

Министерство образования и науки Российской Федерации
ФГАО УВО “Севастопольский государственный университет”



МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к выполнению лабораторного практикума по дисциплине
“Теоретические основы построения компиляторов”
для студентов основного профиля
09.03.02 – “Информационные системы и технологии”
всех форм обучения



Севастополь
2015

УДК 004.4

Методические указания к выполнению лабораторно - вычислительного практикума по дисциплине “Теоретические основы построения компиляторов” для студентов основного профиля 09.03.02 – “Информационные системы и технологии” всех форм обучения / Разраб. В.Ю. Карлусов, С.А. Кузнецов. – Севастополь: Изд-во СевГУ, 2015. – 36 с.

Цель методических указаний: выработка у учащихся необходимых исследовательских и практических навыков применения основных положений теорий конечных автоматов, формальных грамматик и синтаксического анализа и компиляции при решении задач, связанных с построением трансляторов.

Методические указания рассмотрены и утверждены на заседании кафедры Информационных систем,
протокол № 01 от 03 февраля 2014 г.

Допущено учебно-методическим центром СевГУ в качестве методических указаний.

Рецензент Кожаев Е.А., кандидат техн. наук, доцент кафедры кибернетики и вычислительной техники.

СОДЕРЖАНИЕ

Введение	4
1. Лабораторная работа № 1. “Исследование детерминированного конечного автомата”	4
2. Лабораторная работа № 2. “Исследование сканера при анализе простых языковых конструкций”	9
3. Лабораторная работа № 3. “Исследование алгоритма нисходящего распознавателя языка $LL(k)$ - грамматики”	12
4. Лабораторная работа № 4. “Исследование алгоритма восходящего анализатора языка грамматики простого предшествования”	16
Заключение	21
Библиографический список	21
Приложение А. Варианты заданий к работе № 1.	23
Приложение Б. Варианты заданий к работам № 2 – 4.	24
Приложение В. Дидактические материалы к работе № 4.	30

ВВЕДЕНИЕ

Одним из немаловажных компонентов информационных технологий является компьютерный анализ синтаксиса текстов. К настоящему времени элементы его присутствуют практически повсеместно: в трансляторах языков программирования – от ассемблеров до объектно-ориентированных алгоритмических языков, контроля орфографии редактора “Word”, с использованием которого набраны настоящие методические указания, разнообразных переводчиков типа Prompt, РУМП и им подобных, наконец, квинтэссенции информационных систем – системах искусственного интеллекта. Поэтому трансляторы являются важной практической областью научных исследований, связанных с изучением ЭВМ[4].

В настоящем практикуме изучаются методы, ориентированные на использование теории формальных языков и грамматик при решении задач распознавания синтаксиса. Вопросы генерации объектного кода и его оптимизации опущены по следующим соображениям: наряду со значительными объёмами текстов соответствующих программ, алгоритмы генерации и оптимизации [6, 13] кодов обладают известной степенью инвариантности и универсализма, что, в немалой степени, препятствует формированию нетривиальных заданий при их изучении и исследовании. Последнее обстоятельство, как известно, толкает студента на путь формалистики и поверхностного изучения дисциплины.

Варианты практических заданий сформулированы на основании [8, 10, 14], примеры аналитических преобразований и вычислений, необходимых в ходе выполнения работ, помещены в [10 – 12].

ЛАБОРАТОРНАЯ РАБОТА № 1.

“ИССЛЕДОВАНИЕ ДЕТЕРМИНИРОВАННОГО КОНЕЧНОГО АВТОМАТА”

1. Цель работы

1.1. Научиться производить построение детерминированных конечных автоматов (ДКА), допускающих определённые цепочки символов языка.

1.2. Освоить приёмы описания конечных автоматов (КА) в виде графов, таблиц переходов и регулярных выражений.

1.3. Научиться выполнять построения ДКА по недетерминированным конечным автоматам (НКА).

1.3. Научиться проводить построение минимальных детерминированных конечных автоматов (МДКА).

2. Теоретические сведения

Конечным автоматом (КА), под которым, в дальнейшем, будем понимать конечный автомат Мили, формально называют совокупность следующих объектов [15, 16, 19]:

$$A (Q, \Sigma, \delta, q_0, F),$$

где Q – конечное множество состояния автомата; Σ - алфавит входных символов (букв или литер); δ - функция переходов КА, определяемая на декартовом произведении $Q \otimes \Sigma$; $q_0 \in Q$ – начальное состояние КА; $F \subset Q$ – множество заключительных состояний КА. Автомат функционирует на основании следующих эмпирических правил [15].

1. Исходным состоянием КА является начальное состояние q_0 .
2. Цепочка, составленная из литер алфавита Σ , по одной букве поступает на вход КА. Возврат к начальным литерам цепочки не возможен.
3. Литера S_j допускается КА, находящимся в состоянии q_i , если $\delta(q_i, S_j) \neq \{\}$ – иными словами, образ функции переходов не пуст, определён переход из текущего состояния КА в следующее.
4. Если литера входной цепочки не допускается, то цепочка отвергается, признаётся не принадлежащей входному языку КА, сам автомат остаётся в текущем состоянии.
5. Цепочка литер допускается КА, если после поступления её последнего символа автомат оказывается в одном из заключительных состояний $q_r \in F$. В противном случае цепочка отвергается.

Известны теоремы [16], устанавливающие соответствие между классами конечных автоматов и классами цепочек, называемыми регулярными, которые могут быть проверены на принадлежность языку, порождаемому определённой грамматикой, с помощью КА.

Элементы КА могут быть построены и описаны несколькими способами [7 – 9, 15, 16], указанные процедуры излагаются и в методической литературе [10, с. 9 – 17; 11, с. 23 - 33].

Одним из эффективных способов описания КА является регулярное выражение, особенно когда известен вид цепочек, подлежащих опознаванию.

Правила написания регулярных выражений для типовых конструкций таковы [10, 16], при этом фигурные скобки обозначают итерацию, то, что в них заключено, может повторяться от нуля до бесчисленного числа раз. круглые или квадратные – используют для объединения альтернатив, из которых одна обязательно присутствует.

Пусть имеется алфавит, составленный из символов (букв, литер) $V_T = \{x_1, x_2, \dots, x_n\}$.

1. Множество всевозможных цепочек, составленных из букв $x_i \in V_T$:

$$L = \{x_1 \vee x_2 \vee x_3 \vee \dots \vee x_n\}.$$

2. Множество цепочек, составленных из литер $x_i \in V_T$ и оканчивающихся литерой x_l .

$$L = \{x_1 \vee x_2 \vee x_3 \vee \dots \vee x_n\} \wedge x_l.$$

3. Множество цепочек, составленных из литер $x_i \in V_T$ начинающихся цепочкой l_1 и оканчивающихся l_2 .

$$L = l_1 \wedge \{x_1 \vee x_2 \vee x_3 \vee \dots \vee x_n\} \wedge l_2.$$

4. Множество однолитерных цепочек (однобуквенных слов) совпадает с алфавитом

$$L = x_1 \vee x_2 \vee x_3 \vee \dots \vee x_n.$$

5. Множество двулитерных цепочек (двухбуквенных слов)

$$L = (x_1 \vee x_2 \vee x_3 \vee \dots \vee x_n) \wedge (x_1 \vee x_2 \vee x_3 \vee \dots \vee x_n).$$

6. Множество m – буквенных слов

$$L = (x_1 \vee x_2 \vee x_3 \vee \dots \vee x_n) \wedge \dots \wedge \underbrace{(x_1 \vee x_2 \vee x_3 \vee \dots \vee x_n)}_{m-2}$$

С точки зрения техники реализации КА на ЭВМ, предпочтительнее является его описание в виде таблиц переходов и выходов, иногда совмещённых, первая из них есть табличный аналог δ , символам входного алфавита соответствуют строки, а состояниям – столбцы.

Иногда функция переходов определена неоднозначно, в этом случае КА будет недетерминированным. При этом необходимо получить детерминированный конечный автомат (ДКА) по недетерминированному КА. Приёмы построения ДКА по недетерминированному КА, иллюстрируются многочисленными примерами, приводимыми в [10, с. 10 – 13]. Известен [10, 16, 19] следующий алгоритм построения ДКА по НКА.

1. Рассмотрим функцию переходов.

Для случаев, когда значение функции определено неоднозначно, введём новое состояние, которое является объединением состояний, образующих неоднозначность, во множество состояний автомата,.

2. Для вновь введенного состояния строим функцию переходов; анализируя функцию переходов в состояниях, определяющих данное состояние.

3. Повторяем п.п. 1 – 2 до тех пор, пока множество состояний автомата не будет изменено, а функция переходов приобретёт однозначность.

Технически оправдано иметь таблицу переходов КА минимального размера. Очевидно, что число строк таблицы изменению не подлежит, как определяемое внешними условиями, а число состояний КА и, соответственно, количество столбцов, может быть уменьшено в соответствии с теоремой Майхилла – Нерода [16]. Примеры такого упрощения приводятся в [10, с. 15 – 21; 11, с. 23 – 27]. Программная реализация КА может быть различной [1, с. 60 – 61, 86; 10, с. 15; 11, с. 26 – 37], тем не менее, неизменно наличие следующих этапов.

1. Начальное состояние – нулевое, соответствующий столбец – первый, он же является текущим.

2. Выполнение действий, связанных с обработкой текущего состояния КА.

3. Определение строки управляющей таблицы, соответствующей поступившей литере.

4. Определение следующего состояния КА и соответствующего номера столбца.

5. Пункты 2 – 3 выполняются, пока не будет достигнуто одно из заключительных состояний, после чего принимается решение о принадлежности или непринадлежности к одному из множеств цепочек.

3. Ход работы

Выполнение работы занимает два аудиторных занятия (4 академических часа) и предполагает интенсивную домашнюю подготовку.

3.1. Домашняя подготовка. На основании кода варианта в виде регулярного выражения, полученного у преподавателя, взять регулярное выражение из приложения А. Выполнить разметку регулярного выражения согласно [10], по необходимости минимизировать. Если КА не является детерминированным, необходимо привести его к детерминированному виду. Построить таблицу переходов МДКА. Выполнить минимизацию по таблице переходов. Построить граф конечного автомата для визуального определения недостижимых состояний, и, если таковые обнаружатся, окончательно минимизировать ДКА, построив окончательную таблицу МДКА. Написать черновой вариант программы на языке C (C⁺⁺).

3.2. Занятие 1 (2 часа). Предъявить результаты домашней подготовки для проверки преподавателю, устранить ошибки согласно замечаниям преподавателя. Ввести программу, реализующую КА, довести её до успешной компиляции.

3.3. Домашняя подготовка. Разработать тестовые последовательности литер, допускаемые и отвергаемые КА. Найти допускаемые последовательности минимальной длины. Предусмотреть соответствующие диагностические сообщения. Каждая цепочка литер должна быть снабжена соответствующей последовательностью вершин КА, которую он проходит в процессе анализа цепочки.

Подготовить пункты отчёта, касающиеся выполненной работы.

3.4. Занятие 2 (2 часа). Предъявление материалов тестирования преподавателю и отладка программы. Окончательное оформление отчёта и защита результатов выполнения лабораторной работы.

4. Содержание отчёта

4.1. Вариант задания.

4.2. Аналитические выкладки по проведению минимизации и, по необходимости построения ДКА, управляющая таблица КА.

4.3. Тестовые примеры для отладки включая допускаемые последовательности минимальной длины.

4.4. Листинги программы, результаты выполнения тестирования и граф конечного автомата.

4.5. Содержательные выводы, в которых отражено: изменение числа состояний КА в ходе выполнения лабораторной работы; размер исходного модуля (в строках программного кода), размер исполнимого модуля (в байтах).

5. Вопросы для самоконтроля и самоподготовки

5.1. Определение, способы описания и построения конечных автоматов.

5.2. Формулировки теорем о связи КА и формальной грамматики, соответствия недетерминированного КА и детерминированного КА.

5.3. Привести алгоритм построения ДКА по НКА.

5.4. Теорема Майхилла – Нерода.

5.5. Минимизация КА по регулярному выражению и таблицам переходов и выходов.

5.6. Что такое конечный автомат?

5.7. Какие гипотезы положены в основу функционирования КА?

5.8. Какой КА называется недетерминированным?

5.9. Правила написания регулярных выражений.

5.10. Какая цепочка входных литер допускается КА?

5.11. Какая цепочка входных литер не допускается КА?

5.12. В чём заключается отношение эквивалентности применительно к цепочкам, обрабатываемым конечным автоматом?

5.13. Почему для программной или схемной реализации КА необходимо его приведение к детерминированному виду?

5.14. Привести примеры задач, решаемых с применением КА [19].

ЛАБОРАТОРНАЯ РАБОТА № 2. “ИССЛЕДОВАНИЕ СКАНЕРА ПРИ АНАЛИЗЕ ПРОСТЫХ ЯЗЫКОВЫХ КОНСТРУКЦИЙ”

1. Цель работы

1.1. Изучить принципы построения и программирования лексического анализатора на языке С (C⁺⁺) для простых языковых конструкций.

1.2. Получить навыки практического построения лексического анализатора (сканера) на основе теории конечных автоматов.

1.3. Освоить приёмы составления регулярных выражений для описания лексем.

1.4. Закрепить навыки построения минимального КА, осуществляющего сканирование текста программ.

2. Теоретические сведения

Из общей структуры компилятора [3 – 4; 5, с. 19] видно, что лексический анализатор (сканер) занимает заметное место в аналитической части транслятора и является первой (по времени начала выполнения) системной программой, анализирующей исходный модуль пользователя.

Сканер получил свое наименование благодаря особенности обработки входной информации. В его задачу входит: анализ исходной программы на уровне лексем [7, 9, 18]; выдача диагностических сообщений об обнаруженных ошибках; построение информационных таблиц констант, идентификаторов, служебных слов, разделителей; перекодирование исходной программы в соответствии с построенными таблицами.

Входной информацией сканера является литеры исходной анализируемой программы, а выходной - внутренняя, перекодированная форма программы и таблицы лексем [7]. Для построения сканера широко используют регулярные выражения и конечные автоматы, которые описываются регулярными (автоматными грамматиками) [15, 16, 19]. К сожалению, возможности регулярных языков ограничены. Поэтому роль грамматик, порождающих регулярные языки, сведена к распознаванию лексем, используемых в программе [15]. Указанный этап компиляции получил название лексического анализа.

Построение сканера наиболее формально и точно можно осуществить по регулярному выражению, при этом творческая компонента присутствует при составлении регулярного выражения, уступая затем место строгим алгоритмам.

Примеры построения регулярного выражения и конечного автомата на его основе, осуществляющего процедуру лексического анализа, приводятся в [10, с. 22 – 24; 11, с. 33 - 36].

Возможен подход к построению сканера на основании диаграммы состояний (графа) автомата. Указанный подход проектирования является, по сути, эмпирическим, его ясное и подробное изложение выполнено Д. Грисом [7]. За исключением построения управляющей таблицы КА сканера, в нём соблюдается вся методология проектирования программных систем [13, 17], что вполне может быть использовано студентами при выполнении настоящей лабораторной работы.

3. Ход работы

Выполнение работы занимает два аудиторных занятия (4 академических часа) и предполагает интенсивную домашнюю подготовку.

3.1. Домашняя подготовка. Согласно выданному заданию составить регулярное выражение, описывающее КА, который осуществляет лексический анализ.

Структура такого выражения примерно такова

$$Cc_1L_1 \cup Cc_2L_1 \cup \langle iden \rangle L_1 \cup \langle data \rangle L_2 \cup d_1 \cup d_2 \cup \dots \cup d_j \cup \dots \cup d_n \cup L_3.$$

В записи обозначено:

Cc_i – i -тое служебное слово.

$\langle iden \rangle$ – описание переменной.

$\langle data \rangle$ – описание константы или литерала.

L_1 – лексема, состоящая в том, что текущая литера не буква и не цифра.

L_2 – текущая литера не принадлежит к образующим константу или литерал.

d_j – разделители j -того типа.

L_3 – литера, не принадлежащая алфавиту конечного автомата.

В ходе домашних эскерциций составить эскизные выражения для описания дизъюнктивных компонент. Построить управляющую таблицу МДКА, Продумать модернизацию программы из предыдущей работы под нужды лексического анализа и текст диагностических сообщений, генерируемых в его ходе. Разработать схему кодирования лексем.

3.2. Занятие 1 (2 часа). Согласовать текст диагностических сообщений с преподавателем, сконструировать текст программы сканера на основании программы МДКА с учётом домашней подготовки. Выявить и устранить синтаксические ошибки.

3.3. Домашняя подготовка. Разработать тестовые примеры правильных и неправильных лексем языка, уточнить диагностические сообщения. Продумать

структуру выходной информации программы на предмет использования другими программными модулями компилятора. Подготовить пункты отчёта.

3.4. Занятие 2 (2 часа). Тестировать и окончательно отладить программу. Экспериментально определить время работы сканера для качественно различных структур строк и характера ошибок. Завершить оформление отчёта и защитить результаты выполнения лабораторной работы.

4. Содержание отчёта

4.1. Вариант задания.

4.2. Регулярное выражение КА, осуществляющего лексический анализ; аналитические преобразования, выполненные при построении детерминированного минимального КА; управляющая таблица КА.

4.3. Тестовые примеры и тексты диагностических сообщений.

4.4. Листинги сервисных процедур и результатов тестирования.

4.5. Осмысленные выводы, в которых должно быть отражено: изменение объёмов (в символах) входной и выходной информации; размер исходного модуля (в строках программного кода), размер исполнимого модуля (в байтах).

5. Вопросы для самоконтроля и самоподготовки

5.1. Назначение сканера и его место в схеме трансляции.

5.2. Типы лексем и система их кодирования.

5.3. Виды таблиц, создаваемых и используемых лексическим анализатором и способы их организации [7, 9, 18].

5.4. Характерные особенности функционирования лексического анализатора.

5.5. Методология построения лексических анализаторов.

ЛАБОРАТОРНАЯ РАБОТА № 3.
“ИССЛЕДОВАНИЕ АЛГОРИТМА НИСХОДЯЩЕГО РАСПОЗНАВАТЕЛЯ
ЯЗЫКА $LL(k)$ - ГРАММАТИКИ”

1. Цель работы

1.1. Изучить процедуру синтаксического анализа, основанную на осуществлении рекурсивного спуска.

1.2. Приобрести навыки в исследовании грамматики на принадлежность к классу $LL(k)$.

1.3. Освоить методы синтеза формальных грамматик класса $LL(k)$ и, по необходимости, преобразования грамматики к этому классу

2. Теоретические сведения

Входной информацией для синтаксического анализатора является последовательность кодов лексем, соответствующая тексту исходной программы, которую готовит сканер (смотри лабораторную работу № 2). На этапе синтаксического анализа проверяется правильность программного кода: является ли анализируемая цепочка кодов лексем одним из возможных предложений языка, порождаемого некой формальной грамматикой. Ныне эта задача является хорошо проработанной [2 – 4, 6 – 9, 19], известно большое число алгоритмов синтаксического анализа. Различают нисходящий (сверху вниз) и восходящий (снизу вверх) методы разбора.

В качестве основного недостатка алгоритма простого нисходящего разбора указывается [7, с. 115 – 120] отсутствие прогноза при выборе возможного правила формальной грамматики из множества альтернатив для построения дерева. Следствием неправильного определения правила, гипотетически использованного при выводе и ведущего к неудаче, последующее восстановление правильного хода вывода, приводит к непроизводительным затратам ресурсов ЭВМ и времени выполнения трансляции, варьируемым в широких пределах.

Таким образом, проблемой решаемой на каждом шаге разбора является определение очередного правила подстановки. В этом случае, если по k проанализированным символам исходной программы можно однозначно определить, какую продукцию грамматики необходимо использовать, грамматика называется $LL(k)$ - грамматикой и, в этом случае, применяют алгоритм рекурсивного спуска [5, 16], а сам алгоритм разбора относится к синтаксически управляемым прогнозирующим разборам.

Одной из особенностей данной схемы трансляции является её жёсткая ориентация на конкретную грамматику, поэтому метод относят к группе специализированных методов. При этом на каждый символ формальной грамматики, как терминальный, так и нетерминальный, разрабатывается алгоритм, и пишется процедура его распознавания. Порядок вызова процедур определяется местоположением распознаваемых символов в правилах исходной грамматики. Поэтому алгоритм относят к группе методов синтаксических функций [2, 9, 18].

Исследование грамматики на принадлежность к классу $LL(k)$ состоит в отыскании уникальных цепочек терминальных символов, выводимых из головы правой части правила. Изложение алгоритма, иллюстрация его применения и примеры для закрепления приводятся в [10, с. 24 – 26; 11, 38 – 40]. Там же приводится текст программы на С (C^{++}). Заметим, что рекурсивные вызовы при работе распознавателя могут быть организованы как явным, так и неявным способами. При кажущейся простоте алгоритма, он весьма эффективен и широко распространён: достаточно отметить, что компилятор с алгоритмического языка Pascal построен по данной схеме на основании свойств $LL(1)$ – грамматики. Как недостаток можно отметить однопроходовость такого транслятора: его работа завершается нахождением первой встреченной от начала программы ошибки. Обычно отдают предпочтение $LL(1)$ – грамматике, что упрощает процедуру обработки анализируемого фрагмента строки, а $LL(k)$ – грамматики с большим индексом k используют вынужденно.

В качестве входной информации нисходящий рекурсивный распознаватель использует перекодированную программу [7, с. 19], принимаемую после работы сканера.

3. Ход работы

Выполнение работы занимает два аудиторных занятия (4 академических часа) и предполагает интенсивную домашнюю подготовку.

3.1. Домашняя подготовка. Для Вашего варианта задания, согласно рекомендации [2, 4, 5, 7 – 12], по образу и подобию [1] описания языков программирования в форме Бекуса – Наура, выполнить построение формальной грамматики. Особое внимание следует на устранение явной левой рекурсии в правилах, паче таковая непременно возникнет:

Исходные правила

$A \rightarrow a | Aa$

Преобразованные правила

$A \rightarrow a | aB$

$B \rightarrow a | aB.$

Чтобы убедиться в том, что формальная грамматика составлена корректно, следует с её помощью построить синтаксическое дерево. Если же дерево построить не удастся, то, изменяя правила грамматики и добавляя новые пра-

вила к уже существующим правилам, необходимо добиться правильного построения.

Для большей устойчивости работы анализатора рекомендуется преобразовать грамматику в нормальную форму Грейбах [10, с. 37 – 41; 11, с.9 - 10; 16], правда, подобное преобразования приведёт к заметному увеличению количества правил грамматики и размера информационных структур, её отображающих. Таких как общий объём программного кода при использовании метода рекурсивного спуска, либо количества операций в устройстве управления в МП-распознавателе.

3.2. Далее необходимо преобразовать полученную Вами грамматику к классу $LL(1)$ – грамматик, свободную от левой рекурсии.

Процедура исследования грамматики на свойства $LL(k)$ подробно и на примерах изложена в [10, 11]. Маловероятно, но возможно, что построенная грамматика сразу окажется принадлежащей к классу $LL(k)$.

Если грамматика не является $LL(1)$ – грамматикой, то её необходимо преобразовать к такому виду [7]. Типичной проблемой, препятствующей отнесению грамматики к классу $LL(1)$, является наличие альтернативных правых частей правил с одинаковыми головами. Преобразование заключается во введении новых правил.

Исходные правила	Преобразованные правила
$P \rightarrow \langle iden \rangle + \langle data \rangle$	$P \rightarrow \langle iden \rangle R \langle data \rangle$
$P \rightarrow \langle iden \rangle - \langle data \rangle$	$R \rightarrow + -$

Рекурсия тоже препятствует причислению грамматики к этому классу. Преобразование заключается в использовании итерационного повторителя $\{ \}$, который программно осуществляется итерационным циклом типа `while` или `until`.

Исходные правила	Преобразованные правила
$\langle spisok \rangle \rightarrow \langle iden \rangle ; \langle spisok \rangle \langle iden \rangle$	$\langle spisok \rangle \rightarrow \langle iden \rangle \{ ; \langle iden \rangle \}$

Составить текст программы, реализующей рекурсивный нисходящий алгоритм разбора. **Предусмотреть** средства хронометража и фиксации времени выполнения синтаксического анализа. Продумать диагностические сообщения, которые должны выдаваться при возникновении синтаксических ошибок.

3.3. Занятие 1 (2 часа). Предъявить формальную $LL(1)$ – грамматику, совместно с доказательством её свойств, преподавателю. Ввести программу в компьютер, добиться успешной компиляции.

3.4. Домашняя подготовка. Модифицировать грамматику согласно результатам обсуждения с преподавателем, если возникнет необходимость. Подготовить диагностические сообщения и определить места их размещения в тексте анализатора. Продумать ход тестирования программы и разработать тесты. Подготовить пункты отчёта.

3.4. Занятие 2 (2 часа). Согласовать тесты с преподавателем. Осуществить отладку логики рекурсивного анализатора. Предъявить материалы выполнения.

Окончательно оформить отчёт и защитить результаты выполнения лабораторной работы.

4. Содержание отчёта

4.1. Вариант задания.

4.2. Ход построения формальной грамматики, порождающей фрагменты программы, соответствующие заданному варианту, синтаксическое дерево, исследования на свойства $LL(1)$, преобразования грамматики.

4.3. Текст программы рекурсивного нисходящего распознавателя.

4.4. Тестовые примеры и тексты диагностических сообщений.

4.5. Листинг результатов тестирования .

4.5. Осмысленные выводы, в которых должно быть отражено: особенности полученной Вами грамматики; показатели затраты времени на обработку тестов; размер исходного модуля (в строках программного кода), размер исполнимого модуля (в байтах).

5. Вопросы для самоконтроля и самоподготовки

5.1. Назначение синтаксического разбора.

5.2. Принцип работы нисходящего распознавателя.

5.3. Основные недостатки простого нисходящего разбора.

5.4. Способы и особенности организации данных для осуществления нисходящего разбора.

5.5. Формулировка теорем о преобразовании формальной грамматики в нормальную форму Грейбах [16].

5.6. Приёмы преобразования грамматик, направленные на устранения левой рекурсии.

5.7. Процедура определения принадлежности грамматики к классу $LL(k)$.

5.8. Сущность и особенности метода синтаксических функций.

5.9. Преимущества нисходящего разбора, основанного на свойствах $LL(k)$ грамматик перед обычным нисходящим разбором.

ЛАБОРАТОРНАЯ РАБОТА № 4. “ИССЛЕДОВАНИЕ АЛГОРИТМА ВОСХОДЯЩЕГО АНАЛИЗАТОРА ЯЗЫКА ГРАММАТИКИ ПРОСТОГО ПРЕДШЕСТВОВАНИЯ”

1. Цель работы

1.1. Изучить синтаксические анализаторы, базирующиеся на свойствах простого предшествования.

1.2. Освоить описание синтаксиса языков программирования грамматиками предшествования.

1.3. Получить исследовательские навыки построения отношений предшествования.

1.4. Освоить правила преобразования произвольной грамматики в грамматику предшествования

1.5. Исследовать временные показатели программы, реализующие алгоритм разбора предложений языка грамматики простого предшествования.

2. Теоретические сведения

Термин “восходящий разбор” характеризует направление построения синтаксического дерева (дерева разбора), предполагая движение от сентенциальной формы (оператора) к вершине, которую представляет аксиома грамматики, предложения которой разбираются [2, 4 - 9, 13, 14, 16]. Простой восходящий разбор инвариантен, то есть обладает универсальностью по отношению к свойствам формальной грамматики, за исключением свойства однозначности. Так как при обработке предложения алгоритмического языка априори отсутствуют сведения о том, какие правила грамматики и в каком порядке использованы для его получения, то в ходе применения алгоритма не избежать шtroкого применения метода проб и ошибок. Выбор простой фразы в качестве возможной основы разбора при неудаче обуславливает восстановление (полностью либо частично) исходного текста. Указанные действия в алгоритме соответствуют операциям подстановки и отхода, действие которых наглядно проиллюстрировано [5, с. 113]. Одним из удачных решений является алгоритм [5, с. 109 - 118]. Так же, как и все синтаксические анализаторы, он использует информацию, подготовленную сканером.

Отношение простого предшествования между рядом стоящими символами алфавита (объединённого словаря) R и S позволяет определить аргумент продукции при поиске основы синтаксического разбора, так как указывает на

одну из четырёх ситуаций: а) R и S стоят рядом в каком либо правиле; б) R – конец правила; в) S - начало правила; г) R и S несовместимы в синтаксически правильной программе [2, 5 – 9, 14].

Однозначные отношения существуют для класса грамматик простого предшествования, прилагательное “простой” нередко опускается. Если грамматика принадлежит к такому классу, то возможно построить эффективный распознаватель её языка, который, по отношению к обычному распознавателю, будет обладать повышенными скоростными характеристиками. Иногда представляется возможным построить функции предшествования (когда последние существуют) [7, параграф 5.4.]. В этом случае удаётся минимизировать размеры матрицы, которой представлены отношения предшествования, с размера $n \times n$ до $2 \times n$. Идеи, положенные в основание алгоритма разбора (рисунок 4.1), просты.

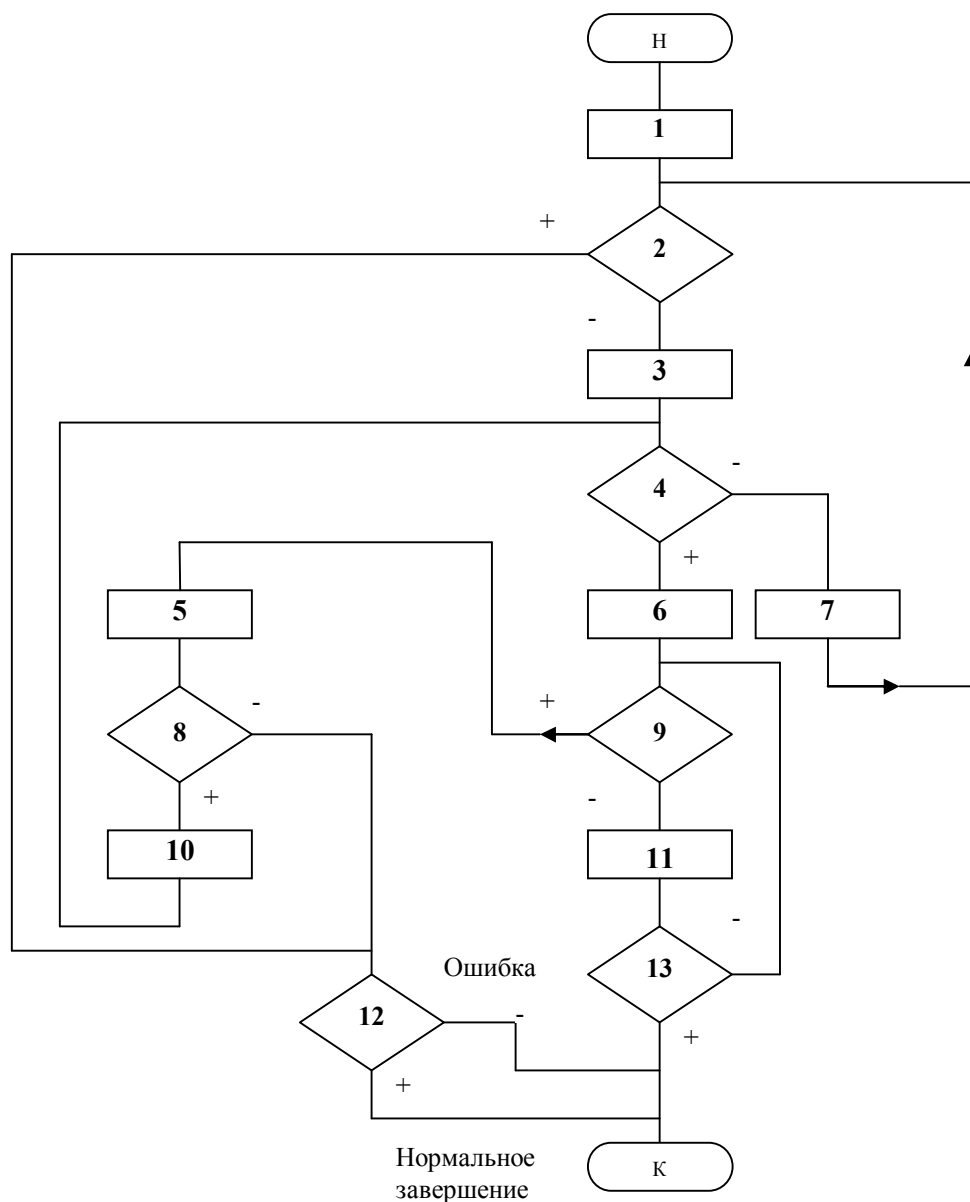


Рисунок 4.1 – Алгоритм восходящего разбора языка грамматики простого предшествования

1. Символы последовательно просматриваются слева направо до выполнения отношения $\bullet>$. Его выполнение означает, что предпоследний символ является концевым в каком-то правиле формальной грамматики.

2. Правая часть правила определяется отношением эквивалентности \equiv между символами, входящими в него, включая и предпоследний символ. Слева она ограничивается выполнением отношения $<\bullet$ (Рисунок 4.2).

3. Выполняется подстановка, после чего часть строки сжимается, и проверяется отношение предшествования между текущим анализируемым и подставленным символами.

4. Процесс продолжается итерационно и завершается подстановкой аксиомы, в случае удачи, либо появлением пары символов, отношение между которыми не определено.

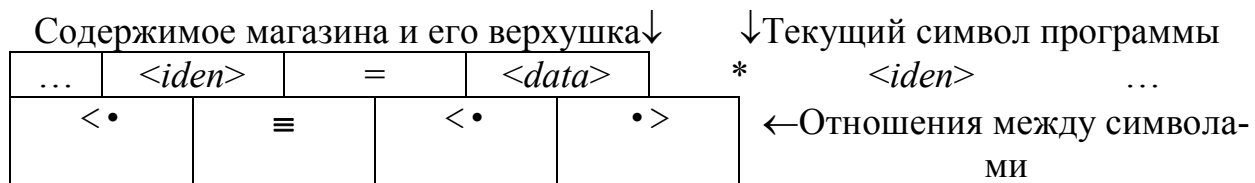


Рисунок 4.2 – Возможное состояние в ходе разбора

Функциональное содержание блоков рисунку 4.1 суть следующее.

1. $i \leftarrow 1$; МАГ[i] $\leftarrow \#$ (выталкиватель); $R \leftarrow \#$.
2. Конец разбираемой конструкции?
3. Чтение очередного символа программы S .
4. $R \bullet > S$?
5. Поиск ПЧП вида МАГ[j] ... МАГ[i]
6. $j = i$;
7. $i++$; МАГ[i] $\leftarrow S$; $R \leftarrow S$
8. Найдено соответствующее правило?
9. МАГ[$j-1$] $< \bullet$ МАГ[j]?
10. $i \leftarrow j$; МАГ[i] \leftarrow ЛЧП (подстановка); $R \leftarrow$ ЛЧП.
11. $j--$;
12. На вершине магазина аксиома или символ $\#$?
13. $j = 1$?

В пояснениях употреблены сокращения: МАГ – магазин (стек); ЛЧП – левая часть правила; ПЧП – правая часть правила.

3. Ход работы

Выполнение работы занимает два аудиторных занятия (4 академических часа) и предполагает интенсивную домашнюю подготовку.

3.1. Домашняя подготовка. Для позаимствованной из предыдущей работы № 3 или специально разработанной формальной грамматики, построить отношения предшествования. Выполнение необходимых преобразований проиллюстрировано [10, с. 27 – 36; 11, с. 42 – 48]. Если отношение предшествования построить не удалось, необходимо выполнить стратификацию правил грамматики [7, с. 141 – 144]. Стратификация выполняется при неоднозначности отношений и наличии явной левой или правой рекурсии, например

Исходные правила	Преобразованные правила
$P \rightarrow \dots SU \dots$	$P \rightarrow \dots SW \dots$
$U \rightarrow U \dots$	$W \rightarrow U$
$S \equiv U, S < \bullet U$	$U \rightarrow U \dots$
	$S \equiv W, S < \bullet U$

3.2. Занятие 1 (2 часа). Предъявить грамматику и ход построения отношений предшествования преподавателю, обсудить грамматику с преподавателем, ввести программу в компьютер, добиться успешной компиляции. В ходе отладки синтаксиса программы вести учёт синтаксических ошибок, выявляемых в ходе каждой компиляции.

3.3. Домашняя подготовка. Выполнить построение функций предшествования [7, с. 137 – 141; 10, с. 34 – 35; 11, с. 46 – 48]. Если функции удалось построить, внести необходимые изменения в программу. Определить содержание и место диагностических сообщений. Разработать тестовые примеры. Позаимствовать из предыдущей работы средства хронометража. Подготовить отчёт.

3.4. Занятие 2 (2 часа). Предъявить тесты преподавателю. Осуществить отладку логики восходящего анализатора. Зафиксировать время, затрачиваемое анализатором на обработку тестов с разной синтаксической структурой, сравнить временные характеристики с характеристиками рекурсивного нисходящего распознавателя. Продемонстрировать выполнение преподавателю. Окончательно оформить отчёт и защитить результаты выполнения лабораторной работы.

4. Содержание отчёта

4.1. Вариант задания.

4.2. Грамматика, порождающая фрагменты программы, соответствующие заданному варианту. Выкладки, связанные с построением отношений и функций предшествования, матрицы и функции предшествования.

4.3. Текст программы восходящего распознавателя, основанного на свойствах простого предшествования.

4.4. Тестовые примеры и тексты диагностических сообщений, результаты тестирования.

4.5. Статистика синтаксической и логической отладки программ, полученная по аналогии предыдущих работ

4.5. Осмысленные выводы, в которых должно быть отражено: среднее время, затрачиваемое на синтаксический разбор, и его сопоставление с временными оценками других программ синтаксического разбора; анализ динамики выявленных ошибок; размер исходного модуля, размер исполнимого модуля.

5. Вопросы для самоконтроля и самоподготовки

5.1. В чём заключаются преимущества и недостатки алгоритма простого восходящего разбора?

5.2. Какова сущность понятий подстановки и отхода?

5.3. Каким образом осуществляется поиск основы в ходе анализа алгоритмом простого восходящего разбора?

5.5. Возможен ли точный прогноз времени выполнения простого восходящего анализа?

5.6. Что представляют собой отношения простого предшествования?

5.7. Поясните свойства грамматики простого предшествования.

5.8. Способы и алгоритмы, применяемые при построении отношений.

5.9. Транзитивное замыкание отношений: сущность и построение. Алгоритм S. Warshall.

5.10. Изложите идеи, положенные в основу функционирования алгоритма разбора на основании свойств грамматики простого предшествования.

ЗАКЛЮЧЕНИЕ

Предлагаемый Вашему вниманию лабораторно – вычислительный практикум охватывает весьма незначительную часть теоретической области, принадлежащей научной информатике и посвящённой вопросам построения трансляторов с языков программирования [2 – 9, 13]. Тем не менее, в ходе его выполнения закладывается фундамент, позволяющий ориентироваться в проблемных вопросах и продолжать дальнейшее самообразование, а приобретённые навыки, вне сомнения, позволят, руководствуясь [7, 8, 13, 18], писать собственные трансляторы.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Алкок Д. Язык Паскаль в иллюстрациях [Текст] /Д. Алкок. – М.: Мир, 1991. – 192 с.
2. Ахо А. Теория синтаксического анализа, перевода и компиляции. [Текст]: Т.1. Синтаксический анализ / А. Ахо, Дж. Ульман. - М.: Мир, 1978. - 619 с.
3. Ахо А. Теория синтаксического анализа, перевода и компиляции. [Текст]: Т.2. Компиляция / А. Ахо, Дж. Ульман. - М.: Мир, 1978. - 560 с.
4. Ахо А. Компиляторы: принципы, технологии и инструменты [Текст] / А. Ахо, Р. Сети, Дж. Ульман. – М.: Издательский дом «Вильямс», 2002. – 512 с.
5. Вайнгартен Ф. Трансляция языков программирования [Текст] / Ф. Вайнгартен - М.: Мир, 1977. - 190 с.
6. Гордеев А.В. Системное программное обеспечение [Текст] / А.В. Гордеев, А.Ю. Молчанов. – СПб.: Питер, 2003. – 736 с.
7. Грис Д. Конструирование компиляторов для ЦВМ [Текст] / Д. Грис - М.: Мир, 1975. - 544 с.
8. Компаниец Р.И. Основы построения трансляторов [Текст] / Р.И. Компаниец, Маньков Е.В., Н.Е. Филатов. – СПб.: Корона-принт, 2000, - 256 с.
9. Льюис Ф. Теоретические основы проектирования компиляторов [Текст] / Ф. Льюис, Д. Розенкранц, Р. Стирнз. –М.: Мир, 1979. – 564 с.
10. Методические указания к выполнению лабораторных и контрольных работ по дисциплине “Теоретические основы построения компиляторов” для студентов всех форм обучения основного профиля 09.03.02 – “Информационные системы и технологии”. Основные алгоритмы [Текст] / Разраб. В.Ю. Карлусов. – Севастополь: Изд-во СевГУ, 2015. – 44 с.

11. Методические указания к выполнению самостоятельной работы по дисциплине “Теоретические основы построения компиляторов” для студентов всех форм обучения основного профиля 09.03.02 – “Информационные системы и технологии” [Текст] / Разраб. В.Ю. Карлусов. – Севастополь: Изд-во СевГУ, 2015. – 56 с.
12. Методические указания к выполнению контрольных работ по дисциплине “Теоретические основы построения компиляторов” для студентов заочной формы обучения основного профиля 09.03.02 – “Информационные системы и технологии” [Текст] / Разраб. В.Ю. Карлусов, С.А. Кузнецов. - Севастополь: Изд-во СевГУ, 2015. – 36 с..
13. Молчанов А.Ю. Системное программное обеспечение. [Текст]: учебник для вузов / А.Ю. Молчанов. – СПб.: Питер, 2006. – 396 с.
14. Молчанов А.Ю. Системное программное обеспечение. [Текст]: лабораторный практикум / А.Ю. Молчанов. – СПб.: Питер, 2005. – 284 с.
15. Оллонгрэн А. Определение языков программирования интерпретирующими автоматами [Текст] / А. Оллонгрэн - М.: Мир, 1972. - 288 с.
16. Рейуорд-Смит В.Дж. Теория формальных языков. Вводный курс / В.Дж. Рейуорд-Смит- М.: Мир, 1986. - 128 с.
17. Тамре Л. Введение в тестирование программного обеспечения [Текст] / Л. Тамре - М.: Издательский дом “Вильямс”, 2003. - 368 с.
18. Хантер Р. Основные концепции компиляторов [Текст] / Р. Хантер. - М.: Издательский дом “Вильямс”, 2002. - 256 с.
- 19.. Хопкрофт Дж. Введение в теорию автоматов, языков и вычислений [Текст] / Дж. Хопкрофт, Р. Мотвани, Дж. Ульман. – М.: Издательский дом “Вильямс”, 2002. – 528 с.

ПРИЛОЖЕНИЕ А

(обязательное)

Варианты, по усмотрению преподавателя, могут быть выданы по номерам зачётной книжки с использованием таблицы А.1.

Таблица А.1 – Список вариантов задания к лабораторной работе № 1.

1-я цифра	2-я цифра									
	1	2	3	4	5	6	7	8	9	0
1	A∨B∨C	D∨EF	GH∨I	J∨K∨L	KL∨A	E∨A∨L	B∨KC	JD∨I	F∨H∨G	G∨EA
2	B∨CD	E∨F∨G	HI∨J	H∨EB	I∨A∨L	LB∨K	C∨J∨D	I∨FH	GH∨K	B∨E∨I
3	J∨FC	DG∨K	I∨A∨J	F∨GH	BD∨E	L∨CF	JA∨D	F∨G∨E	L∨FG	EH∨D
4	A∨D∨G	B∨IK	CL∨H	F∨J∨E	G∨BI	I∨C∨J	B∨KA	AL∨F	G∨E∨H	D∨IC
5	KC∨L	H∨FJ	E∨A∨D	L∨HF	JE∨A	JB∨K	J∨E∨K	A∨D∨F	H∨BI	LC∨G
6	D∨G∨B	I∨KC	AL∨J	F∨C∨D	G∨KH	G∨J∨E	K∨AD	FH∨B	I∨L∨C	C∨GJ
7	EB∨I	C∨D∨E	D∨IF	IA∨L	J∨F∨C	BA∨D	I∨C∨J	E∨FK	GL∨H	H∨B∨A
8	D∨GK	AL∨B	K∨C∨J	H∨GE	HK∨B	D∨IC	JE∨F	K∨G∨L	E∨KA	DF∨H
9	E∨I∨L	C∨FJ	AD∨G	D∨G∨H	K∨BE	B∨I∨L	L∨HB	AD∨I	C∨J∨E	F∨KG
0	IL∨C	F∨J∨A	H∨DI	CJ∨B	K∨A∨L	BE∨D	G∨L∨H	C∨IL	JA∨H	I∨C∨A

В таблице А.1 приняты следующие обозначения:

A	–	$ab\{b\vee a\}$	G	–	$b\{a\vee ac\}$
B	–	$\{ab\}c\{c\}$	H	–	$c\{bb\vee ba\}$
C	–	$ca\{b\}$	I	–	$b\{b\}a$
D	–	$b\{c\vee a\}$	J	–	$a\{a\vee ab\}$
E	–	$\{a\vee c\}b$	K	–	$b\{ab\}c\{a\}$
F	–	$c\{a\vee b\}$	L	–	$b\{ba\vee bb\vee bc\}$.

ПРИЛОЖЕНИЕ Б

(обязательное)

Варианты, по усмотрению преподавателя, могут быть выданы по номерам зачётной книжки с использованием таблицы Б.1. Содержимое первой цифры варианта (описание синтаксиса идентификатора учебной программы) раскрывается в таблице Б.2, второй (описание синтаксиса констант) – в Б.3, третья (служебные слова, разделители и фрагменты программ) – в Б.4.

Таблица Б.1 – Список вариантов общего контрольного задания к лабораторным работам № 2 – 7 по теории синтаксического анализа.

1-я цифра	2-я цифра									
	1	2	3	4	5	6	7	8	9	0
1	1.1.15	14.8.2	3.8.8	13.7.4	15.6.5	16.4.3	17.3.6	1.2.7	2.1.8	3.8.9
2	12.2.10	2.7.11	13.3.12	4.2.13	5.7.14	6.6.1	9.5.1	10.4.9	13.3.2	4.2.4
3	8.3.5	11.6.6	3.4.14	9.1.7	12.6.8	5.7.9	11.4.9	12.3.10	14.2.11	5.1.12
4	7.4.14	10.5.13	6.5.15	4.8.1	14.8.2	11.7.4	15.6.5	16.5.6	17.4.3	6.3.7
5	17.8.8	1.7.9	2.6.10	3.7.1	5.2.2	15.1.4	8.4.14	1.2.5	2.3.6	7.4.10
6	16.7.7	11.4.14	15.5.8	14.6.9	13.5.10	6.7.11	16.6.12	10.8.13	3.1.14	8.2.1
7	7.6.2	8.3.4	9.6.5	10.5.6	11.3.7	12.4.10	7.5.8	17.6.9	11.7.10	4.8.11
8	6.5.12	5.2.13	4.7.14	3.4.1	2.1.2	1.2.4	17.3.5	8.4.3	1.5.6	12.6.7
9	9.4.10	10.1.8	11.8.9	12.3.10	13.7.11	14.8.12	15.1.13	16.2.14	9.3.1	2.4.9
0	8.3.2	7.2.4	7.1.5	6.2.6	5.5.7	4.6.8	3.7.9	2.8.10	1.6.11	10.8.12

Таблица Б.2- Типы идентификаторов

№ вариан-та	Описание типа
(1)	(2)
1	Содержит чередующиеся пары букв и цифр, заканчивается последовательностью символов "1"
2	Содержит чередующиеся буквы и цифры, заканчивается последовательностью символов "0"
3	Если встречается символ "a", то за ним всегда следует символ "c"
4	За встреченным символом "b" следует последовательность нулей и единиц
5	Оканчивается символом "m", которому не предшествует буква
6	Оканчивается символом "s", которому не предшествует цифра
7	Включает последовательность "abs"
8	Содержит последовательность символов "1" и оканчивается цифрой
9	Содержит последовательность символов "0" и оканчивается буквой
10	Оканчивается последовательностью цифр и включает символ "a"
11	Оканчивается последовательностью букв и включает последовательность "11"
12	Оканчивается символом "0" и включает последовательность "0100"
13	Следом за начальной буквой идентификатора следует "111"
14	Цифры, встречающиеся в идентификаторе, если они присутствуют в его записи, упорядочены по возрастанию
15	Цифры, включенные в состав идентификатора, сгруппированы в одну последовательность
16	В идентификаторе не встречается более трех букв, идущих подряд
17	В идентификаторе не встречается более трёх цифр, идущих подряд
18	Никакие две одинаковые буквы или цифры не стоят рядом

Таблица Б.3 - Типы констант

№ варианта	Тип	Пояснения	Формат
1	F	С фиксированной точкой	$\pm p \dots p.q \dots q$
2	E	С плавающей точкой	$\pm p \dots p.q \dots q E \pm ss$
3	B	Двоичная	$\pm 10 \dots 001 B$
4	C	Символьная	'a...Z0...9'
5	X	Шестнадцатеричная	$\pm 0x12 \dots 9a \dots f$
6	O	Восьмеричная	$\pm 0o012 \dots 7$
7	D	Удвоенная, с плавающей точкой	$\pm p \dots p.q \dots q D \pm sss$
8	L	Удвоенная, целая	$\pm 1001 \dots 010011 L$

Примечание. Для типа F задаются: максимальный размер целой части Р и максимальный размер дробной части q. Для типов В, Х, О, L - задается число цифр в записи. Тип Е имеет максимально 6 цифр в мантиссе и две в показателе степени s. Тип D имеет максимально 16 цифр в мантиссе и 3 в показателе степени.

Таблица Б.4 - Фрагмент программ для анализа

№ варианта	Служебные слова	Разделители		Фрагмент программы для анализа
		одно-литерные	двулитерные	
(1)	(2)	(3)	(4)	(5)
1	BEGIN END	; = * /		BEGIN; x=4; B=4.5*x/3; END;
2	START STOP	* + , -	:=	START, x:=-0.5, y:=3*x+1, STOP
3	PUSK OFF	& ! =	\	PUSK \\ x='ABC' \\ y=x&'A'!B \\ OFF

Таблица Б.4 – продолжение.

(1)	(2)	(3)	(4)	(5)
4	PRINT SCAN	, ; + - ()		PRINT (A+B,'C'); SCAN (F)
5	IF THEN	() & + -	== :=	IF (A==B) & (C==0x0f) THEN K:=K+1;
6	DO WHILE END	= - + , ()	<= >=	WHILE (K<=0o171) DO K=K+1, A=A-1, END,
7	FOR TO	() = ; * /		FOR (I=1) TO 4 (K=K*5; C=C/2E+01;);
8	CASE SWITCH	{ } () + - ; = :		K=5; C=5+x; SWITCH(C) { case 1: K=K+1; case 2: K=K+2; }
9	CHAR INTEGER	, ; = ()	!! << >>	CHAR (A,B); INTEGER (C); A='SANTAS DUMON'; B=A!!A;
10	PROC END	= ,	&& !! << >>	PROC, A=0xA1011, B=0xae13, C=A&&B, END,

Таблица Б.4 – продолжение.

(1)	(2)	(3)	(4)	(5)
11	READ WRITE	() * / - + , \$		WRITE (A*B, C+23) \$ READ (D) \$
12	IF THEN ELSE	< > = + - () & :		IF (A>B & C<0) THEN (A=A+B+7) ELSE (A=A-B; C=D+B);
13	DO UNTIL	() = + - * / < > #	<>	DO x=3+y # z=2/x+1 C=K*C+1 UNTIL(z<>0.01)#
14	FUNC FINAL	: \ ! { } ()	:= && !!	A: FUNC (x)\ { X:=(x&&0xfe)\ A:=!(x!!0x13)\ } FINAL
15	WAIT SIGNAL	() > = + %	>= <=	WAIT(S) K=K+5.3 % C=K+C+1 % SIGNAL(S<=0xf)

Таблица Б.4 – продолжение.

(1)	(2)	(3)	(4)	(5)
16	FOR TO NEXT	() = ; + - * /		FOR I=0 TO 8 K=K+5*(I-3); C=C/2E+01; NEXT I
17	FORALL DO END FORALL	- + , * / ()	:=	FORALL X DO X:=A+0o17*(B+2), C=K/C+1, END FORALL
18	SUB ENDSUB	: () = ; + - * /	:=	MODUL1: SUB(A, B, C); F:=A+B*7; C:=(F+3.14)*A; ENDSUB;

Примечание – Вид разбираемой инструкции, приводимой в графе (5) таблицы Б.4, весьма условен в части идентификаторов и констант. Они должны находиться в полном соответствии с вариантом, приводимым в таблицах Б.2 и Б.3.

Приложение В (обязательное)

Рассмотрим, в общих чертах, пример построения программной реализации восходящего распознавателя грамматики простого предшествования. Построение грамматики, порождающей конструкции языков высокого уровня вида

STA
$xE := yE + 1.1$
$y123E := xE * yE - 2.257$
STO

с четырьмя действиями арифметики над операндами в левой части оператора присваивания подробно изложено в [12]. В развёрнутой нотации Бекуса-Наура, формальная грамматика выглядит следующим образом

$\langle \text{фрагмент} \rangle ::= \text{STA } \langle \text{тело программы} \rangle$
$\langle \text{тело программы} \rangle ::= \langle \text{оператор} \rangle \langle \text{тело программы} \rangle$
$\langle \text{тело программы} \rangle ::= \text{STO}$
$\langle \text{оператор} \rangle ::= \langle \text{iden} \rangle := \langle \text{выражение} \rangle$
$\langle \text{выражение} \rangle ::= \langle \text{операнд} \rangle \langle \text{знак} \rangle \langle \text{выражение} \rangle$
$\langle \text{выражение} \rangle ::= \langle \text{операнд} \rangle$
$\langle \text{операнд} \rangle ::= \langle \text{iden} \rangle \mid \langle \text{data} \rangle$
$\langle \text{знак} \rangle ::= + \mid - \mid * \mid /$
$\langle \text{знак} \rangle ::= -$
$\langle \text{знак} \rangle ::= *$
$\langle \text{знак} \rangle ::= /$

1. По известным алгоритмам необходимо построить отношения простого предшествования. Для данной формальной грамматики существуют отношения предшествования, приведённые в [12, таблица 2.4].

Для программной реализации необходимо добавить вспомогательный символ “#”, являющийся ограничителем анализируемой строки. Физически этот символ слева (в начале строки) означает выталкиватель стека, а справа (в конце строки) является маркером (признаком) конца строки.

Между символом “#” и всеми символами формальной грамматики устанавливаются отношения “<•”, чтобы обеспечивать запись кодов символов поверх выталкивателя стека. В противном случае, между всеми символами формальной грамматики и символом “#” устанавливаются отношения “•>”, чтобы обеспечивать свёртку фрагмента программного кода, находящегося в стеке. Соседнее положение пар символов “#” и “#” или $\langle \text{фрагмент} \rangle$ и “#” означает успешное завершение синтаксического анализа.

Символ “#”, с соответствующими ему столбцом и строкой, помещается в таблицу отношений предшествования.

2. По таблице отношений предшествования следует построить диагностическую таблицу, управляющую процессом синтаксического анализа и справочную таблицу, обеспечивающую преобразование внутренних кодов грамматики в номера столбцов и строк диагностической таблицы.

Будем использовать систему кодирования, принятую в [12] на этапе разработки сканера, дополненную кодами нетерминальных символов и метасимволов, используемых в нотации НФБ.

Таблица В.1 – Кодирование лексем, нетерминалов и вспомогательных символов, используемых в учебном примере

Символ	Код	Нумерация строк и столбцов таблиц отношения предшествования и диагностики
(1)	(2)	(3)
<фрагмент>	701	0
<тело программы>	702	1
<оператор>	703	2
<выражение>	704	3
<операнд>	705	4
<знак>	706	5
STA	100	6
STO	200	7
<iden>	300	8
<data>	400	9
:=	600	10
+	501	11
–	502	12
*	503	13
/	504	14
#	999	15
::=	800	
конец правила	900	

Для обращения к диагностической таблице нам понадобится конвертирующая функция, которая по коду лексемы или нетерминала возвращает номер строки или столбца. Возможная программная реализация приведена ниже.

```
int NumberColRow(int *cod)
{
    switch (cod) {
        case 100: return 6; break; //STA
        case 200: return 7; break; //STO
        case 300: return 8; break; //iden
```

```

        case 400: return 9; break;      //data
        case 501: return 11; break;    // +
        case 502: return 12; break;    // -
        case 503: return 13; break;    // *
        case 504: return 14; break;    // =
        case 600: return 10; break;    // :=
        case 701: return 0; break;     // <фрагмент>
        case 702: return 1; break;     // <тело программы>
        case 703: return 2; break;     // <оператор>
        case 704: return 3; break;     // <выражение>
        case 705: return 4; break;     // <операнд>
        case 706: return 5; break;     // <знак>
        case 999: return 15; break;    // #
        default;;
    }
}

```

При построении диагностической таблицы воспользуемся данными [12, таблица 2.5]. При этом была использована следующая система обозначений. Пусть, отношения предшествования кодируются цифрами: 1 – \prec , 2 – \equiv , 3 – \succ . Для обозначения сообщений об ошибках используем: 4 – “Неверная структура программы”; 5 – “Ошибка компоновки тела модуля”; 6 – “Ошибка составления выражения”; 7 – “Недопустимая комбинация знаков”.

Фрагмент программного кода, реализующий диагностическую таблицу, показан ниже.

```

int SR[VOC_POWER][VOC_POWER]={
//////// 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
/*0*/ {4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 99},
/*1*/ {4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 1},
/*2*/ {4, 2, 1, 6, 6, 6, 4, 1, 1, 6, 6, 6, 6, 6, 6, 1},
/*3*/ {4, 3, 3, 6, 6, 6, 6, 3, 3, 6, 6, 6, 6, 6, 6, 1},
/*4*/ {4, 3, 3, 6, 6, 2, 6, 3, 3, 6, 6, 1, 1, 1, 1, 1},
/*5*/ {4, 6, 6, 2, 1, 6, 6, 6, 1, 1, 6, 6, 6, 6, 6, 1},
/*6*/ {4, 2, 1, 5, 5, 5, 5, 1, 1, 5, 5, 5, 5, 5, 5, 1},
/*7*/ {4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 1},
/*8*/ {4, 6, 6, 6, 6, 1, 4, 4, 6, 6, 2, 3, 3, 3, 3, 1},
/*9*/ {4, 6, 6, 6, 6, 1, 4, 4, 6, 6, 6, 3, 3, 3, 3, 1},
/*11*/ {4, 7, 7, 3, 3, 7, 4, 4, 3, 3, 7, 7, 7, 7, 7, 1},
/*10*/ {4, 7, 7, 2, 1, 7, 4, 4, 1, 1, 7, 7, 7, 7, 7, 1},
/*12*/ {4, 7, 7, 3, 3, 7, 4, 4, 3, 3, 7, 7, 7, 7, 7, 1},
/*13*/ {4, 7, 7, 3, 3, 7, 4, 4, 3, 3, 7, 7, 7, 7, 7, 1},
/*14*/ {4, 7, 7, 3, 3, 7, 4, 4, 3, 3, 7, 7, 7, 7, 7, 1},
/*15*/ {3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
99}};

```


Нумерация строк и столбцов показана при описании диагностической таблицы в виде комментариев.

3. Для выполнения операций свёртки нам необходимо представить правила формальной грамматики в удобной для поиска левых частей правил форме.

Во внутренних промежуточных кодах данная грамматика может быть представлена (как вариант), согласно таблице В.1, нижеследующим образом.

701	800	100	702	900	
702	800	200	900		
702	800	300	600	704	900
703	800	705	706	704	900
704	800	705	900		
705	800	300	900		
705	800	400	900		
706	800	501	900		
706	800	502	900		
706	800	503	900		
706	800	504	900		

Для использования в алгоритме эти данные оформлены как одномерный массив, в котором коды 800 и 900 играют роль ограничителей “поискового образа” – левой части правила, а код правой части правила находится непосредственно перед кодом 800.

```
int *G[NT_POWER]={
    701, 800, 100, 702, 900,
    702, 800, 200, 900,
    702, 800, 300, 600, 704, 900,
    703, 800, 705, 706, 704, 900,
    704, 800, 705, 900,
    705, 800, 300, 900,
    705, 800, 400, 900,
    706, 800, 501, 900,
    706, 800, 502, 900,
    706, 800, 503, 900,
    706, 800, 504, 900};
```

Учитывая наличие в языке C++ оператора препроцессорной директивы компилятору # define, содержимое массива продукций G и функции NumberColRow может быть оформлено более осмысленно с “человеческой” точки зрения. В частности:

```
# define FRAGMENT 701
...
# define STA 100
# define STO 200
```

```

...     ...
switch (cod) {
    case 100: return 6; break;    //STA
...     ...

```

4. Построение кода анализатора программного кода, реализующего известные алгоритмы, например [4, 6 – 8], основанные на использовании стека.

```

/// Начальные установки
magazine[0] = 999; // Код знака "#", выталкиватель стека
mp = 0;           // Указатель заполнения магазина (стека)
. . .
// Такт работы распознавателя (управляющее устройство)
NoStrk = NumberColRow(magazin[mp]); // Адресация таблицы
NoStlb = NumberColRow(code);
switch (SR[NoStrk][NoStlb]){
    case 1: // Операция переноса для случая <•
        mp++;
        magazine[mp] = 1; // Ограничитель начала фразы
        mp++;
        magazine[mp] = code;
        nf_pointer = mp; // Запоминаем голову последней
                        // простой фразы
        break;
    case 2: // Операция переноса для случая эквивалентности
        mp++;
        magazine[mp]=code;
        break;
    case 3: // Операция свёртки, основа в магазине
        NonTerm = FindRule(GR, magazine, mp, nf_pointer);
        if (SR[NumberColRow(magazin[nf_pointer - 1])][
            NumberColRow(NonTerm)] == 1) // <•
        {
            magazin[nf_pointer] = NonTerm;
            mp = nf_pointer;
        }
        else // отношение эквивалентности
        {
            nf_pointer --;
            magazin[nf_pointer] = NonTerm;
            mp = nf_pointer;
            nf_pointer = FindPointer(magazine, mp);
        }
        break;
    case 4:

```

```

        printf ("Invalid program structure.");
        getch();
        return 0;
        break;
case 5:
    printf ("Module composition mistake.");
    getch();
    return 0;
    break;
case 6:
    printf ("Invalid statement.");
    getch();
    return 0;
    break;
case 7:
    printf ("Illegal sign combination.");
    getch();
    return 0;
    break;
case 99:
    printf ("Syntax checked. Ok.");
    getch();
    return 0;
    break;
default:
    printf ("Unknown Error in syntax.");
    getch();
    return 0;
}

```

Работу устройства управления обеспечивает, помимо NumberColRow, ряд функций.

Функция для поиска правила, правая часть которого совпадает с основой `int FindndRule(int *Grammar, int *stack, int stack_pointer, int head_pointer)` и возвращающая код нетерминального символа, который описывается данной продукцией.

Активизируется всякий раз, когда отношение предшествования указывает на наличие основы в магазине. Реализует любой понятный Вам алгоритм поиска заданной последовательности чисел в массиве.

При организации массива правил формальной грамматики по принципам, изложенным выше, код искомого нетерминала будет находиться в этом массиве через элемент впереди от начала найденной искомой последовательности.

Функция поиска местоположения головы основы в стеке `int Find-Pointer(int *stack, int stack_pointer)`. Тело функции представляет цикл с уменьшением управляющей переменной от значения, равного значению указателя стека, выполняющегося до нахождения в стеке кода отношения $< \bullet$ (в нашем случае это 1).

Вызов её осуществляется, когда нетерминал, полученный в результате свёртки, оказывается с соседним элементом магазина в отношении эквивалентности.

Функция возвращает номер элемента стека, соответствующий коду отношения $< \bullet$, ближайший его вершине.