслятор переходит к

этапам синтеза. В ходе *синтеза* происходит построение целевого представления программы на основе полученного при анализе промежуточного представления и информации из таблицы символов.

В зависимости от специфики задачи, а также ограничений платформы, для которой разрабатывается транслятор, его структура может незначительно отличаться. Например, в интерпретаторах место генерации кода может занимать непосредственное выполнение команд, а в компиляторах современных языков программирования этап оптимизации может разделяться на две части: до генерации кода выполняются машинно-независимые оптимизации, а после – машинно-зависимые.

На этапе *лексического анализа* производится выделение так называемых *лексем* – групп символов, имеющих в языке самостоятельное значение. В памяти транслятора лексемы представляют в виде специальных структур данных – *токенов*, содержащих как саму лексему (фрагмент текста), так и информацию о ней – тип лексемы, положение в тексте транслируемой программы и т. п.

На этапе *синтаксического анализа* происходит выявление структуры высказываний на исходном языке: вложенность операторов, структура выражений и т. п. Исходными данными для этого этапа является последовательность токенов, полученных на этапе лексического анализа, а результатом – так называемое *дерево разбора* – древовидная структура данных, представляющая структуру всей исходной программы или отдельных ее частей, а также взаимосвязь между ними до уровня отдельных лексем.

Зачастую построение дерева разбора осуществляется неявно, а вместо него строят *абстрактное синтаксическое дерево* – древовидную структуру данных, которая содержит структуру исходной программы в форме, не зависящей от вида использованной грамматики языка. Данная особенность позволяет сделать модули, реализующие последующие этапы трансляции, независимыми от этапа синтаксического анализа.

Древовидная структура данных, построенная синтаксическим анализатором, передается на этап *семантического анализа*. Задачей этого этапа является выявление ошибок в семантике программы (например, обнаружение несоответствий типов, отсутствия необходимых объявлений и т. д.). Кроме того, на этом этапе дерево разбора или абстрактное синтаксическое дерево может быть дополнено информацией, которую проблематично выделить на предыдущих этапах.

Задачей этапа оптимизации является получение (в виде внутреннего представления) программы, эквивалентной исходной, но имеющей лучшие характеристики производительности или потребления ресурсов. В зависимости от решаемой транслятором задачи этот этап может полностью отсутствовать (например, если задача транслятора – выполнить перевод программы с одного языка высокого уровня на другой).

Этап генерации кода заключается в построении программы на целевом языке по полученному на предыдущих этапах внутреннему представлению исходной программы.

Формальные языки и грамматики

Формальным языком называется множество строк, построенных из символов некоторого алфавита по некоторым правилам. *Алфавит языка* – множество символов, которые могут быть использованы для построения высказываний на этом языке.

Формальные языки могут задаваться различными способами (в том числе перечислением всех образующих язык строк), однако чаще всего для этих целей применяются *формальные грамматики* (или просто *грамматики*). Можно считать, что грамматика – это набор правил, по которым из символов алфавита языка строятся высказывания на этом языке.

Грамматика включает в себя четыре элемента:

– Σ – множество терминальных символов (*терминалов*);

– N – множество нетерминальных символов (*нетерминалов*);

– P – множество правил вида $\alpha \to \beta$, где α – непустая последовательность терминалов и нетерминалов, β – любая последовательность терминалов и нетерминалов;

– S – стартовый (начальный) символ, $S \in N$.

Как правило, терминальными символами Σ являются неделимые элементы языка (часто также называемые просто алфавитом языка), нетерминальными N – фрагменты высказываний формального языка, полученные путем комбинирования терминальных символов. При этом правила P можно рассматривать, как правила подстановки, а стартовый символ S используется для обозначения всех возможных строк языка.

Рассмотрим в качестве примера формального языка множество строк, являющихся корректными идентификаторами. Корректным идентификатором будем называть последовательность символов, начинающуюся с латинской буквы, за которой следует произвольное количество латинских букв или арабских цифр. Очевидно, что множество терминалов Σ в этом случае будет

образовано латинскими буквами и арабскими цифрами. Тогда можно записать следующий набор правил:

Identifier → Letter (Letter | Digit)*
Letter → 'A' | 'B' | … | 'Z' | 'a' | 'b' | … | 'z'
Digit → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

В роли стартового символа S в данном случае будет выступать нетерминал *Identifier*.

Для записи грамматик принято использовать обозначения, приведенные в табл. 1.

Таблица 1

Обозначения, используемые при записи правил грамматик

Обозначение	**Смысл**
'c'	Символ 'c', записанный без апострофов
ε	Пустая строка
$A \mid B$	Альтернатива: либо A, либо B
AB	Последовательность: сначала A, затем B
$A*$	Итерация: A, встречающееся 0 или более раз
( )	Используются для группировки

Другим примером формального языка может быть множество строк, являющихся синтаксически корректными программами на языке Pascal. В этом случае правила грамматики могут иметь следующий вид:

PascalProgram → *ProgramHeader UsesClause ProgramBlock*
ProgramHeader → *ProgramKeyword Identifier* ';'
ProgramKeyword → ('P' | 'p') ('R' | 'r') ('O' | 'o') ('G' | 'g') ('R' | 'r') ('A' | 'a') ('M' | 'm')
…
Identifier → *Letter* (*Letter* | *Digit*)*
Letter → 'A' | 'B' | … | 'Z' | 'a' | 'b' | … | 'z'
Digit → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
…

Однако очевидно, что на практике такой способ оказывается громоздким. Во-первых, многие конструкции языков программирования нечувствительны к регистру, что будет приводить к записям наподобие правой части правила для *ProgramKeyword*. Во-вторых, в подавляющем большинстве языков программирования между любыми двумя лексемами допустимо включение

произвольного количества пробельных символов и комментариев, что потребовало бы существенного усложнения грамматики:

PascalProgram → SC ProgramHeader SC UsesClause SC ProgramBlock SC
ProgramHeader → SC ProgramKeyword SC Identifier SC ';' *SC*
SC → (*SpaceCharacter* | *Comment)**
SpaceCharacter → ('␣' | *TabCharacter* | *LineFeed* | *CarriageReturn*)
…

По этой причине, как правило, для грамматик сложных языков в качестве терминальных символов выбирают не отдельные символы, а целые лексемы.

В общем случае грамматика может описывать правила записи не только какой-либо текстовой информации, но и взаимное расположение графических элементов, соотношения между элементами звукозаписей и т. д. Тем не менее для дальнейшего рассмотрения наибольший интерес представляют именно грамматики, задающие правила записи текстов.

В соответствии с классификацией, предложенной Ноамом Хомским, все формальные языки могут быть отнесены к одному из четырех типов:

– тип 0 – неограниченные;

– тип 1 – контекстно-зависимые;

– тип 2 – контекстно-свободные;

– тип 3 – регулярные.

Данную классификацию также называют *иерархией Хомского*, поскольку взаимосвязи между типами языков имеют иерархический характер. Критерием для отнесения формального языка к тому или иному типу является возможность построения его грамматики из правил того или иного вида. Причем, поскольку один и тот же формальный язык может быть задан разными грамматиками, относящимися к разным типам в иерархии, считается, что формальный язык относится к наиболее простому возможному типу.

Другими словами, регулярные языки могут быть описаны грамматиками любого типа, контекстно-свободные языки – любыми грамматиками, кроме регулярных, и т. д. Таким образом, можно считать, что множество языков

каждого следующего типа в иерархии Хомского является подмножеством языков предыдущего типа.

Наиболее ограниченным типом формальных языков являются регулярные языки. Для того чтобы формальный язык мог быть регулярным, его грамматика должна записываться с использованием только *продукций* (правил) вида

$A \to a$
$A \to aB$ или $A \to Ba$ (только один из двух видов)
$A \to \varepsilon$,

где $A, B \in N$, $a \in \Sigma$.

Для практических целей достаточно понимать, что регулярные грамматики не способны задавать языки с рекурсивными конструкциями, например, вложенными круглыми скобками или операторными скобками.

Для описания синтаксиса многих языков программирования оказывается достаточно контекстно-свободных грамматик. Для того чтобы быть контекстно-свободной, грамматика должна состоять из правил вида

$A \to \beta$,

где $A \in N$, β – произвольная последовательность терминалов и нетерминалов. Такие грамматики задают правила подстановки, которые одинаковы для одного и того же нетерминала независимо от его местоположения в тексте, причем допускается замена нетерминала на последовательность символов грамматики, включающих этот же нетерминал.

Контекстно-зависимые языки задаются грамматиками, состоящими только из правил вида

$\alpha A \beta \to \alpha \gamma \beta$,

где $A \in N$, а α , β и γ – произвольные последовательности терминалов и нетерминалов, причем α и β еще называют *контекстом*. Фактически такие грамматики позволяют задать правила подстановки для нетерминалов, зависящие от того, какие символы грамматики окружают данный нетерминал в тексте.

К неограниченным языкам относят все остальные формальные языки.

Лабораторная работа №1
Распознавание и поиск лексем

Теоретические сведения

Общие сведения

Лексический анализатор (сканер) – это часть транслятора, которая считывает символы программы на исходном языке и строит из них слова (лексемы) этого языка. На вход лексического анализатора поступает текст исходной программы, а выходная информация представляет собой поток токенов, который передается для дальнейшей обработки на этап синтаксического анализа.

Состав лексем, выделяемых лексическим анализатором, а также множество решаемых им вспомогательных задач могут изменяться в зависимости от особенностей исходного языка, а также принятых разработчиком транслятора архитектурных и инженерных решений. Тем не менее, как правило, сканер решает три основные задачи:

– выделение лексем языка;

– классификация выделенных лексем;

– удаление лексем, не влияющих на смысл программы.

Лексема – группа символов, имеющих в исходном языке самостоятельное значение. В зависимости от того, какую роль та или иная лексема играет в исходном языке, ее относят к одному из классов лексем (токенов). В большинстве языков программирования выделяются следующие классы токенов:

– идентификаторы;

– литералы (строковые, числовые и другие константы);

– ключевые слова;

– комментарии;

– блоки пробельных символов;

– знаки операций;

– символы-разделители.

В случае если лексическому анализатору не удается выделить или классифицировать ту или иную лексему, говорят об ошибке в исходной программе. Как правило, такие ошибки связаны с использованием в тексте программы недопустимых символов, ошибками в записи числовых и строковых литералов и т. д.

К лексемам, не влияющим на смысл программы, в большинстве языков относятся комментарии и блоки пробельных символов. На обработку синтаксическому анализатору соответствующие этим лексемам токены, как правило, не передаются.

Регулярные языки и конечные автоматы

Для большинства компьютерных языков множества корректных лексем образуют регулярные языки. *Регулярными выражениями* называется способ задания регулярных языков (метаязык), который задается следующей грамматикой:

R → ε
  | 'c'
  | R + R
  | RR
  | R*

причем справедливы следующие равенства:

$$L(\varepsilon) = \{""\}$$
$$L('c') = \{"c"\}$$
$$L(AB) = \{ab \mid a \in L(A), b \in L(B)\}$$
$$L(A+B) = \{a \mid a \in L(A)\} \cup \{b \mid b \in L(B)\}$$
$$L(A*) = \bigcup_{i=0}^{\infty} L(A^i), A^i = \underbrace{AA\ldots A}_{i\ раз}$$

где $L(R)$ – функция значения, задающая соответствие между регулярным выражением R и множеством соответствующих ему строк.

Кроме того, для удобства обозначений часто используются расширенные конструкции, некоторые из них приведены в табл. 2.

Таблица 2

Расширенные конструкции регулярных выражений

Расширенная конструкция	**Эквивалентная базовая конструкция**
A | B	A + B
A+	AA*
[A–Za–z$%]	'A' | 'B' | … | 'Z' | 'a' | 'b' | … | 'z' | '$' | '%'
A?	A + ε

Для выделения лексем используются конечные автоматы. Математически доказано, что следующие множества языков эквивалентны:

– языки, которые могут быть обработаны конечными автоматами;

– языки, которые могут быть заданы регулярными выражениями;

– регулярные языки.

Конечным автоматом называют математическую абстракцию, которую можно рассматривать как некоторое устройство, в каждый момент времени находящееся в одном из конечного множества состояний, а также имеющего один вход (на которые подаются сигналы извне) и один выход (на который подаются выходные сигналы, сформированные автоматом). Как правило, в теории трансляторов выходные сигналы конечных автоматов не рассматриваются, внимание уделяется только состояниям и входным сигналам.

Таким образом, конечный автомат задается пятеркой $M = (Q, S, \delta, q_0, F)$, где Q – конечное множество состояний автомата; S – конечное множество допустимых входных сигналов; δ – множество переходов, являющееся отображением $\delta : Q \times S \to Q$; $q_0 \in Q$ – начальное состояние автомата; $F \subseteq Q$ – множество заключительных (допускающих) состояний автомата.

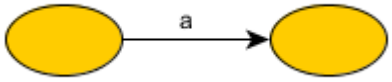
Работа автомата выполняется по тактам. На каждом такте i автомат, находясь в некотором состоянии q_i, считывает очередной сигнал (символ) $w \in S$ из входной цепочки символов и изменяет свое состояние на $q_{i+1} = \delta(q_i, w)$. При этом указатель в цепочке входных символов передвигается на следующий символ и начинается такт $i+1$. Так продолжается до тех пор, пока цепочка входных символов не закончится. Считается также, что перед тактом 1 автомат находится в начальном состоянии q_0.

Говорят, что автомат *допускает* («accept») цепочку символов, если в результате выполнения всех тактов обработки этой цепочки он оказывается в состоянии $q \in F$. Если же такое состояние не достигнуто (автомат находится не в заключительном состоянии или не смог выполнить переход при поступлении очередного сигнала), говорят, что автомат *не допускает* («reject») цепочку символов.

Конечный автомат часто изображается диаграммой переходов – взвешенным ориентированным графом, вершинами которого являются состояния, а дугами – переходы, причем дуги помечены сигналами, по которым такие переходы происходят. В дальнейшем для удобства будут применяться обозначения, приведенные в табл. 3.

Таблица 3

Обозначения, используемые в диаграммах переходов

Обозначение	**Смысл**
	Состояние автомата
	Начальное состояние автомата
	Заключительное состояние автомата
	Переход по сигналу *a*

Существует также особый вид переходов – *ε-переходы*. Такие переходы позволяют автомату изменять свое внутреннее состояние без поглощения символа входной цепочки.

Конечные автоматы принято разделять на детерминированные и недетерминированные. Для того чтобы конечный автомат был *детерминированным* (ДКА), должно выполняться два условия:

– из любого состояния $q \in Q$ по любому сигналу $w \in S$ должно существовать не более одного перехода $\delta(q, w)$;

– в автомате не должно быть ε-переходов.

Если хотя бы одно из этих условий не выполняется, автомат называют *недетерминированным* (НКА).

Разработка лексического анализатора

В общем случае построение лексического анализатора включает в себя следующие этапы:

– подготовка лексической спецификации;

– запись регулярных выражений по лексической спецификации;

– построение НКА по регулярным выражениям;

– преобразование НКА в ДКА;

– программная реализация полученного ДКА.

В роли лексической спецификации может выступать любое описание лексем, подлежащих выделению и распознаванию. Программная реализация может производиться не только для ДКА, но и для НКА: в этом случае становится возможной реализация некоторых расширенных конструкций регулярных выражений за счет снижения производительности.

Рассмотрим пример программной реализации конечного автомата, допускающего строки, состоящие из целых чисел, разделенных знаками операций +, –, * и /.

Составим регулярное выражение, задающее такие строки.

```
[0–9]+ (('+' | '–' | '*' | '/') [0–9]+)*
```

Очевидно, что включение к конечный автомат отдельных дуг для каждого возможного сигнала в данном случае нецелесообразно, поэтому разделим все возможные входные сигналы на классы и перепишем регулярное выражение следующим образом:

// Классы символов:
Digit ::= [0–9]
Operator ::= '+' | '–' | '*' | '/'

// Регулярное выражение для допустимых строк:
ValidString ::= *Digit*+ (*Operator Digit*+)*

Построим НКА для данного регулярного выражения, используя для пометки дуг классы символов (рис. 2).

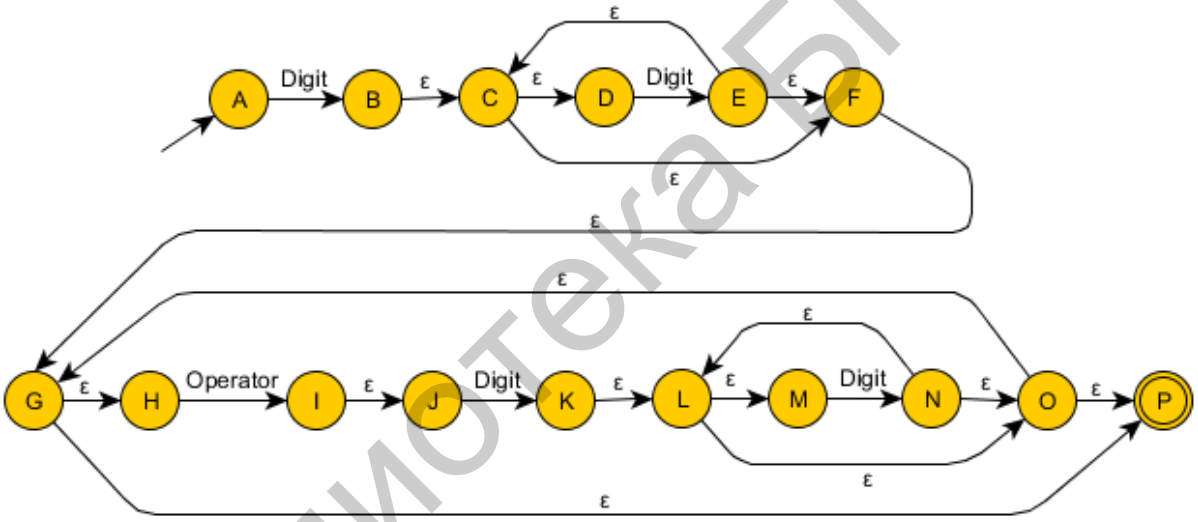


Рис. 2. Пример НКА

Выполним преобразование полученного НКА в ДКА, заменяя множества состояний НКА на одиночные состояния ДКА путем вычисления ε-замыканий соответствующих множеств состояний. Диаграмма переходов соответствующего автомата приведена на рис. 3.

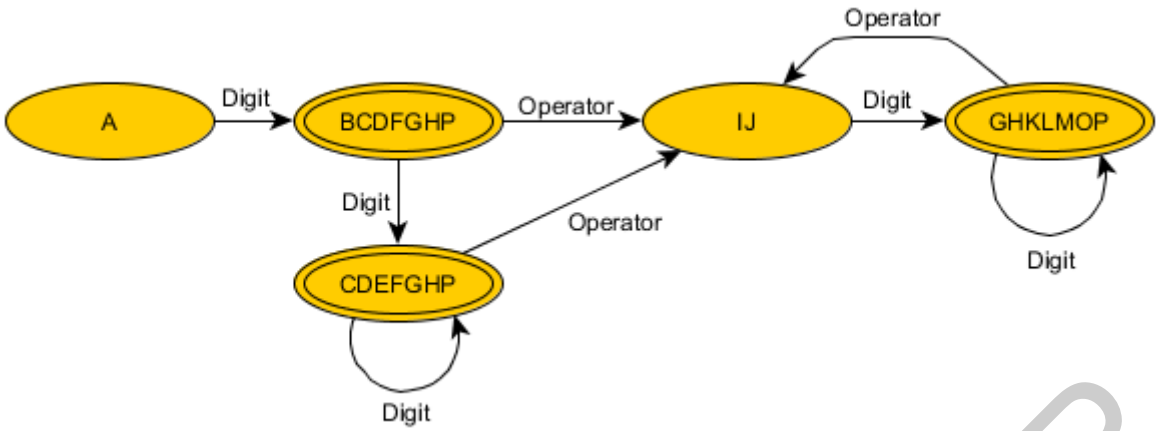


Рис. 3. Пример эквивалентного ДКА

Обратим внимание на то, что полученный ДКА не является оптимальным, так как состояния BCDFGHP и CDEFGHP имеют абсолютно идентичные правила переходов и оба являются заключительными. Это означает, что можно заменить их одним состоянием, как показано на рис. 4.

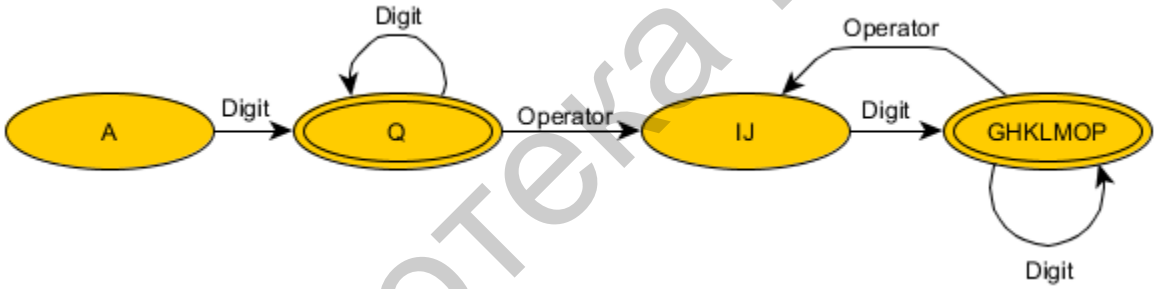


Рис. 4. Пример оптимизированного эквивалентного ДКА

Заменим имена состояний на номера начиная с 1, как показано на рис. 5.

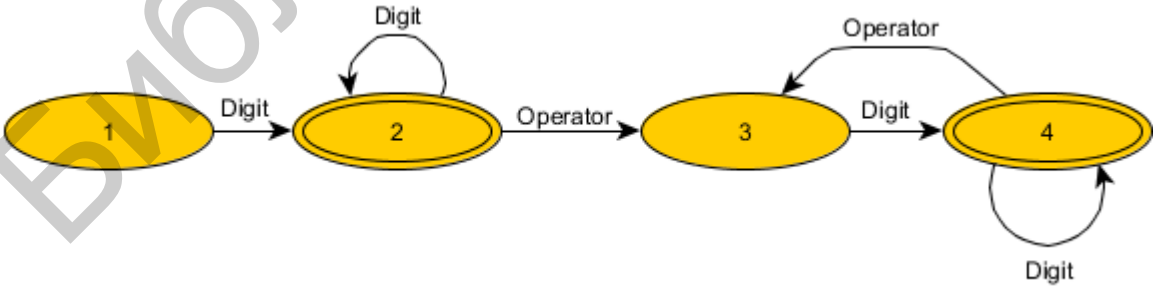


Рис. 5. Пример ДКА, готового к программной реализации

Следует отметить, что рассмотренный метод преобразования не гарантирует получения минимального ДКА: полученный в примере ДКА может очевидным образом быть сокращен до двух состояний. Тем не менее даже без

минимизаци он может быть использован для программной реализации. Следует обратить внимание на то, что нахождение полученного ДКА в каждом из состояний может быть охарактеризовано терминами решаемой задачи, как показано в табл. 4.

Таблица 4

Значение состояний полученного ДКА

Номер состояния	**Значение**
1	Прочитано: ничего. Ожидается: цифры первого числа
2	Прочитано: одна или несколько цифр. Ожидается: новые цифры первого числа, знак операции или конец строки
3	Прочитано: знак операции. Ожидается: цифры следующего числа
4	Прочитано: одна или несколько цифр. Ожидается: новые цифры числа, знак операции или конец строки

Для программной реализации ДКА потребуются:

– переменная State;

– двумерный массив-константа Transitions;

– одномерный массив-константа IsFinalState.

Реализуем полученный ДКА на языке Pascal. Для удобства реализации вводится дополнительное состояние 0, в которое автомат переходит в случае отсутствия соответствующего перехода на диаграмме переходов и в котором остается до перезапуска. Кроме того, для повышения сопровождаемости кода используются объявления вспомогательных типов.

```
type
  TCharType = (ctUnknown, ctDigit, ctOperator);
  TState = 0..4;

const
  Transitions: array [TState, TCharType] of TState =
    (
      (0, 0, 0),  // Состояние «Ошибка»
      (0, 2, 0),  // Состояние «Ждем первое число»
      (0, 2, 3),  // Состояние «Читаем первое число, ждем знака»
      (0, 4, 0),  // Состояние «Ждем следующее число»
      (0, 4, 3)   // Состояние «Читаем число, ждем знака»
    );

  IsFinalState: array [TState] of Boolean =
    (False, False, True, False, True);
```

Реализуем также вспомогательную функцию, выполняющую классификацию символа.

```
function GetCharType(C: Char): TCharType;
begin
  case C of
    '0'..'9':
      Result := ctDigit;
    '+', '-', '*', '/':
      Result := ctOperator;
    else
      Result := ctUnknown;
  end;
end;
```

Программная реализация непосредственно конечного автомата сводится к простому циклу по элементам строки.

```
function CheckString(const S: String): Boolean;
var
  i: Integer;
  State: TState;
begin
  State := 1;
  for i := 1 to Length(S) do
    State := Transitions[State, GetCharType(S[i])];
  Result := IsFinalState[State];
end;
```

Полученная функция может использоваться для проверки строки на соответствие исходной лексической спецификации. Аналогичным образом реализуется и функция выделения лексемы из начала строки.

Задание

Для заданного по варианту вида текстовых данных (табл. 5) проанализировать допустимые значения и разработать:

– **регулярное выражение**;

– **НКА** для этого регулярного выражения;

– **эквивалентный ДКА** и его **таблицу переходов**;

– **программное средство**, реализующее работу этого ДКА.

Программное средство должно поддерживать следующие возможности:

– проверка корректности произвольной строки;

– поиск всех подстрок, соответствующих требованиям, в произвольной строке.

Таблица 5

Варианты заданий к лабораторной работе №1

Номер варианта	Требования	Примеры корректных строк
1	Строковый литерал языка C с поддержкой следующих экранирующих последовательностей: • \n • \t • \\ • \"	"Hello, world!" "This \"string\" is valid." "Hello\nworld\t!" "Quotes are written this way: \\\"."
2	Строковый литерал языка Pascal (апостроф записывается как два апострофа подряд)	'Hello, world!' 'I don''t know.'
3	16-ричное число в синтаксисе C или Pascal	$c001c0De 0xCa5EFeED
4	Классический абсолютный путь к файлу в Windows: • латинская буква от A до Z; • двоеточие; • обратный слеш; • 0 или более блоков из допустимых символов, разделенных обратными слешами. К допустимым относятся все символы, кроме «*», «|», «\», «:», «"», «<», «>», «?» и «/». (Для поиска подстрок недопустимым символом пути следует также считать пробел « »)	C:\Windows\winmine.exe D:\WebServer\home\site.by\www\.htaccess Z:\autoexec.bat N:\testfile. X:\testfile2
5	Двоичное, восьмеричное или 16-ричное число в синтаксисе FASM (постфиксная форма записи). 16-ричное число должно начинаться с цифры от 0 до 9	01101011B 54212110o 51245H 0B8Ch
6	Корректный идентификатор: начинается с буквы или нижнего подчеркивания, затем произвольное количество букв, цифр и/или нижних подчеркиваний	abCde x0 Y z_1 test_value_is_here _ __custom
7	Адрес электронной почты (упрощенно). Имя ящика – произвольный набор букв, цифр и символов «.» (точка). Доменное имя – одна или более групп, состоящих из букв, цифр и дефисов, разделенных точками	test@localhost vasya@pupkin.ru ivan.ivanov@mail.bsuir.by a.b.c@d-e-f.com

Лабораторная работа №2
Преобразование текстов с использованием генератора лексических анализаторов flex

Теоретические сведения

Общие сведения

Построение НКА и ДКА для лексем большинства компьютерных языков вручную является трудоемкой задачей, требующей повышенного внимания и тщательной проверки полученных результатов, причем даже незначительные изменения в лексической спецификации языка могут потребовать повторного выполнения всех преобразований.

Как правило, при разработке трансляторов для сложных компьютерных языков применяют инструменты, автоматизирующие наиболее трудоемкие преобразования. Одним из таких инструментов стала утилита lex, первоначально разработанная Майком Леском и Эриком Шмидтом и ставшая стандартным генератором лексических анализаторов в операционных системах UNIX, а также включенная в стандарт POSIX.

В настоящее время в ходе развития инструментария на смену классической утилите lex пришла flex, которая, однако, в значительной степени сохранила обратную совместимость с lex. Существует несколько реализаций flex для ОС Windows, имеющих ряд несущественных отличий. В данном учебно-методическом пособии будет рассмотрена реализация flex в рамках проекта GNU Win32.

Flex – это генератор лексических анализаторов, исходными данными для которого является лексическая спецификация анализируемого языка, а результатом – исходный код лексического анализатора, способного выполнять разбор этого языка.

Описание для flex представляет собой текстовый файл, который в общем случае состоит из трех секций:

```
... Определения ...
%%
... Правила ...
%%
... Подпрограммы ...
```

Секция определения содержит объявления вспомогательных символов и начальных условий. Секция правил задает поведение лексического анализатора при обнаружении в анализируемом тексте тех или иных последовательностей символов. Содержимое секции подпрограмм просто копируется в генерируемый flex модуль лексического анализатора.

Секция правил является обязательной, поэтому простейшее описание для flex выглядит следующим образом:

%%

Полученный в результате обработки такого описания лексический анализатор будет выполнять посимвольное копирование анализируемых текстов из потока ввода в поток вывода, так как действием по умолчанию является именно копирование.

Между тем кода, сгенерированного flex, в этом случае будет недостаточно для сборки проекта. В частности, для такого описания flex не сгенерирует код для двух функций: `main()`, являющейся точкой входа программы, и `yywrap()` – вспомогательной функции, определяющей поведение лексического анализатора по окончании анализируемого текста.

Для более гибкой настройки flex может использоваться специальная директива `%option`, указываемая в секции определений (первой секции файла-описания). Допускается указание нескольких таких директив, а также указание нескольких опций (настроек) в одной директиве. Большинство опций задается именем, которое может дополняться словом `no`. Кроме того, ряд опций может быть задан не только с помощью данной директивы, но и при запуске flex из командной строки.

К числу опций, доступных только из файла-описания, относятся `yywrap` и `main`. Первая в случае отрицания указывает flex, что по окончании

анализируемого текста сгенерированный лексический анализатор должен считать, что больше нет файлов, требующих обработки, и возвращать управление вызывающей программе. Вторая опция заставляет flex помимо кода лексического анализатора сгенерировать и функцию `main()`. При этом опция `main` автоматически устанавливает и опцию `noyywrap`.

Таким образом, минимальное описание, позволяющее получить готовую к компиляции программу, выглядит так:

```
%option main
%%
```

Сохраним описание в файл с именем `Test1.l`. Теперь для генерации лексического анализатора необходимо запустить flex из командной строки, передав файл-описание в качестве параметра. В простейшем случае команда будет выглядеть так:

```
flex Test1.l
```

В случае отсутствия ошибок в файле-описании flex создаст файл `lex.yy.c`, который и будет содержать сгенерированный лексический анализатор. Соответствующий исполняемый файл будет представлять собой консольное приложение, которое выводит на экран вводимые пользователем строковые данные. Чтобы завершить работу программы, можно ввести специальный символ с кодом 26, который интерпретируется стандартной библиотекой C/C++ как признак конца текстового файла. Для этого необходимо нажать сочетание клавиш Ctrl + Z и завершить ввод (рис. 6).

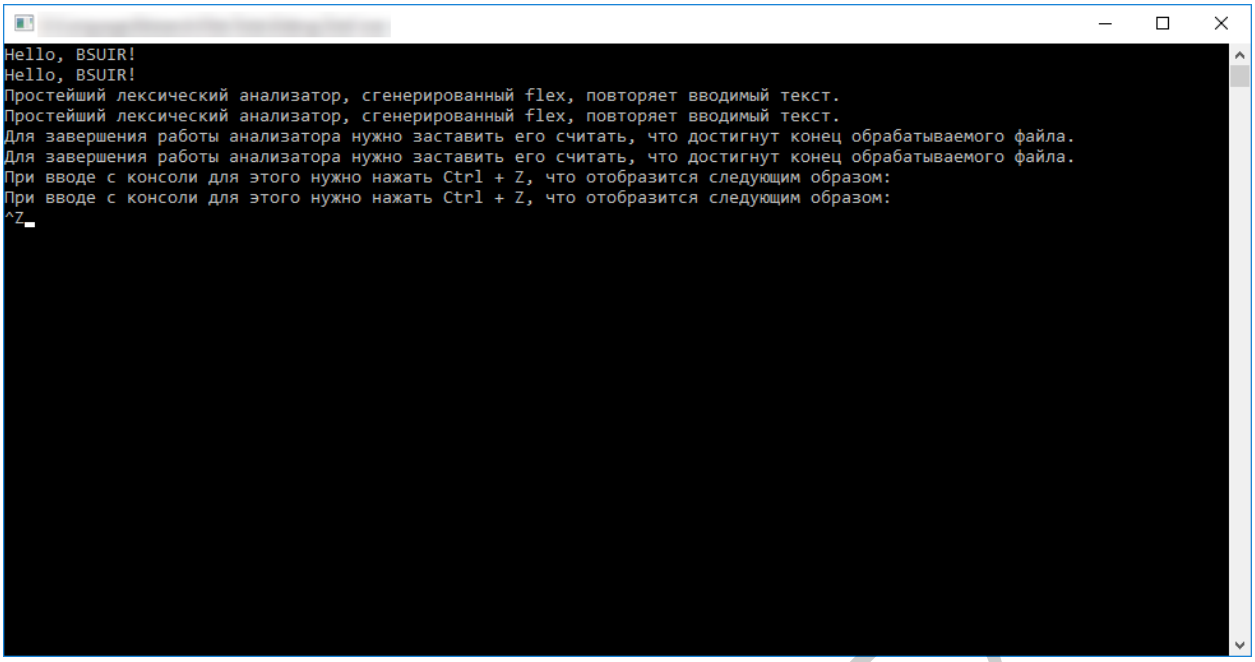


Рис. 6. Простейший лексический анализатор, сгенерированный flex

Полученная программа считывает исходные данные из стандартного потока ввода (`stdin`) и записывает результаты обработки в стандартный поток вывода (`stdout`). Благодаря этому возможно использовать перенаправление для обработки с ее помощью произвольных файлов, например, так:

```
test1 < input.txt > output.txt
```

где `test1` – имя исполняемого файла, `input.txt` и `output.txt` – соответственно имена исходного файла и файла с результатами обработки. При таком способе запуска полученная программа будет осуществлять посимвольное копирование файлов.

Сгенерированная flex функция `main()` имеет следующий вид:

```
int main()
  {
  yylex();
  return 0;
    }
```

Таким образом, эта функция является всего лишь «оберткой», запускающей функцию `yylex()`, которая и реализует лексический анализатор.

Как правило, лексический анализатор является лишь частью большой программы. В этом случае автоматически сгенерированная функция `main()` оказывается ненужной. Чтобы заставить flex сгенерировать лексический анализатор без этой функции, но при этом не требовать реализации функции `yywrap()`, можно использовать другой параметр директивы `%option`:

```
%option noyywrap
%%
```

В этом случае вызов функции `yylex()` должен будет выполнить программист.

Для того чтобы лексический анализатор делал что-нибудь более полезное, необходимо заполнить его секцию правил (вторую). Например, такое описание:

```
%option main
%%
username        printf("User12345");
```

приведет к генерации кода лексического анализатора, который будет копировать исходный текст в выходной поток, заменяя подстроки `username` на `User12345` (рис. 7).

Рис. 7. Лексический анализатор, заменяющий подстроку `username`

Левая часть правила называется *шаблоном*, правая – *действием*. Как было показано ранее, действие по умолчанию заключается в выводе каждого прочитанного символа. Однако, если какая-либо подстрока во входном тексте соответствует шаблону, лексический анализатор будет выполнять соответствующее действие. Действия представляют собой фрагменты кода на языке C, которые встраиваются в сгенерированный лексический анализатор.

Рассмотрим более сложный пример:

```
%option noyywrap
%{
int nLines = 0, nChars = 0;
%}

%%

\n      {
      ++nLines; ++nChars;
        }

.       ++nChars;

%%

int main()
{
    yylex();
    printf("Lines: %d. Characters: %d", nLines, nChars);
}
```

Работа функции `main()` будет отличаться от стандартной реализации, поэтому вместо `%option main` используется директива `%option noyywrap`, смысл которой уже был описан ранее: она позволяет избавиться от необходимости самостоятельно реализовывать функцию `yywrap()`, которую сгенерированный лексический анализатор вызывает, когда доходит до конца обрабатываемого им файла.

В секциях определений и правил текст, записанный с отступами или заключенный между `%{` и `%}`, копируется в модуль лексического анализатора в неизменном виде. При этом `%{` и `%}` должны быть записаны в отдельных строках без отступов. Следовательно, следующие три строки заставляют flex

вставить в модуль лексического анализатора объявление с инициализацией двух целочисленных переменных – `nLines` и `nChars`.

Далее следует раздел правил. Шаблон, записываемый в левой части правила, является регулярным выражением. Шаблон `\n` соответствует символу с кодом 10 (Line Feed), шаблон «`.`» означает «любой символ, кроме перевода строки (Line Feed)». Соответствующие этим шаблонам действия приводят к тому, что для любого символа входной последовательности будет наращиваться значение переменной `nChars`, а для символа перевода строки, кроме того, будет наращиваться и значение переменной `nLines`.

Содержимое секции подпрограмм включается в код генерируемого flex модуля без изменений.

Таким образом, программа, генерируемая для приведенного описания, осуществляет подсчет символов и строк, подаваемых на вход, и выводит результаты подсчетов. Этот и последующие примеры наиболее удобно рассматривать с использованием перенаправления в файлы, как было показано ранее.

Синтаксис шаблонов flex

Шаблоны в файле-описании представляют собой расширенные регулярные выражения. Основные конструкции, которые могут применяться в шаблонах, приведены в табл. 6.

В символьных классах кроме перечисления символов и диапазонов могут применяться специальные обозначения, задающие наиболее часто используемые подмножества символов. Список таких обозначений приведен в табл. 7. В частности, следующие шаблоны эквивалентны:

```
[[:alnum:]]
[[:alnum:][:digit:]]
[[:alpha:]0-9]
[a-zA-z0-9]
```

Также следует помнить, что внутри конструкции, задающей символьный класс, все спецсимволы регулярных выражений, такие как, например, `*` или `|`, теряют свое особое значение и становятся обычными символами. Исключение составляют символ экранирования `\`, спецсимволы, используемые при задании символьного класса (`-` и `]`) и расположенный в начале описания символьного класса символ `^`.

Важно принимать во внимание приоритет операций. Приведенные в табл. 6 операции (конкатенация, альтернатива и др.) перечислены в порядке убывания приоритета. Так, например, шаблон

```
foo|bar*
```

эквивалентен шаблону

```
(foo)|(ba(r*))
```

так как операция «*» имеет более высокий приоритет, чем конкатенация. Шаблон, задающий альтернативу между одиночным `foo` и повторением 0 или более раз `bar`, можно записать следующим образом:

```
foo|(bar)*
```

Повторение же `foo` или `bar` запишется так:

```
(foo|bar)*
```

Таблица 6

Основные конструкции, используемые в шаблонах правил flex

Шаблон	**Значение**
`x`	Соответствует символу `x`
`.`	Любой символ (байт), кроме перевода строки
`[xyz]`	Символьный класс. В этом примере соответствует одному из символов `x`, `y` или `z`
`[abj-oZ]`	Символьный класс. В этом примере соответствует символу `a`, символу `b`, одному из символов от `j` до `o` или символу `Z`
`[^A-Z]`	Символьный класс с отрицанием. Соответствует любому символу, кроме входящих в символьный класс. В этом примере – любой символ, кроме символов от `A` до `Z`
`[^A-Z\n]`	Символьный класс с отрицанием. В этом примере – любой символ, кроме символов от `A` до `Z` и символа перевода строки
`r*`	0 или более раз `r`, где `r` – произвольный шаблон
`r+`	1 или более раз `r`, где `r` – произвольный шаблон
`r?`	0 или 1 раз `r`, где `r` – произвольный шаблон
`r{2,5}`	От 2 до 5 раз `r`
`r{2,}`	2 и более раза `r`
`r{4}`	4 раза `r`
`{name}`	Подстановка определения `name`. Используется для сокращения описания. Определение должно быть задано в секции определений
`"[xyz]\"foo"`	Строковый литерал: [xyz]"foo
`\X`	Если `X` – один из символов `a`, `b`, `f`, `n`, `r`, `t` или `v`, означает запись символа по правилам ANSI C (аналогично строковым константам в языке C). В остальных случаях означает символ `X`. Используется для экранирования спецсимволов наподобие `*`
`\0`	Символ с кодом 0
`\123`	Символ с кодом $123_{(8)}$
`\x2a`	Символ с кодом $2A_{(16)}$
`(r)`	Круглые скобки используются для изменения приоритетов
`rs`	Конкатенация строки, заданной шаблоном `r`, со строкой, заданной шаблоном `s`
`r|s`	Либо `r`, либо `s`

Окончание табл. 6

Шаблон	Значение
r/s	r при условии, что далее следует s. После того как получено совпадение с шаблоном r, лексический анализатор будет просматривать последующий текст на предмет совпадения с шаблоном s (предпросмотр). Следует использовать с осторожностью, так как не все варианты s корректно обрабатываются flex
^r	r, но только в начале строки
r$	r, но только в конце строки (эквивалентно r/\n)

Таблица 7

Подмножества символов для символьных классов

Обозначение	Значение
[:alnum:]	Символы алфавита (буквы) и арабские цифры
[:alpha:]	Символы алфавита (буквы)
[:blank:]	Пробел и символ табуляции
[:cntrl:]	Управляющие символы
[:digit:]	Арабские цифры
[:graph:]	Символы, имеющие графическое представление (все печатные символы, кроме пробела)
[:lower:]	Строчные буквы
[:print:]	Печатные символы (все, кроме управляющих)
[:punct:]	Знаки пунктуации
[:space:]	Пробельные символы (пробел, символ табуляции, возврат каретки, перевод строки и т. п.)
[:upper:]	Прописные буквы
[:xdigit:]	16-ричные цифры

За информацией о поддержке национальных алфавитов (букв, отличных от латиницы) следует обращаться к документации конкретной версии flex.

Действия в правилах flex-описаний

Правая часть правила во flex-описаниях задает действия, которые необходимо выполнить при выявлении в анализируемом тексте подстроки, соответствующей шаблону из левой части правила.

Действие может быть произвольным фрагментом кода на языке C. При этом flex учитывает лексическую структуру языка C и, если действие начинается с открывающей фигурной скобки `{`, продолжит копировать такой фрагмент кода до момента, пока не встретит соответствующую ей закрывающую фигурную скобку `}`. Тем не менее допускается также обрамление действия скобками в стиле flex: `%{ ... }%`.

Если действие состоит из единственной вертикальной черты `|`, это означает «такое же действие, как и для следующего правила».

Код действия может быть совершенно произвольным, в том числе содержать оператор `return`. Поскольку код действия подставляется в генерируемую flex функцию лексического анализа `yylex()`, такой оператор заставит эту функцию вернуть произвольное значение из лексического анализатора. При каждом вызове функция `yylex()` продолжает анализ исходного текста с того места, где произошла остановка при предыдущем вызове. В частности, это полезно при разработке трансляторов, в которых вместо преобразования текстов требуется разбиение исходного текста на смысловые единицы – лексемы.

Действия могут получить доступ к фрагменту анализируемого текста, который совпал с шаблоном, обращаясь к переменной `yytext`. Эта переменная может объявляться как указатель `char *` или как массив `char`. Контролировать эту особенность можно явным указанием одной из директив: `%pointer` или `%array`. Использование `%pointer` повышает быстродействие генерируемого анализатора, однако ограничивает разработчика в допустимых действиях над переменной `yytext`.

При генерации кода flex также определяет несколько макросов для наиболее часто используемых действий. Одно из таких действий – `ECHO` – выполняет копирование `yytext` в выходной поток.

Еще одно предопределенное действие – `REJECT`. Оно заставляет сгенерированный анализатор перейти к обработке следующего правила по степени соответствия шаблона строке. Суть такого действия можно показать на примере следующего описания:

```
%option noyywrap

int nWords = 0;

%%

frob            special(); REJECT;
[^ \t\n]+       ++nWords;
```

Данное описание предназначено для подсчета количества слов в анализируемом тексте с дополнительной обработкой вхождений слова `frob`. Сгенерированный flex анализатор всегда выбирает одно наиболее подходящее подстроке правило. Слово `frob` соответствует шаблонам обоих правил, однако шаблон первого правила при этом более конкретный, поэтому для обработки будет выбрано именно оно. Однако, если убрать обращение к `REJECT` из соответствующего действия, слово `frob` обрабатывалось бы исключительно вызовом функции `special()`, а значит, наращивание счетчика слов либо не производилось бы вовсе, либо должно было бы быть включено в саму функцию `special()`.

Код, записанный после `REJECT`, выполнен не будет. Кроме того, следует помнить, что это действие снижает производительность анализатора, а также не может использоваться вместе с некоторыми настройками flex.

Задание

Для заданного по варианту способа преобразования текстовых данных (табл. 8) разработать:

– **flex-описание** анализатора;

– **программное средство**, сгенерированное на основе описания и выполняющее заданное преобразование.

Программное средство должно преобразовывать данные, поступающие из стандартного потока ввода `stdin`, и выводить результат своей работы в стандартный поток вывода `stdout`.

Таблица 8

Варианты заданий к лабораторной работе №2

Номер варианта	**Исходный текст**	**Преобразование**
1	Программа на языке Pascal	Удалить комментарии: (* … *) и { … }. Директивы компилятора не удалять
2	Программа на языке C	Удалить комментарии: /* … */ и // …
3	HTML-документ	Удалить комментарии: <!-- … -->
4	Текстовый файл	Добавить отступ в четыре пробела в начало каждого абзаца. Абзацами считать фрагменты текста, разделенные пустой строкой
5	Текстовый файл	Привести расстановку пробелов вокруг знаков препинания к следующему виду: * пробел не ставится перед и ставится после следующих знаков препинания: . , ! ? : ;) * пробел ставится перед и не ставится после открывающей круглой скобки
6	HTML-документ	Преобразовать документ так, чтобы все теги и содержимое парных тегов начинались с новой строки: <!DOCTYPE html> <html> <head> <title> Test file </title> </head> <body> Hello, world! </body> </html>

Продолжение табл. 8

Номер варианта	Исходный текст	Преобразование
7	Текстовый файл	Удалить пробельные символы, записанные в начале строки
8	Текстовый файл	Подсчитать, сколько раз в тексте использованы названия операционных систем Windows, Linux и Android. Получившиеся числа вывести в выходной файл по одному в строке
9	Текстовый документ с перепиской в формате: Имя1: Текст сообщения Имя2: Текст сообщения …	Определить и вывести долю сообщений пользователя с именем Admin, в процентах от общего количества сообщений
10	Текстовый документ	Зашифровать текст шифром Цезаря. Шифрованию подвергать только латинские буквы
11	Текстовый документ	Пронумеровать строки. Результатом работы должен быть текстовый документ, в котором перед началом каждой строки записан ее порядковый номер
12	Текстовый документ	Раскодировать содержащиеся в тексте urlencoded-последовательности. Например, текст Привет%20всем! превратить в Привет␣всем! После символа % указывается код символа в 16-ричной системе счисления

Окончание табл. 8

Номер варианта	Исходный текст	Преобразование
13	Программа на языке Pascal	Выделить все использованные в программе строковые литералы и записать их значения (без кавычек) по одному в строке
14	Программа на языке C	Выделить все использованные в программе строковые литералы и записать их значения (без кавычек) по одному в строке
15	Программа на языке ассемблера (диалект FASM)	Подсчитать и вывести количество глобальных, локальных и анонимных меток, использованных в программе

Лабораторная работа №3
Разработка синтаксического анализатора, основанного на методе рекурсивного спуска

Теоретические сведения

Общие сведения

Синтаксический анализ – это этап трансляции, задача которого заключается в построении древовидного представления анализируемого текста (например, исходного кода программы). Как правило, на вход синтаксического анализатора (который также называют *парсером*) подаются токены, получаемые в результате обработки анализируемого текста лексическим анализатором.

Теоретически возможно записать грамматику, терминалами в которой будут символы в той или иной кодировке, однако такая грамматика для большинства применяемых на практике компьютерных языков окажется излишне громоздкой, так как должна будет учитывать возможность появления пробельных символов или комментариев между любыми двумя лексемами.

Построение *дерева разбора* может выполняться в одном из двух направлений: от корня к листьям или от листьев к корню. По этому принципу методы синтаксического анализа принято разделять на две большие группы:

– нисходящие – дерево строится от корня к листьям;

– восходящие – дерево строится от листьев к корню.

Независимо от порядка построения листья в деревьях разбора содержат терминальные символы грамматики, промежуточные узлы (в том числе и корневой) – нетерминалы. При этом любой фрагмент дерева разбора, состоящий из промежуточного узла и его дочерних элементов, в точности соответствует одной из продукций грамматики – той, в левой части которой записан нетерминал из выбранного промежуточного узла, а в правой – последовательно заданы терминалы и нетерминалы дочерних узлов.

Очевидным недостатком деревьев разбора является зависимость их структуры от структуры грамматики. Фактически это означает, что любое изменение в грамматике может существенно изменить структуру дерева разбора, что потребует внесения изменений в код, отвечающий за последующие этапы трансляции. По этой причине более целесообразным, как правило, оказывается использование вместо деревьев разбора так называемых *абстрактных синтаксических деревьев* (abstract syntax trees – AST).

Конкретный способ представления тех или иных синтаксических конструкций языка в AST выбирается разработчиком транслятора исходя из специфики решаемой задачи. Тем не менее, как правило, AST представляет собой отражение логической структуры анализируемого текста, не зависящей от состава продукций грамматики.

Неоднозначные грамматики

Наиболее широко в синтаксическом анализе применяются контекстно-свободные грамматики. Это обусловлено, с одной стороны, их достаточно широкими возможностями для описания большинства компьютерных языков, а с другой – наличием детально проработанных методов разбора.

При разработке грамматики для синтаксического анализатора важно понимать, что не все контекстно-свободные грамматики подходят для практического применения.

Рассмотрим следующую грамматику:

Expr → *Expr* + *Expr*
| *Expr* * *Expr*
| int

Данная грамматика задает произвольные выражения, состоящие из целых чисел и операций сложения и умножения, при этом запись грамматики достаточно проста и наглядна. Тем не менее у нее есть один существенный недостаток, делающий невозможным ее практическое применение.

Рассмотрим последовательность токенов:

int + int * int

Вышеприведенная грамматика способна породить такую строку, однако может сделать это несколькими различными способами. При этом, как показано на рис. 8, эти способы будут отличаться приоритетами операций. В частности, дерево разбора, приведенное слева, задает выражение как произведение суммы на число, а дерево разбора, приведенное справа, – как сумму числа и произведения.

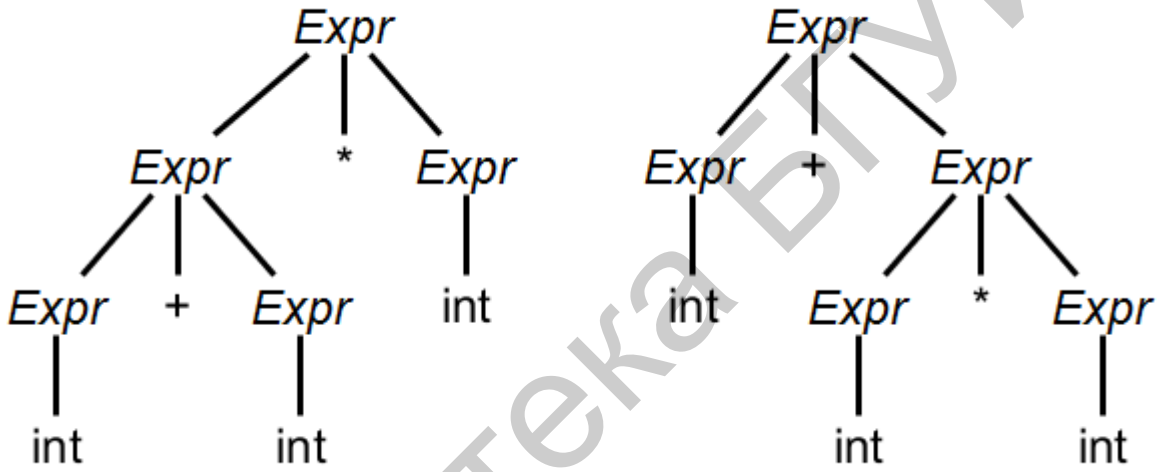


Рис. 8. Деревья разбора при неоднозначной грамматике

Очевидно, данная грамматика позволяет проверить синтаксическую корректность строки, но не задает приоритета операций. Решение этой проблемы заключается в том, чтобы переписать грамматику.

Expr → *ExprMul* + *Expr* | *ExprMul*
ExprMul → int * *ExprMul* | int

При этом в результате анализа той же самой строки будет возможно построение только дерева разбора, показанного на рис. 9.

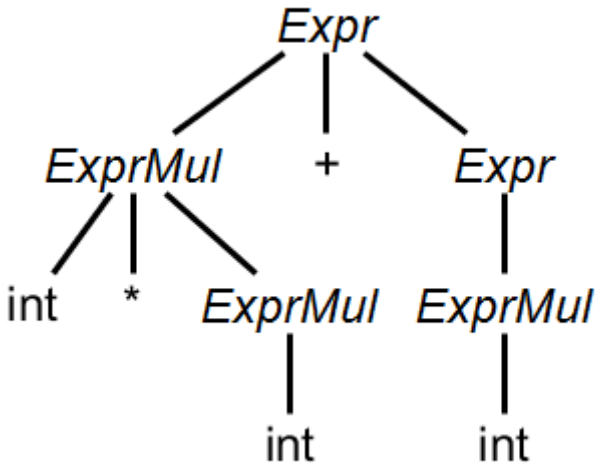


Рис. 9. Дерево разбора для однозначной грамматики

Другим примером неоднозначности, возникающей при построении грамматик языков программирования, является так называемая «проблема висячего else» (dangling else problem). Сущность этой проблемы можно продемонстрировать, записав грамматику в естественной форме:

IfStatement → if *Condition* then *Statement*
| if *Condition* then *Statement* else *Statement*

Проблема с использованием такой грамматики возникает в ситуации, когда требуется проанализировать программу, в которой присутствуют вложенные операторы `if`, причем некоторые из них имеют ветвь `else`, а некоторые – нет.

```
if a > b then
if c > d then
x := 1
else
x := 2;
```

Проблема в этом случае заключается в невозможности определения принадлежности необязательной ветви `else`. В большинстве языков программирования в подобной ситуации ветвь `else` относится к ближайшему оператору `if`, т. е. корректной интерпретацией является следующая:

```
if a > b then
  if c > d then
    x := 1
  else
    x := 2;
```

Тем не менее в соответствии с вышеприведенной грамматикой допустима и другая интерпретация.

```
if a > b then
  if c > d then
    x := 1
else
  x := 2;
```

Наиболее распространенным способом устранения данной неоднозначности является переписывание грамматики с выделением двух видов операторов `if`: с ветвью `else` и без нее.

IfStatement → *MatchedIf* | *UnmatchedIf*
MatchedIf → if *Condition* then *MatchedIf* else *MatchedIf*
| *Other*
UnmatchedIf → if *Condition* then *IfStatement*
| if *Condition* then *MatchedIf* else *UnmatchedIf*

Тем не менее использование такой грамматики все еще затруднительно с некоторыми методами синтаксического анализа.

Другим способом решения проблемы висячего else является использование «жадного» анализатора, т. е. подход, при котором анализатор ставит в соответствие нетерминалу как можно более длинную цепочку. Это позволяет устранить данную проблему, однако может привести к неправильному разбору других частей грамматики, для которых предпочтительнее «нежадный» алгоритм.

Наконец наиболее действенным способом (и в то же время наиболее трудоемким) является адаптация используемого метода синтаксического анализа, т. е. внесение в этот метод изменений, которые позволяют устранить неоднозначность. Так, например, это может быть ручное редактирование таблиц разбора или кода анализатора.

Определить, является ли некоторая грамматика неоднозначной, в общем случае алгоритмически невозможно. Тем не менее для некоторых методов синтаксического разбора необходимо построение вспомогательных структур данных и для неоднозначных грамматик их построение оказывается невозможным.

Метод рекурсивного спуска

Простым в реализации и вместе с тем эффективным методом синтаксического анализа является *метод рекурсивного спуска*. Суть данного метода заключается в том, что начиная со стартового символа грамматики поочередно перебираются все возможные комбинации продукций до тех пор, пока не будет найдено порождение, соответствующее анализируемому тексту, либо пока не окажется, что такого порождения не существует.

Рассмотрим метод рекурсивного спуска на примере следующей грамматики (курсивным начертанием обозначены нетерминалы, обычным – терминалы):

Operator → *Operator_If* | int
Operator_If → if *Operator* then *Operator* endif
| if *Operator* then *Operator* else *Operator* endif

Пусть получаемые из лексического анализатора токены формируют следующий поток:

if int_5 then if if int_8 then int_9 endif then int_{11} else if int_3 then int_4 else int_6 endif endif endif

Тогда ход работы синтаксического анализатора, основанного на методе рекурсивного спуска, будет следующим (рис. 10–24). На начальном этапе дерево разбора состоит из единственного узла, соответствующего стартовому символу грамматики, при этом ни один символ из последовательности токенов не прочитан.

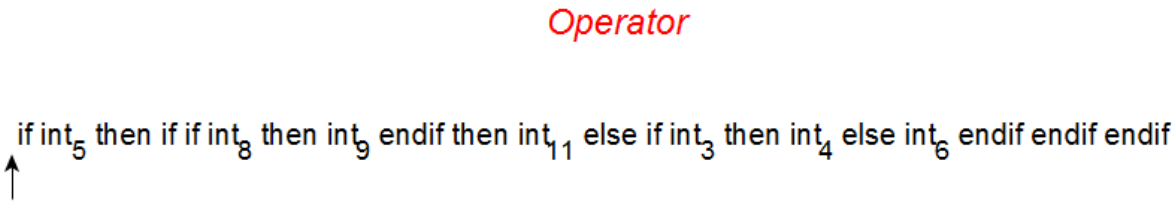


Рис. 10. Дерево разбора и поток токенов (шаг 1)

Узел в данный момент является текущим и соответствует нетерминалу. Для нетерминалов выполняется их раскрытие поочередно в соответствии с каждой из имеющихся продукций. Таким образом, на следующем шаге дерево разбора дополняется дочерними элементами по первой продукции символа *Operator* и первый из этих элементов становится текущим. В данном случае такой элемент является единственным.

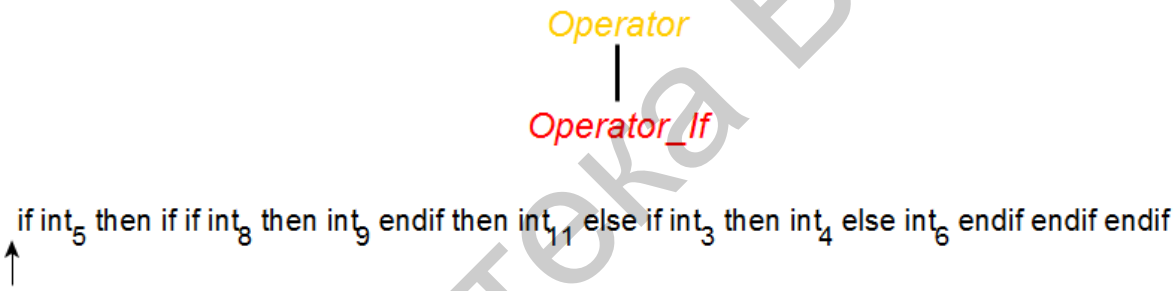


Рис. 11. Дерево разбора и поток токенов (шаг 2)

Текущим символом является нетерминал. Дерево разбора снова необходимо дополнить дочерними элементами в соответствии с продукцией этого нетерминала. Текущим при этом станет первый из этих элементов.

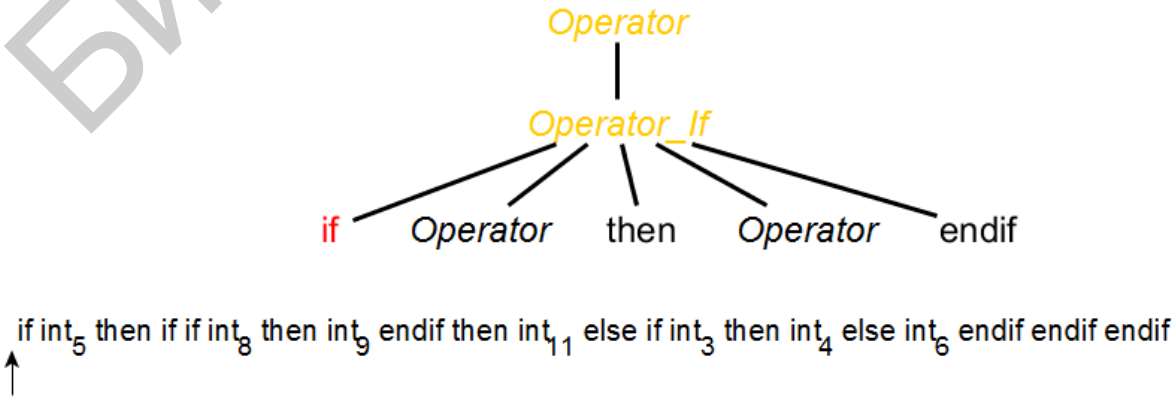


Рис. 12. Дерево разбора и поток токенов (шаг 3)

Текущим символом является терминал, а это означает, что необходимо сопоставить его с символом из входного потока токенов. В данном случае имеет место совпадение, поэтому указатель входного потока передвигается к следующему символу и следующий узел дерева становится текущим.

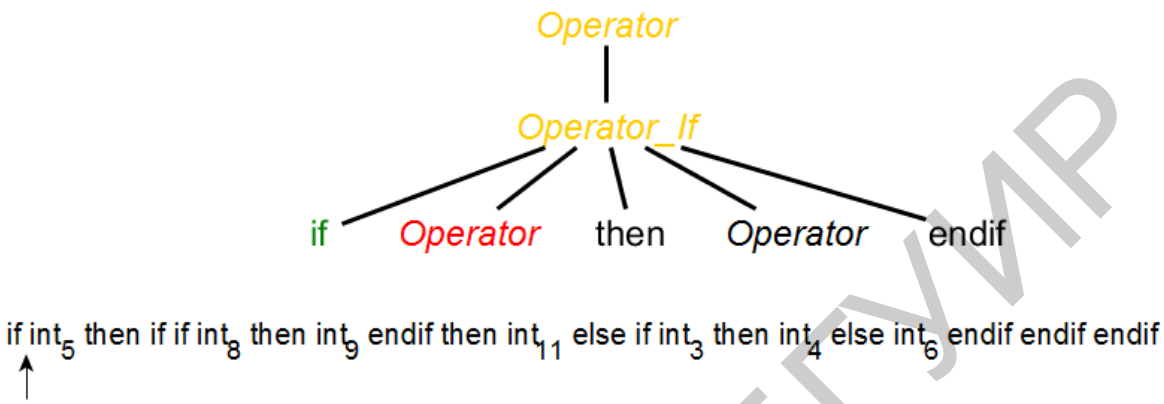


Рис. 13. Дерево разбора и поток токенов (шаг 4)

Текущим символом является нетерминал, поэтому производится дополнение дерева разбора дочерними элементами в соответствии с первой продукцией этого нетерминала.

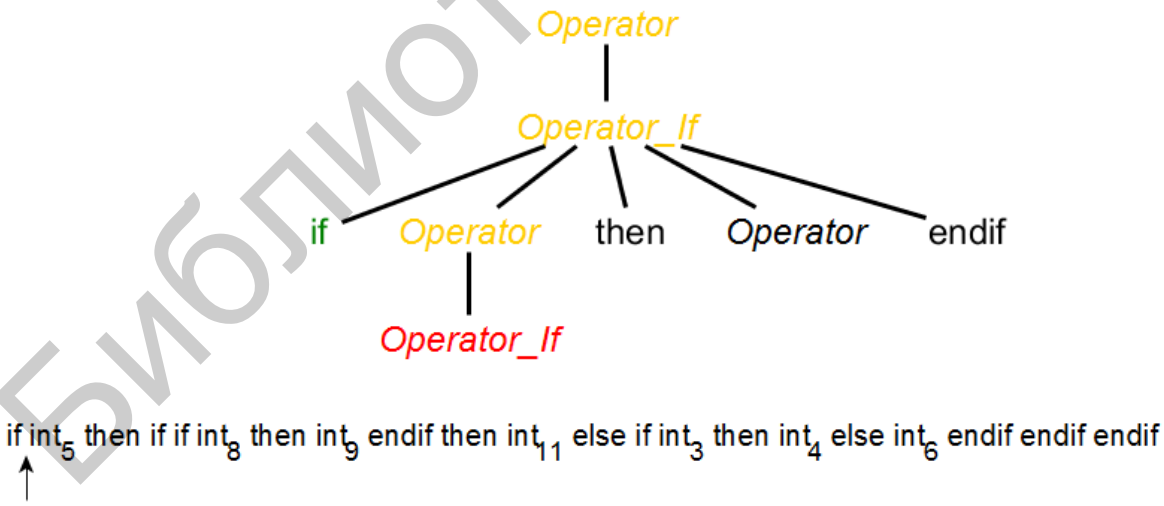


Рис. 14. Дерево разбора и поток токенов (шаг 5)

Текущий символ является нетерминалом, дерево разбора дополняется элементами в соответствии с первой продукцией этого нетерминала.

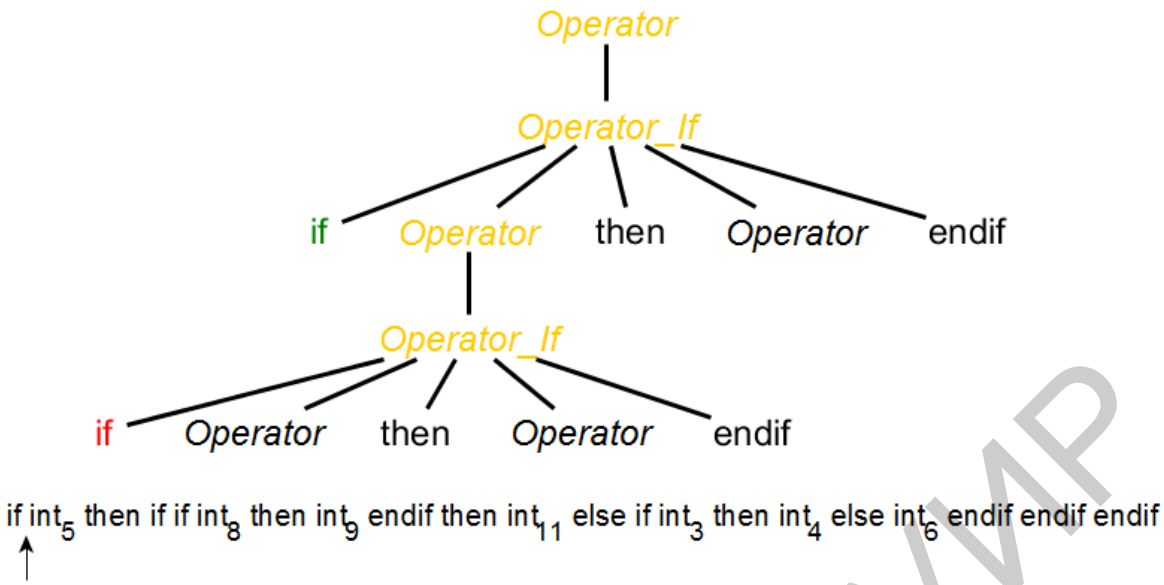


Рис. 15. Дерево разбора и поток токенов (шаг 6)

Текущий символ является терминалом, однако его тип не совпадает с классом токена из входного потока. Это означает, что выбранная продукция не подходит. Необходимо выполнить откат и попробовать применить другую продукцию.

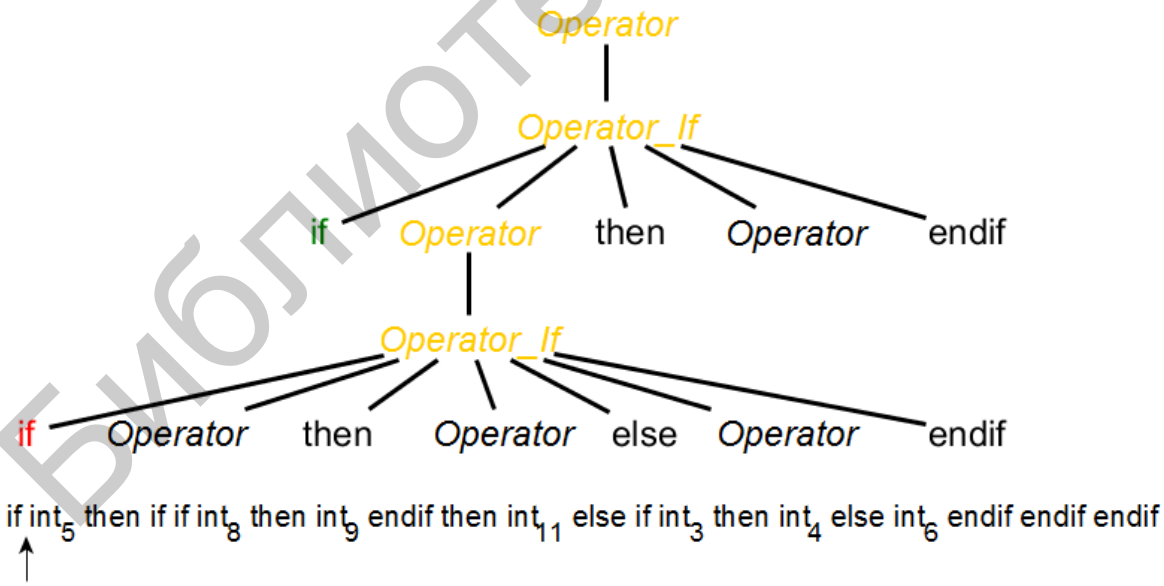


Рис. 16. Дерево разбора и поток токенов (шаг 7)

Текущий символ снова является терминалом и снова не соответствует входному потоку. Необходимо перейти к следующей продукции нетерминала *Operator_If*, однако эта продукция была последней. Значит, следует выполнить

откат сразу на два шага и применить следующую продукцию нетерминала *Operator*.

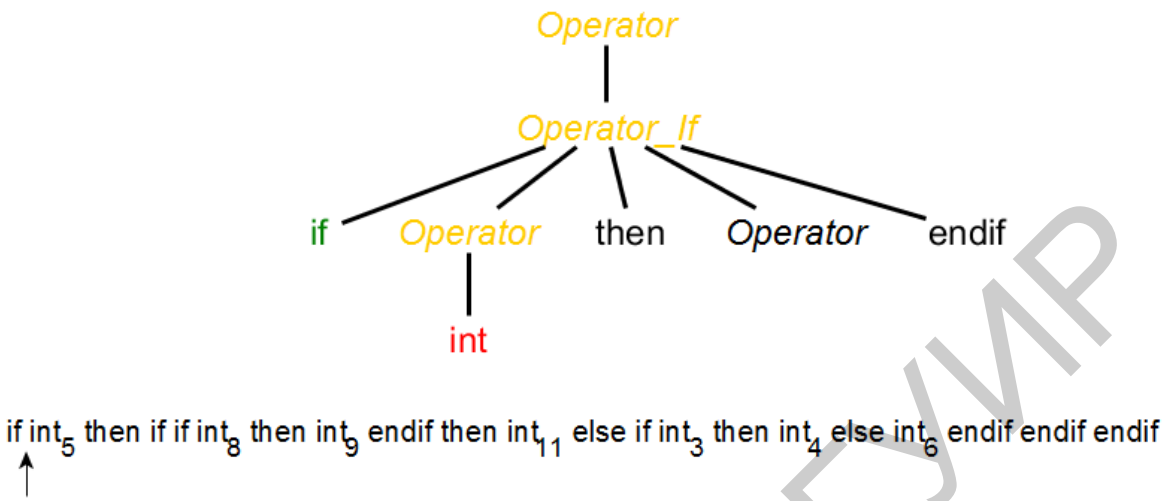


Рис. 17. Дерево разбора и поток токенов (шаг 8)

Текущий символ – терминал, совпадающий с терминалом из входного потока токенов. Следовательно, необходимо перейти к следующему необработанному узлу дерева разбора и следующему токену входного потока.

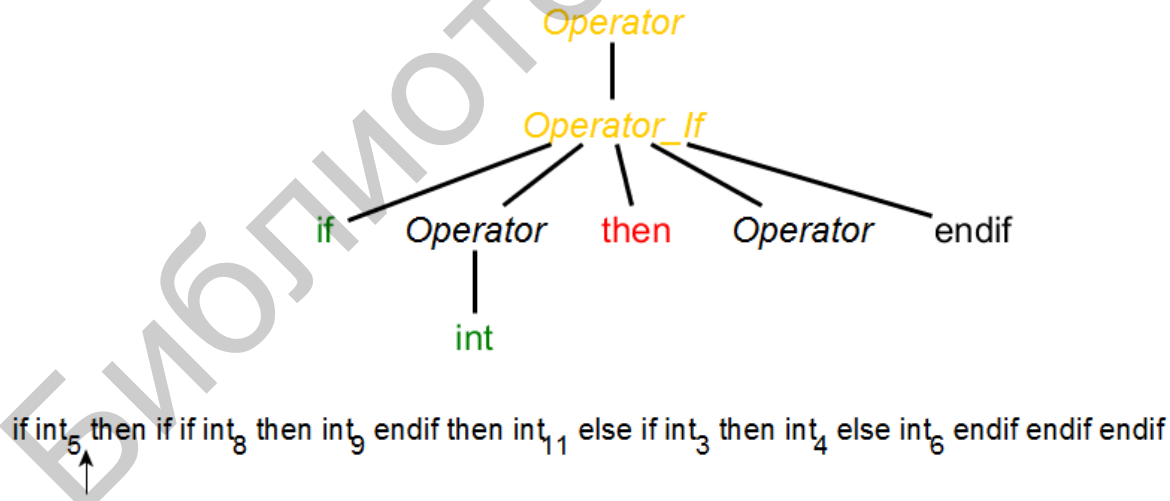


Рис. 18. Дерево разбора и поток токенов (шаг 9)

Текущий символ – терминал, совпадающий с терминалом из входного потока токенов. Необходимо перейти к следующему символу.

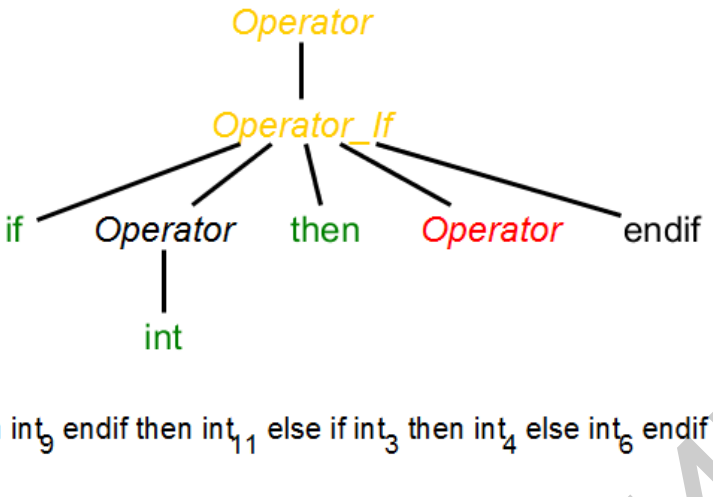


Рис. 19. Дерево разбора и поток токенов (шаг 10)

Текущий символ – нетерминал, необходимо раскрыть его в соответствии с первой продукцией.

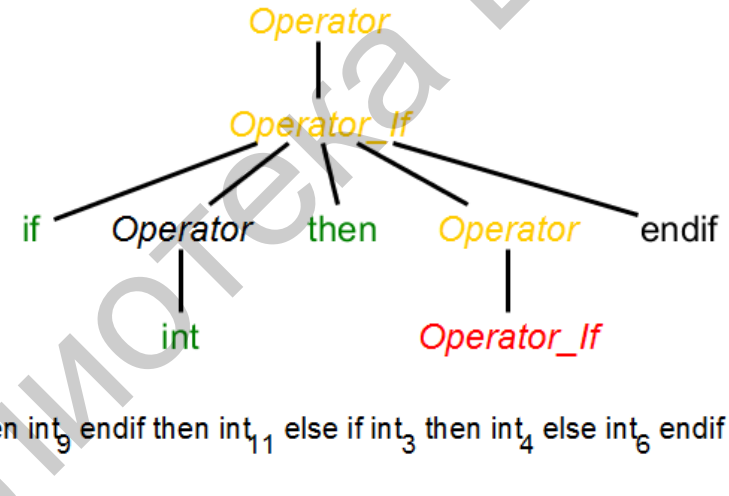


Рис. 20. Дерево разбора и поток токенов (шаг 11)

Нетерминал *Operator_If* раскрывается по первой продукции.

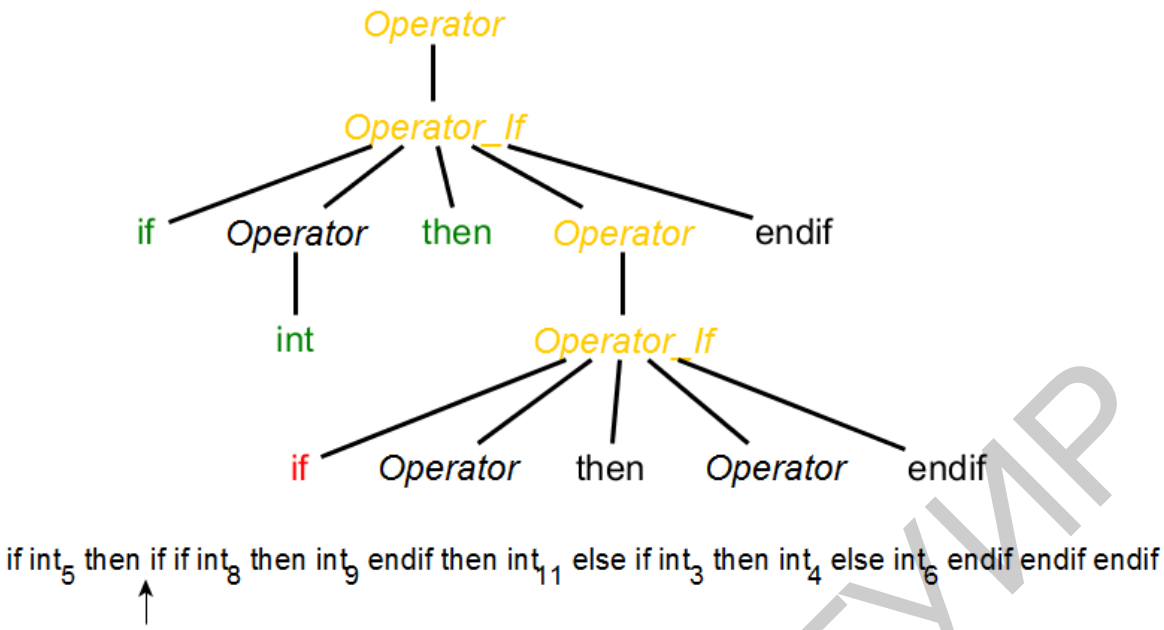


Рис. 21. Дерево разбора и поток токенов (шаг 12)

Терминалы совпадают, переход к следующему символу.

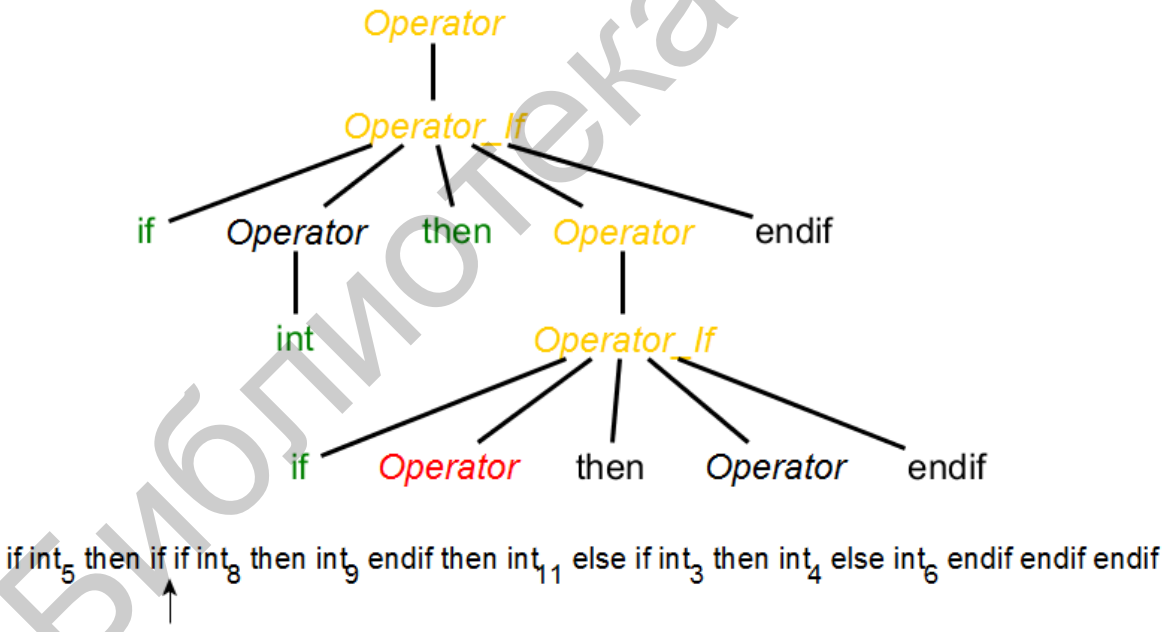


Рис. 22. Дерево разбора и поток токенов (шаг 13)

Нетерминал: дерево разбора дополняется по первой продукции.

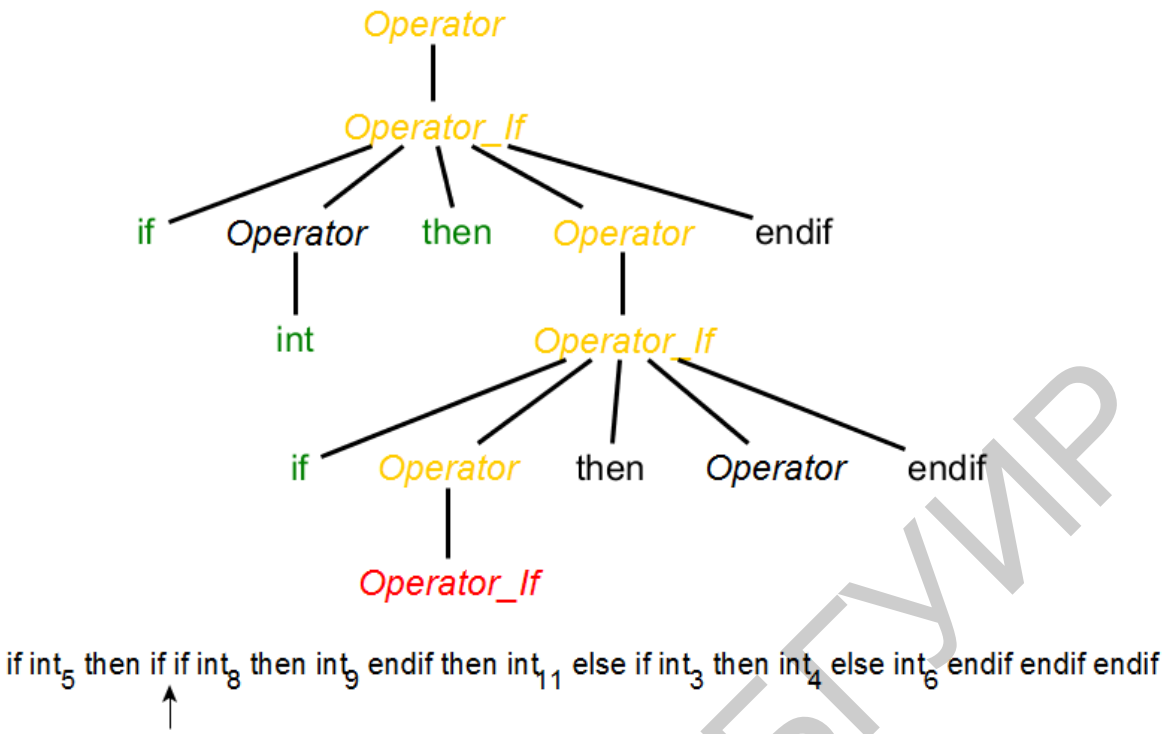


Рис. 23. Дерево разбора и поток токенов (шаг 14)

Нетерминал раскрывается по первой продукции, после чего на нескольких последующих шагах происходит дальнейшее раскрытие нетерминалов и совпадение терминалов до наступления ситуации, приведенной на рис. 24.

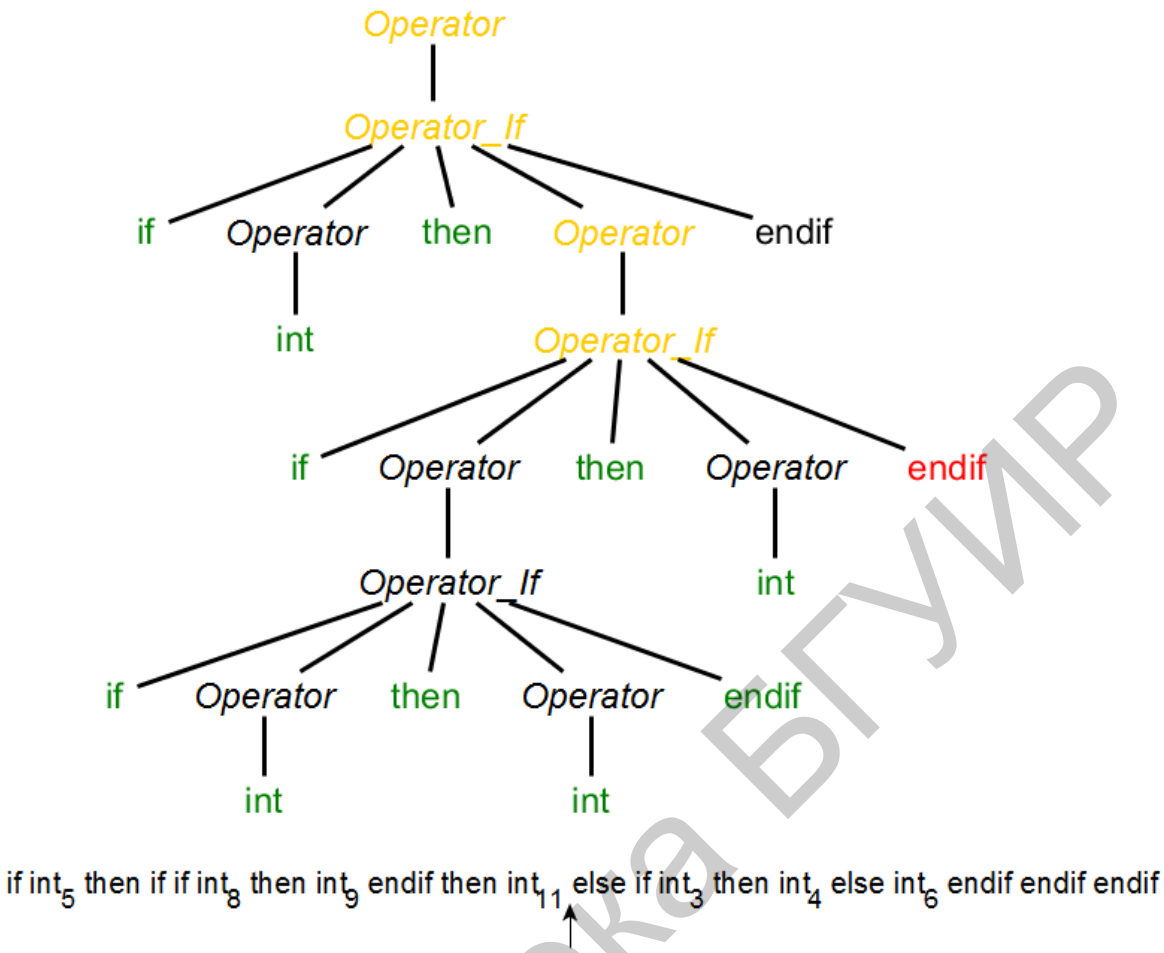


Рис. 24. Дерево разбора и поток токенов (перед откатом)

В данной ситуации терминал *else* из входного потока токенов не совпадает с ожидаемым терминалом *endif*. По правилам метода рекурсивного спуска необходимо сделать вывод о том, что выбор продукции, частью которой является ожидаемый терминал, был ошибочным и необходимо предпринять попытку применения следующей продукции.

Существенным недостатком метода рекурсивного спуска является необходимость выполнения подобных откатов. Например, в данном случае до момента отката было прочитано восемь терминалов и выполнено построение трех уровней поддерева, причем в ходе этого построения также производились откаты. Очевидно, что это увеличивает время, требуемое методу рекурсивного спуска для нахождения правильного порождения.

Следует также понимать, что решение о соответствии входного потока токенов грамматике может быть принято только при совпадении двух условий:

– прочитаны все символы входного потока;

– обработаны все узлы дерева разбора.

Для того чтобы принять решение о несоответствии грамматике, необходимо, чтобы в результате перебора всех возможных порождений ни одно из них не подошло входному потоку токенов.

Очевидно, это означает, что в общем случае методу рекурсивного спуска требуется просмотреть весь текст программы для того, чтобы оценить его принадлежность грамматике, причем из-за откатов одни и те же терминалы входного потока токенов будут просматриваться неоднократно, что особенно неэффективно в случаях, когда синтаксическая ошибка допущена в самом начале текста программы.

Вместе с тем важным преимуществом метода рекурсивного спуска является простота его реализации вручную.

Программная реализация метода рекурсивного спуска

Метод рекурсивного спуска в самом общем виде предполагает перебор всех возможных альтернатив на каждом шаге порождения, что делает время работы соответствующего синтаксического анализатора в худшем случае экспоненциальным.

Для того чтобы частично устранить эту проблему, на практике часто используют разновидность метода рекурсивного спуска, основанную на так называемых *грамматиках, разбирающих выражение* (parsing expression grammars, PEG). Отличительной особенностью таких грамматик по сравнению с обычными контекстно-свободными грамматиками является интерпретация продукций в рамках нетерминалов.

Так, в классических контекстно-свободных грамматиках все продукции одного *и того же нетерминала считаются равноправными независимо от*

порядка записи, вследствие чего обнаружение какого-либо порождения заданной строки не гарантирует отсутствия других порождений. В то же время PEG предполагают, что каждая следующая продукция нетерминала может быть применена только в том случае, если все предшествующие не подошли. Это делает невозможным построение неоднозначной PEG.

Кроме того, код для реализации разбора по PEG оказывается намного проще в написании.

Рассмотрим построение синтаксического анализатора, основанного на грамматике, разбирающей выражение. Для этого преобразуем ранее рассмотренную грамматику:

Operator → *Operator_If* | int
Operator_If → if *Operator* then *Operator* endif
| if *Operator* then *Operator* else *Operator* endif

Отличием нотации для PEG от классической нотации является замена операции альтернативы `|` на операцию упорядоченного выбора `/`. Перепишем грамматику с учетом этой особенности:

Operator → *Operator_If* / int
Operator_If → if *Operator* then *Operator* endif
/ if *Operator* then *Operator* else *Operator* endif

Реализация синтаксического анализатора на основе грамматики, разбирающей выражение, заключается в разработке набора процедур, рекурсивно вызывающих друг друга, причем каждая из них соответствует одной продукции либо одному из символов (терминалов либо нетерминалов) грамматики.

Введем вспомогательный тип данных:

```
type
  TTokenType = (ttInt, ttIf, ttThen, ttElse, ttEndIf);
  PTokenType = ^TTokenType;
```

Последовательность токенов будем рассматривать как массив, для работы с которым используется указатель на текущий элемент. При разработке полноценного компилятора целесообразнее предусмотреть структуру данных,

реализующую потоковый интерфейс, для более эффективного взаимодействия между лексическим и синтаксическим анализаторами.

```
var
  Next: PTokenType;
```

Для всех терминалов грамматики будем использовать общую функцию, задача которой заключается в сопоставлении ожидаемого в соответствии с грамматикой терминала и фактически имеющегося во входном потоке токена.

```
function Term(const Expected: TTokenType): Boolean;
begin
  Result := (Next^ = Expected);
  Inc(Next);
end;
```

Для каждого нетерминала разработаем функции, которые будут последовательно перебирать продукции до тех пор, пока либо не будет найдена подходящая, либо не окажется, что подходящих продукций нет.

```
function Operator: Boolean;
var
  Save: PTokenType;
begin
  Save := Next;

  Result := True;
  if Operator_1() then Exit;
  Next := Save;
  if Operator_2() then Exit;
  Result := False;
end;

function OperatorIf: Boolean;
var
  Save: PTokenType;
begin
  Save := Next;

  Result := True;
  if OperatorIf_1() then Exit;
  Next := Save;
  if OperatorIf_2() then Exit;
  Result := False;
end;
```

Для каждой продукции разрабатывается отдельная функция, отражающая структуру этой продукции. Здесь и далее предполагается, что в настройках проекта задано сокращенное вычисление логических выражений. Того же эффекта можно достигнуть указанием директивы `{$B-}`.

```
function Operator_1: Boolean;
begin
  Result := OperatorIf();
end;

function Operator_2: Boolean;
begin
  Result := Term(ttInt);
end;

function OperatorIf_1: Boolean;
begin
  Result := Term(ttIf)
        and Operator()
        and Term(ttThen)
        and Operator()
        and Term(ttEndIf);
end;



function OperatorIf_2: Boolean;
begin
  Result := Term(ttIf)
        and Operator()
        and Term(ttThen)
        and Operator()
        and Term(ttElse)
        and Operator()
        and Term(ttEndIf);
end;
```

Теперь для запуска синтаксического анализатора достаточно:

– установить указатель `Next` на начало потока токенов;

– вызвать функцию, соответствующую стартовому символу грамматики (*Operator*).

Приведенный анализатор выполняет только проверку на соответствие последовательности токенов грамматике. При этом дерево разбора строится неявно в форме последовательности рекурсивных вызовов.

Поскольку при написании компилятора, как правило, предпочтительнее выполнять построение абстрактного синтаксического дерева, структура которого определяется решаемой задачей, в данном учебно-методическом пособии полный код такого анализатора не приводится. Тем не менее основная

идея заключается в том, что при возврате `True` одной из функций, соответствующих продукциям, фактически происходит применение такой продукции, а значит, именно в эти моменты следует достраивать AST. Одна из возможных реализаций заключается в том, чтобы при успешном применении продукции возвращать указатель на корень построенного поддерева, а в случае если продукция не подошла, возвращать нулевой указатель `nil`.

Левая рекурсия

Одним из слабых мест нисходящих методов синтаксического разбора (в том числе всех разновидностей метода рекурсивного спуска, включая PEG) является уязвимость к левой рекурсии. *Леворекурсивная грамматика* – такая грамматика, в которой существует вывод (фрагмент порождения) вида $A \to^* A\beta$, где A – некоторый нетерминал, β – произвольная последовательность терминалов и нетерминалов.

Рассмотрим следующую грамматику:

$S \to S\text{a} \mid \text{b}$

При попытке разработки синтаксического анализатора по методу рекурсивного спуска (в том числе если считать грамматику разбирающей выражение, т. е. заменить альтернативу упорядоченным выбором) получится следующий код:

```
type
  TTokenType = (ttA, ttB);
  PTokenType = ^TTokenType;

var
  Next: PTokenType;

function S: Boolean;
var
  Save: PTokenType;
begin
  Save := Next;

  Result := True;
  if S_1() then Exit;
  Next := Save;
  if S_2() then Exit;
  Result := False;
end;

function S_1: Boolean;
begin
  Result := S() and Term(ttA);
end;

function S_2: Boolean;
begin
  Result := Term(ttB);
end;
```

Очевидной проблемой является взаимная рекурсия между функциями `S` и `S_1`. Для запуска анализатора производится вызов функции `S`, которая после двух присваиваний всегда вызывает функцию `S_1`. Функция же `S_1` сразу после получения управления вызывает функцию `S`. Таким образом, запуск такого анализатора приведет к бесконечной рекурсии, а его выполнение завершится аварийно при переполнении стека.

Как и неоднозначность, левая рекурсия является свойством грамматики и устраняется путем переписывания грамматики. В приведенном примере грамматика может быть переписана следующим образом:

$S \rightarrow b\mathrm{T}$
$T \rightarrow \mathrm{a}T \mid \varepsilon$

В более общем случае леворекурсивная грамматика имеет следующий вид:

$S \rightarrow S\alpha_1 \mid \ldots \mid S\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$

Такая грамматика фактически задает строки, которые начинаются с одной из подстрок β_1, …, β_m, за которой следует некоторое количество подстрок α_1, …, α_n.

После устранения левой рекурсии такая грамматика примет следующий вид:

$S \rightarrow \beta_1 S' \mid \ldots \mid \beta_m S'$
$S' \rightarrow \alpha_1 S' \mid \ldots \mid \alpha_n S' \mid \varepsilon$

Помимо уже приведенного случая, где левая рекурсия очевидна, на практике встречаются менее очевидные случаи, в том числе косвенная левая рекурсия.

$S \rightarrow A\alpha \mid \delta$
$A \rightarrow S\beta$

Общий алгоритм устранения левой рекурсии приведен в [2]. Тем не менее более правильным подходом следует считать изначальное написание грамматики с учетом этой проблемы. В частности, для большинства синтаксических конструкций языков программирования раскрытие нетерминалов удается начинать с терминальных элементов – ключевых слов, причем в этом случае зачастую удается одновременно провести и левую факторизацию.

Левая факторизация

Под *левой факторизацией* понимается такое преобразование грамматики, в результате которого для каждого нетерминала грамматики на каждом шаге порождения выбор продукции становится единственным. Пусть имеется некоторая грамматика:

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

Очевидно, что для принятия решения о том, какую из продукций следует использовать, необходимо полностью просмотреть часть входной строки, соответствующую подстроке α. Однако для раскрытия нетерминала выбор одной из продукций необходимо выполнить в момент, когда просмотрены только символы, предшествующие α. Таким образом, существует две возможности:

– использовать предпросмотр до тех пор, пока не станет понятно, какую из продукций следует выбрать;

– выбрать одну из продукций, а затем, если выбор окажется неправильным, выполнить откат и изменить выбор.

Ни один из предложенных вариантов решения проблемы не является оптимальным: предпросмотр в общем случае может оказаться неограниченным (если α содержит синтаксические конструкции, допускающие неограниченное повторение или иным образом порождающие строки неограниченной длины), а в случае использования откатов цена ошибки окажется достаточно высокой.

Для устранения проблемы грамматика переписывается следующим образом:

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$

В данном случае проблема устраняется за счет того, что выбор откладывается до момента, когда он может быть выполнен, а на начальном этапе остается единственная продукция.

Для метода рекурсивного спуска левая факторизация в общем случае не является обязательной, поскольку в любом случае предполагается перебор всех возможных порождений. Тем не менее большинство используемых на практике методов нисходящего разбора требует выполнения этого преобразования:

– при использовании PEG возможность выбора неправильной продукции приведет к многочисленным откатам и существенному снижению производительности;

– при использовании методов предиктивного анализа невыполнение левой факторизации делает невозможным построение таблицы разбора, за исключением простейших случаев.

Очевидным недостатком левой факторизации является, как и в случае с устранением левой рекурсии, повышение сложности грамматики. В частности, в дереве разбора появляются промежуточные узлы, как правило, не несущие практической пользы. Эта проблема тем не менее сглаживается применением абстрактных синтаксических деревьев.

Задание

Для заданного вариантом языка разработать:

– **лексический анализатор**, распознающий необходимые лексемы;

– **грамматику** языка, включающую в себя базовые конструкции языка;

– **программное средство**, проверяющее исходный код программы на соответствие грамматике.

Лексический анализатор можно разработать вручную или использовать утилиты-генераторы. Поддерживаемое подмножество языка согласовать с преподавателем.

Таблица 9

Варианты заданий к лабораторной работе №3

Номер варианта	**Язык/формат**	**Примечания**
1	Pascal/Delphi	
2	C/C++	
3	Basic	
4	C#/Java	
5	Python	
6	SQL	
7	JSON	
8	HTML	
9	PHP	
10	Markdown	
11	Go	
12	Rust	
13	Ада	
14	Модула	
15	Lisp	
16	Erlang	

Литература

1. Свердлов, С. З. Языки программирования и методы трансляции : учеб. пособие / С. З. Свердлов. – СПб. : Питер, 2007. – 638 с.

2. Компиляторы: принципы, технологии и инструментарий / А. В. Ахо [и др.]. – 2-е изд. ; пер. с англ. – М. : Изд. дом «Вильямс», 2008. – 1184 с.

3. Маккиман, У. Генератор компиляторов / У. Маккиман, Дж. Хорнинг, Д. Уортман ; пер. с англ. С. М. Круговой ; под ред. и с предисл. В. М. Савинкова. – М. : Статистика, 1980. – 527 с.

4. Appel, A. W. Modern Compiler implementation in C / A. W. Appel, M. Ginsburg. – Rev. and expanded. – C. : Cambridge University Press, 1998. – 544 p.

Св. план 2018, поз. 37

Учебное издание

Шостак Елена Викторовна
Марина Ирина Михайловна
Оношко Дмитрий Евгеньевич

ОСНОВЫ ПОСТРОЕНИЯ ТРАНСЛЯТОРОВ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

Редактор *Е. И. Костина*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *М. В. Касабуцкий*

Подписано в печать 20.03.2019. Формат 60x84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 4,07. Уч.-изд. л. 4,0. Тираж 100 экз. Заказ 1.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
ЛП №02330/264 от 14.04.2014.
220013, Минск, П. Бровки, 6