Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«Амурский государственный университет»

Кафедра информационных и управляющих систем

УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС ДИСЦИПЛИНЫ

«СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ»

Основной образовательной программы по специальности

230102.65 – Автоматизированные системы обработки информации и управления

Благовещенск 2012

УМКД разработан канд. техн. наук, доцентом Т.А. Галаган

Рассмотрен и рекомендован на заседании кафедры

Протокол заседания кафедры от «__» ______________201_ г. № ______

Зав. кафедрой _____________________ / А.В. Бушманов /
(подпись) (И.О. Фамилия)

УТВЕРЖДЕН:

Протокол заседания УМСС 230102.65 – Автоматизированные системы обработки информации и управления

от «____» ________________ 201__ г. № ______

Председатель УМСС ______________ /________________________________ /
(подпись) (И.О. Фамилия)

1 Рабочая программа

1. ЦЕЛИ И ЗАДАЧИ ДИСЦИПЛИНЫ

Традиционная архитектура компьютера остается неизменной, неизменны и базовые принципы построения программного обеспечения – трансляторы, компиляторы и интерпретаторы.

Цель дисциплины – обучение студентов теоретическим основами и принципам, лежащим в основе современных средств разработки программного обеспечения.

Задачи дисциплины:

изучение понятий транслятор, интерпретатор, компилятор, их структуры;

изучение алгоритмов, используемых в реализации различных фаз компиляции.

По завершению курса «Системное программное обеспечение» студент должен:

владеть такими понятиями, как распознаватели, формальные языки и грамматики, конечные автоматы, автоматы с магазинной памятью;

знать назначение и функции компиляторов, трансляторов, интерпретаторов, современное состояние методов, используемых при их разработке;

иметь устойчивые практические навыки классификации языков и грамматик, программной реализации алгоритмов работы конечных автоматов и их минимизации.

Материал дисциплины тесно связан с материалом дисциплин «Информатика», «Операционные системы», «Алгоритмические языки и программирование».

2. МЕСТО ДИСЦИПЛИНЫ В СТРУКТУРЕ ООП ВПО

Программа курса «Системное программное обеспечение» составлена в соответствие с требованиями государственного образовательного стандарта специализации – Интегрированные автоматизированные системы, блок специальных дисциплин СД.11.

Пользовательский интерфейс операционной среды; управление задачами; управление памятью; управление вводом-выводом; управление файлами; пример современной операционной системы; ассемблеры; мобильность программного обеспечения; макроязыки; формальные системы и языки программирования; грамматики, компиляторы; интерактивные системы; средства трассировки и отладки программ.

3. СТРУКТУРА И СОДЕРЖАНИЕ ДИСЦИПЛИНЫ (МОДУЛЯ)

Общая трудоемкость дисциплины составляет 104 часа.

№ п/п	Раздел дисциплины	Семестр	Неделя семестра	Виды учебной работы, включая самостоятельную работу студентов и трудоемкость в часах			Формы текущего контроля успеваемости (по неделям семестра) Форма промежуточной аттестации (по семестрам)
				лекц.	лаб.	сам.	
1	Интерфейсы операционной среды	7	1	2	6	6	Отчет по лаб. раб.
2	Трансляторы и компиляторы	7	2 – 4	6	6	6	Отчет по лаб. раб.
3	Формальные языки и грамматики	7	5 - 8	8	10	10	Отчет по лаб. раб.
4	Средства трассировки и	7	9	2		4	

	отладки программ						
5	Управление задачами	7	10 - 11	4		4	
6	Управление памятью	7	12 - 15	4	2	4	Отчет по лаб. раб.
7	Управление вводом-выводом	7	15	4		4	Отчет по лаб. раб
8	Работа с современными ОС	7	13-15		6	6	Отчет по лаб. раб
	ИТОГО			30	30	44	

4. СОДЕРЖАНИЕ РАЗДЕЛОВ И ТЕМ ДИСЦИПЛИНЫ

4.1 Лекции

4.1.1. Интерфейсы операционной среды: текстовый и графический и речевой пользовательский.

4.1.2. Трансляторы и компиляторы: общая схема работы, особенности построения; организация таблиц идентификаторов: простейшие методы, метод бинарного дерева, хэш-адресация, комбинированные методы.

4.1.3. Формальные языки и грамматики: способы задания, классификация, регулярные и автоматные грамматики, конечные автоматы, автоматы с магазинной памятью

4.1.4. Средства трассировки и отладки программ. Мобильность программного обеспечения.

4.1.5. Управление задачами: планирование и диспетчеризация процессов и задач; стратегии планирования; дисциплины обслуживания.

4.1.6. Управление памятью: структуризация виртуального адресного пространства; общие принципы управления памятью.

4.1.7. Управление вводом-выводом: режимы управления вводом-выводом; основные системные таблицы ввода-вывода; синхронный и асинхронный ввод-вывод.

4.2 Лабораторные работы

4.2.1. Работа с файловой системой с использованием JavaScript и Windows Scripting Host (WHS)

4.2.2. Алгоритмы построения таблиц идентификаторов.

4.2.3. Реализация алгоритма работы конечного автомата.

4.2.4. Минимизация конечного автомата.

4.2.5. Автоматы с магазинной памятью.

4.2.6. Работа в Windows, Linux

5. САМОСТОЯТЕЛЬНАЯ РАБОТА

№ п/п	№ раздела (темы) дисциплины	Форма (вид) самостоятельной работы	Трудоемкость в часах
1	Интерфейсы операционной среды	Изучение учебной литературы Приобретение навыков работы в среде JavaScript и Windows Scripting Host	6
2	Трансляторы и компиляторы	Изучение учебной литературы Подготовка отчета по лабораторной работе	6
3	Формальные языки и грамматики	Изучение учебной литературы Подготовка отчета по лабораторной работе	10
4	Средства трассировки и	Изучение учебной литературы	4

	отладки программ		
5	Управление задачами	Изучение учебной литературы	4
6	Управление памятью	Изучение учебной литературы Подготовка отчетов по лабораторным работам	4
7	Управление вводом-выводом	Изучение учебной литературы	4
8	Работа с современными ОС	Подготовка отчетов по лабораторным работам Приобретение навыков работы в современных ОС	6
	Итого		**44**

6. ОБРАЗОВАТЕЛЬНЫЕ ТЕХНОЛОГИИ

К образовательным технологиям, используемым в преподавании данной дисциплины, относятся лекции и лабораторные работы.

В изложении лекционного материала наряду с традиционной лекцией используются такие неимитационные методы обучения, как:

проблемная лекция, начинающаяся с постановки проблемы, которую необходимо решить в ходе изложения материала,

лекция-визуализация, учащая студента преобразовывать устную и письменную информацию к визуальной форме в виде схем, рисунков, чертежей,

лекция с заранее запланированными ошибками, которые студенты должны обнаружить самостоятельно в конце лекции.

На лекциях используются информационные технологии – презентации.

Лабораторные работы проводятся в компьютерных классах и предназначены для решения прикладных задач с использованием современных инструментальных средств.

При проведении лабораторных работ используются неигровые имитационные методы обучения: контекстное обучение, направленное на решение профессиональных задач, работа в команде – совместная деятельность студентов в группе, направленная на решение общей задачи с разделением ответственности и полномочий.

7. ОЦЕНОЧНЫЕ СРЕДСТВА ДЛЯ ТЕКУЩЕГО КОНТРОЛЯ УСПЕВАЕМОСТИ, ПРОМЕЖУТОЧНОЙ АТТЕСТАЦИИ ПО ИТОГАМ ОСВОЕНИЯ ДИСЦИПЛИНЫ И УЧЕБНО-МЕТОДИЧЕСКОЕ ОБЕСПЕЧЕНИЕ САМОСТОЯТЕЛЬНОЙ РАБОТЫ СТУДЕНТОВ

7.1. Вопросы к экзамену

1. Графический интерфейс операционной среды
2. Текстовый интерфейс операционной среды
3. Цепочки символов. Операции над цепочками символов.
4. Формальное определение языка
5. Формальное определение грамматики. Форма Бэкуса-Наура
6. Общая схема работы распознавателя
7. Виды распознавателей
8. Четыре типа грамматик по Хомскому
9. Классификация языков
10. Определение транслятора. Однопроходные и многопроходные трансляторы

11. Этапы трансляции
12. Определение компилятора. Особенности построения и функционирования
13. Определение интерпретатора.
14. Организация таблиц идентификаторов. Простейшие способы построения
15. Построение таблиц идентификаторов по методу бинарного дерева
16. Принципы работы хэш-функций
17. Построение таблиц идентификаторов на основе хэш-функций
18. Построение таблиц идентификаторов по методу цепочек
19. Назначение лексического анализатора
20. Принципы построения лексических анализаторов
21. Конечные автоматы
22. Алгоритмы минимизации и преобразования конечных автоматов
23. Синтаксические анализаторы. Построение синтаксических анализаторов
24. Автоматы с магазинной памятью
25. Виды переменных
26. Виды и областей памяти
27. Статистическое и динамическое связывание
28. Стековая организация памяти.
29. Способы внутреннего представления программ
30. Принципы оптимизации кода
31. Планирование и диспетчеризация процессов и задач
32. Стратегии планирования процессов
33. Дисциплины обслуживания процессов
34. Структуризация виртуального адресного пространства
35. Функции ОС по управления памятью
36. Страничное распределение
37. Сегментное распределение
38. Сегментно-страничное распределение
39. Режимы управления вводом-выводом
40. Основные системные таблицы ввода-вывода
41. Синхронный и асинхронный ввод-вывод
42. Средства трассировки и отладки программ
43. Мобильность программного обеспечения.

8. УЧЕБНО-МЕТОДИЧЕСКОЕ И ИНФОРМАЦИОННОЕ ОБЕСПЕЧЕНИЕ ДИСЦИПЛИНЫ (МОДУЛЯ)

ОСНОВНАЯ ЛИТЕРАТУРА

1 Галаган, Т.А. Системное программное обеспечение: учеб. пособие / Т.А. Галаган – Благовещенск: изд-во Амур. гос. ун-та, 2009. – 80 с. Режим доступа file://10.4.1.254/DigitalLibrary/AmurSU_Edition/1961.pdf

ДОПОЛНИТЕЛЬНАЯ

1 Молчанов, А. Ю. Системное программное обеспечение. (Допущено МинОбр РФ) / Молчанов А.Ю – СПб: Питер, – 2003, 2006. – 396с.

2 Молчанов, А. Ю. Системное программное обеспечение. Лабораторный практикум / Молчанов А.Ю – СПб.: Питер, – 2005. – 284 с.

3 Павловская, Т.А. C/C++. Программирование на языке высокого уровня (Доп. Мин. образования РФ) – СПб.: Питер, 2009, 2010. – 461с.

4 Гордеев, А.В. Операционные системы. Учебник для вузов. (Допущено МинОбр РФ) –

СПб: Питер, – 2006, 2009. – 416 с.

5 Галаган, Т.А. Практикум по лингвистическим основам информатики. /Т.А. Галаган, Л.А. Соловцова. – Благовещенск: изд-во АмГУ, 2005. – 99с.

ИНТЕРНЕТ-РЕСУРСЫ

	Наименование ресурса	Характеристика
1	http://www.intuit.ru	ИНТУИТ - сайт, который предоставляет возможность дистанционного обучения по нескольким образовательным программам, касающимся, в основном, информационных технологий. Содержит несколько сотен открытых образовательных курсов.
2	http://ru.wikipedia.org	Википендия – свободная общедоступная мультиязычная универсальная интернет-энциклопедия. Поиск по статьям, написанным на русском языке. Избранные статьи, ссылки на тематические порталы и родственные проекты.

ПЕРИОДИЧЕСКИЕ ИЗДАНИЯ

Журналы «Информационные технологии и вычислительные системы», «Компьютер-Пресс», «Программные продукты и системы»

9.МАТЕРИАЛЬНО-ТЕХНИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ДИСЦИПЛИНЫ (МОДУЛЯ)

Мультимедийная лекционная аудитория (331)

В качестве программного обеспечения используются свободно распространяемые инструментальные средства.

10. РЕЙТИНГОВАЯ ОЦЕНКА ЗНАНИЙ СТУДЕНТОВ ПО ДИСЦИПЛИНЕ

12.1. Балльная структура оценки за первый семестр

Семестровый модуль дисциплины						
Учебные модули		Виды контроля	Сроки выполнения (недели)	Макс. кол-во баллов	Посещение занятий, активность	Макс. кол-во баллов за уч. модуль
1	Интерфейсы операционной среды	Отчет по лаб. раб.	1	3	3	6
2	Трансляторы и компиляторы	Отчет по лаб. раб.	2 – 4	4	2	6
3	Формальные языки и грамматики	Отчет по лаб. раб.	5 - 8	4	2	6
4	Средства трассировки и отладки программ		9		2	6
5	Управление задачами		10 - 11	5	3	8
6	Управление памятью	Отчет по лаб. раб.	12 - 15	5	2	7
7	Управление вводом-	.	15		1	1

	выводом					
8	Работа с современными ОС	Отчет по лаб. раб	13-15	4		
	Сдача экзамена					40
	Итого					100

12.2. Итоговая оценка

Сумма баллов, набранных в течение семестра (с возможностью проставления предварительной оценки за экзамен)	Общая сумма баллов (с учетом сдачи экзамена в период семестровой аттестации)	Итоговая оценка
56 – 60	91 – 100	отлично
51 – 55	75 – 90	хорошо
46 – 50	51 – 74	удовлетворительно
<46	< 51	неудовлетворительно

2 **Краткое изложение программного материала**

Лекция 1

Тема Интерфейсы операционной среды

Цель – организация целенаправленной познавательной деятельности студентов по овладению программным материалом.

Задача – изучение важных понятий интерфейса взаимодействия с пользователем.

Ключевые вопросы – отличительные особенности, виды интерфейсов.

План лекции

1. Основные термины
2. Виды интерфейса операционной среды
3. Стандарты интерфейса

Термин операционная среда означает необходимые интерфейсные программы пользователя для обращения к операционной системе с целью получить определённый сервис. Операционная оболочка - часть операционной среды, определяющая интерфейс пользователя, его реализацию (текстовый, графический и т.п.), командные и сервисные возможности пользователя по управлению прикладными программами и компьютером.

Операционная система (ОС) должна обеспечить удобный интерфейс не только для прикладных программ, но и для человека, работающего за компьютером. Современные ОС поддерживают развитые функции пользовательского интерфейса для интерактивной работы за терминалом двух типов: алфавитно-цифровыми и графическими.

При работе за алфавитно-цифровым терминалом пользователь имеет систему команд, мощность которой отражает функциональные возможности ОС, позволяет запускать и останавливать приложения, выполнять операции с файлами, получать информацию о состоянии ОС, администрировать систему. Команды могут вводиться в интерактивном режиме или считываться из командного файла. Ввод команды упрощен, если ОС поддерживает графический пользовательский интерфейс. GUI – Graphical User Interface – неотъемлемая часть многих современных ОС.

В Линукс-системах пользователи работают через интерфейс командной строки (CLI), графический интерфейс пользователя (GUI), или, в случае встраиваемых систем, через элементы управления соответствующих аппаратных средств. Настольные системы, как правило, имеют графический пользовательский интерфейс, в котором командная строка доступна через окно эмулятора терминала или в отдельной виртуальной консоли. Командная строка особенно хорошо подходит для автоматизации повторяющихся или отложенных задач, а также предоставляет очень простой механизм межпроцессного взаимодействия. Программа графического эмулятора терминала часто используются для доступа к командной строке с рабочего стола.

На настольных системах наибольшей популярностью пользуются пользовательские интерфейсы, основанные на таких средах рабочего стола как KDE Plasma Desktop, GNOME и Xfce, хотя также существует целый ряд других пользовательских интерфейсов. Самые популярные пользовательские интерфейсы основаны на X Window System предоставляют прозрачность сети и позволяют графическим приложениям, работающим на одном компьютере, отображаться на другом компьютере, на котором пользователь может взаимодействовать с ними.

Частным случаем попытки стандартизировать API является внутренний корпоративный стандарт компании Microsoft – WinAPI. Он включает реализации Win16, Win32s, Win32, WinCE.

Примером стандартизации API служит один из самых распространенных стандартов – POSIX – независимый от платформы системный интерфейс для компьютерного окружения – является стандартом IEEE.

Литература – основная: п.1 , дополнительная: п.3, 4 рабочей программы, интернет-ресурсы.

Лекция 2

Тема Трансляторы и компиляторы

Цель – организация целенаправленной познавательной деятельности студентов по овладению программным материалом.

Задача – изучение важных понятий, реализующих внутренние механизмы работы транслятора.

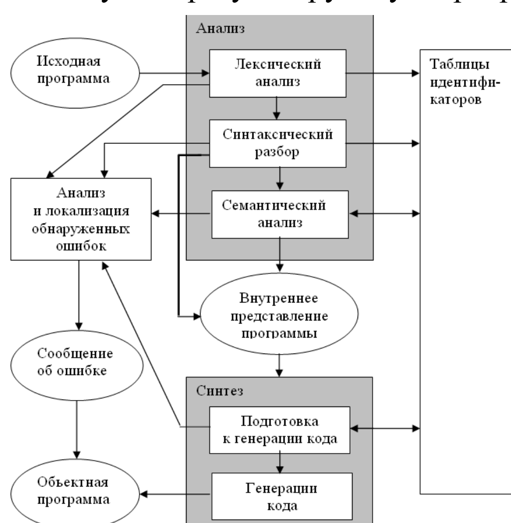
Ключевые вопросы – отличительные особенности транслятора, компилятора, интерпретатора.

План лекции

1. Определения «транслятор», «компилятор», «интерпретатор»
2. Общая схема работы транслятора

Транслятор является программой, переводящей исходную программу в эквивалентную ей программу на *результирующем (выходном) языке*.

Компилятор – транслятор, осуществляющий перевод исходной программы в эквивалентную ей результирующую программу на языке машинных команд или на языке ассемблера.



Итерпретатор – программа, которая воспринимает исходную программу на входном (исходном) языке и выполняет ее. Все составные части транслятора представляют собой динамически загружаемые библиотеки или модули со своими входными и выходными данными.Общая схема работы транслятора приведена на рисунке.

Лексический анализ – часть компилятора, которая читает литеры программы на исходном языке и строит из них слова (лексемы) исходного языка. С теоретической точки зрения лексический анализатор не является обязательной частью компилятора.

Синтаксический разбор – основная часть компилятора на этапе анализа, выполняющая выделение синтаксических конструкций в тексте исходной программы, обработанном лексическим анализатором.

Семантический анализ – часть компилятора, проверяющая правильность текста исходной программы с точки зрения семантики входного языка. Кроме непосредственной проверки, выполняется преобразования текста, требуемые семантикой входного языка.

Подготовка к генерации кода – фаза, на которой компилятором выполняются предварительные действия, непосредственно связанные с синтезом текста результирующей программы, но еще не ведущие к порождению текста на выходном языке, например идентификация элементов языка, распределение памяти.

Генерация кода – фаза, непосредственно связанная с порождением команд, составляющих предложения выходного языка и в целом текст результирующей программы. Это основная фаза на этапе синтеза результирующей программы. Кроме непосредственного порождения текста результирующей программы генерация обычно включает в себя оптимизацию – процесс, связанный с обработкой уже порожденного текста. Иногда оптимизацию выделяют в отдельную фазу компиляции, так как она оказывает существенное влияние на качество и эффективность результирующей программы.

Таблицы идентификаторов – это специальным образом организованные наборы данных, служащие для хранения информации об элементах исходной программы, которые затем используются для порождения текста результирующей программы.

Литература – основная: п.1 , дополнительная: п.1 рабочей программы, интернет-ресурсы.

Лекция 3

Тема Особенности фаз компиляций

Цель – организация целенаправленной познавательной деятельности студентов по овладению программным материалом.

Задача – изучение важных понятий, реализующих внутренние механизмы работы транслятора.

Ключевые вопросы – характеристика основных фаз компиляции.

План лекции

1. Особенности лексического анализа
2. Особенности синтаксического анализа
3. Однопроходные и многопроходные компиляторы

С теоретической точки зрения лексический анализатор не является обязательной частью компилятора. Все его функции могут выполняться на этапе синтаксического разбора, поскольку полностью регламентированы синтаксисом входного языка. Однако существует несколько причин, по которым в состав практически всех компиляторов включают лексический анализ:

применение лексического анализатора упрощает работу с текстом исходной программы на этапе синтаксического разбора и сокращает объем обрабатываемой информации, поскольку структурирует исходный текст программы и отбрасывает всю незначащую информацию;

для выделения в тексте лексем и их разбора можно применять простую, эффективную и теоретически хорошо проработанную технику анализа, в то время как на этапе синтаксического анализа конструкций исходного языка используются достаточно сложные алгоритмы разбора;

сканер отделяет сложный по конструкции синтаксический анализатор от работы с текстом исходной программы, структура которого может варьироваться в зависимости от версии входного языка. В этом случае для перехода от одной версии языка к другой достаточно только перестроить относительно простой лексический анализатор

Функции, выполняемые лексическим анализатором, и состав лексем, которые он выделяет, могут меняться в зависимости от реализации компилятора.

В основе синтаксического анализатора лежит распознаватель текста программы, построенный на основе грамматики входного языка. Как правило, синтаксические конструкции языков программирования могут быть описаны с помощью контекстно-свободных грамматик (КС-грамматик), реже встречаются языки, которые могут быть описаны с помощью регулярных грамматик. Чаще всего регулярные грамматики применимы к языкам ассемблера, а синтаксис языков высокого уровня основан на грамматике КС-языков. В реальных компиляторах состав этих фаз компиляции может несколько отличаться от выше рассмотренного – некоторые из них могут быть разбиты на составляющие, другие, напротив, объединены в одну фазу.

Порядок выполнения фаз компиляции также может меняться в разных вариантах компиляторов. В одном случае компилятор просматривает текст исходной программы, сразу выполняет все фазы компиляции и получает результат – объектный код. В другом варианте он выполняет над исходным текстом только некоторые из фаз компиляции и получает не конечный результат, а набор некоторых промежуточных данных. Эти данные затем снова подвергаются обработке, причем этот процесс может повторяться несколько раз. Реальные компиляторы, как правило, выполняют трансляцию текста исходной программы за несколько проходов.

Проход – процесс последовательного чтения компилятором данных из внешней памяти, их обработки и помещения результата работы во внешнюю память. Чаще всего один проход включает в себя выполнение одной или нескольких фаз компиляции. Результатом промежуточных проходов является внутреннее представление исходной программы, результатом последнего прохода – объектная программа.

Литература – основная: п.1 , дополнительная: п.1, рабочей программы, интернет-ресурсы.

Лекция 4

Тема Организация таблиц идентификаторов

Цель – организация целенаправленной познавательной деятельности студентов по овладению программным материалом.

Задача – изучение способов организации таблиц идентификаторов.

Ключевые вопросы – логарифмический поиск, метод бинарного дерева, хэш-адресация

План лекции

1. Понятие таблицы идентификаторов
2. Простейшие методы построения таблиц идентификаторов
3. Метод бинарного дерева
4. Хэш-адресация, рехэширование

Выделение идентификаторов и других элементов исходной программы происходит на фазе лексического анализа. Для хранения найденных идентификаторов и их характеристик используются специальные хранилища данных, называемые *таблицами символов,* или *таблицами идентификаторов.* Любая таблица идентификаторов состоит из набора полей, количество которых равно числу различных идентификаторов, найденных в исходной программе. Каждое поле содержит в себе полную информацию о данном элементе таблицы. Компилятор может работать как с одной, так и с несколькими таблицам идентификаторов.

Таблицы идентификаторов организуются таким образом, чтобы компилятор имел возможность максимально быстрого поиска требуемого ему элемента. Простейший способ ее организации состоит в добавлении новых элементов в порядке их поступления. В этом случае таблица идентификаторов представляет собой неупорядоченный массив информации. Поиск более эффективен в таблице, элементы которой упорядочены (отсортированы) согласно некоторому порядку. Методом поиска в упорядоченном списке является *бинарный* (или *логарифмический*) *поиск.* Для сокращения времени поиска искомого элемента в таблице идентификаторов без значительного увеличения времени ее заполнения, надо отказаться от организации таблицы в виде непрерывного массива данных.

В методе бинарного дерева каждый узел представляет собой элемент таблицы, причем корневой узел является первым элементом, встреченным при заполнении таблицы. Для определенности ветви дерева называют «правая» и «левая». Первый идентификатор помещается в вершину дерева. Все дальнейшие идентификаторы попадают в дерево по следующему алгоритму:

Шаг 1. Выбрать очередной идентификатор из входного потока данных. Если его нет – построение дерева закончено.

Шаг 2. Сделать текущим узлом дерева корневую вершину.

Шаг 3. Сравнить очередной идентификатор с идентификатором, содержащимся в текущем узле дерева.

Шаг 4. Если очередной идентификатор меньше, то перейти к шагу 5, если равен – сообщить об ошибке и прекратить выполнение алгоритма, иначе перейти к шагу 7.

Шаг 5. Если у текущего узла существует левая вершина, сделать ее текущим узлом и вернуться к шагу 3, иначе перейти к шагу 6.

Шаг 6. Создать новую вершину, поместить в нее очередной идентификатор, сделать эту новую вершину левой вершиной текущего узла и вернуться к шагу 1.

Шаг 7. Если у текущего узла существует правая вершина, сделать ее текущим узлом и вернуться к шагу 3, иначе перейти к шагу 8.

Шаг 8. Создать новую вершину, поместить в нее очередной идентификатор, сделать эту новую вершину правой вершиной текущего узла и вернуться к шагу 1.

Лучших результатов можно достичь методами на основе хэш-функций и хэш-адресации.

Хэш-функцией F называется некоторое отображение множества входных элементов R на множество целых неотрицательных чисел Z: $F(r) = n$, $r \in R$, $n \in Z$. При работе с таблицей идентификаторов хэш-функция выполняет отображение имен идентификаторов на множество целых неотрицательных чисел.

Хэш-адресация заключается в использовании значения, возвращаемого хэш-функцией, в качестве адреса ячейки из некоторого массива данных.

Литература – основная: п.1 , дополнительная: п.1 рабочей программы, интернет-ресурсы.

Лекция 5

Тема Формальные языки и грамматики

Цель – организация целенаправленной познавательной деятельности студентов по овладению программным материалом.

Задача – изучение базовых определений и понятий теории грамматик.

Ключевые вопросы – терминальные и нетерминальные символы, правила грамматики, способы задания грамматики в форме Бэкуса-Наура, классификация формальных языков и грамматик

План лекции

1.Основные определения

2. Формы записи грамматик

3. Классификация грамматик

4.Классификация формальных языков

Алфавит — счетное множество допустимых символов языка. Будем обозначать это множество символом V. Согласно формальному определению, алфавит не обязательно должен быть конечным множеством, но реально существующие языки строятся на основе конечных алфавитов.

Цепочка символов α является цепочкой над алфавитом V: α (V), если в нее входят только символы, принадлежащие множеству символов V. Для любого алфавита V пустая цепочка λ может, как являться, так и не являться цепочкой λ *(V)*. Это условие оговаривается дополнительно.

Если V — некоторый алфавит, то:

V^+ — множество всех цепочек над алфавитом V без λ;

V* — множество всех цепочек над алфавитом V, включая λ.

Справедливо равенство: $V^* = V^+ \cup \{\lambda\}$.

Языком L над алфавитом V: L(V) называется некоторое счетное подмножество цепочек конечной длины из множества всех цепочек над алфавитом V.

Грамматика - это описание способа построения предложений некоторого языка. Грамматика - математическая система, определяющая язык.

Формально грамматика G определяется как четверка G(VT,VN, P, S), где:

VT – множество терминальных символов или алфавит терминальных символов;

VN – множество нетерминальных символов или алфавит нетерминальных символов;

P – множество правил (продукций) грамматики, вида $\alpha \text{->} \beta$, где $\alpha \in (VN \cup VT)^+$, $\beta \in (VN \cup VT)^*$;

S – целевой (начальный) символ грамматики $S \in VN$.

Язык, заданный грамматикой G, обозначается как L(G).

Согласно классификации, предложенной Н.Хомским, формальные грамматики классифицируются по структуре их правил:

Тип 0: грамматики с фразовой структурой;

Тип 1: контекстно-зависимые (КЗ) и неукорачивающие грамматики;

Тип 2: контекстно-свободные (КС) грамматики;

Тип 3: регулярные грамматики.

Одна и та же грамматика в общем случае может быть отнесена к нескольким классификационным типам. Для классификации грамматики всегда выбирают максимально возможный тип, к которому она может быть отнесена. Сложность грамматики обратно пропорциональна номеру типа, к которому относится грамматика.

Языки классифицируются в соответствии с типами грамматик, с помощью которых они заданы. Сложность языка также убывает с возрастанием номера классификационного типа языка.

В основе большинства современных языков программирования лежат контекстно-свободные языки.

Литература – основная: п.1 , дополнительная: п.1 рабочей программы, интернет-ресурсы.

Лекция 6

Тема Регулярные и автоматные грамматики

Цель – организация целенаправленной познавательной деятельности студентов по овладению программным материалом.

Задача – изучение базовых определений и понятий теории грамматик.

Ключевые вопросы – праволинейные и леволинейные грамматики, автоматные грамматики

План лекции

1. Однозначность и эквивалентность грамматик
2. Понятие автоматной грамматики.
3. Соотношение классов автоматной и регулярной грамматик.
4. Алгоритм преобразования грамматики регулярной грамматики к автоматному виду

Грамматика называется *однозначной,* если для каждой цепочки символов языка, заданного этой грамматикой, можно построить единственный левосторонний (и единственный правосторонний) вывод. Грамматика также называется однозначной, если для каждой цепочки символов языка, заданного этой грамматикой, существует единственное дерево вывода. В противном случае грамматика называется *неоднозначной.*

Если грамматика является неоднозначной, необходимо попытаться преобразовать ее в однозначный вид.

Две грамматики эквивалентны, если они задают один и тот же язык.

Леволинейные грамматики G (**VT,VN, P**, S), **V = VN** $\cup$ **VT** могут иметь правила двух видов: A→Bγ или A→γ, где A, B$\in$**VN**, $\gamma \in$ **VT*.**

Праволинейные грамматики G(**VT, VN, P**, S), **V = VN** $\cup$ **VT** могут иметь правила тоже двух видов: A→γ B или A→γ, где A, B$\in$**VN**, $\beta \in$ **VT***.

Эти два класса грамматик эквивалентны и относятся к типу регулярных грамматик.

Регулярные грамматики используются при описании простейших конструкций языков программирования: идентификаторов, констант, строк, комментариев т. д. Эти грамматики исключительно просты и удобны в использовании, поэтому в компиляторах на их основе строятся функции лексического анализа входного языка.

Среди всех регулярных грамматик выделяют отдельный класс – автоматные грамматики. Они также могут быть леволинейными и праволинейными.

Разница между автоматными и обычными регулярными грамматиками заключается в том, что где в правилах обычных регулярных грамматик может присутствовать цепочка терминальных символов, в автоматных грамматиках может присутствовать только один терминальный символ. Любая автоматная грамматика является регулярной, но не наоборот – не всякая регулярная грамматика является автоматной.

Доказано, что классы обычных регулярных грамматик и автоматных грамматик почти эквивалентны. Существует алгоритм, позволяющий преобразовать произвольную регулярную грамматику к автоматному виду.

Регулярные языки являются удобным типом языков. Для них разрешимы многие проблемы: эквивалентности двух языков, принадлежности языку заданной цепочки символов, пустоты языка.

Эти проблемы разрешимы вне зависимости от того, каким из трех способов задан язык:

1) регулярными грамматиками (праволинейным или леволинейными);
2) конечным автоматом;
3) регулярным множеством.

Литература – основная: п.1 , дополнительная: п.1 рабочей программы, интернет-ресурсы.

Лекция 7

Тема Общая схема работы распознавателя

Цель – организация целенаправленной познавательной деятельности студентов по овладению программным материалом.

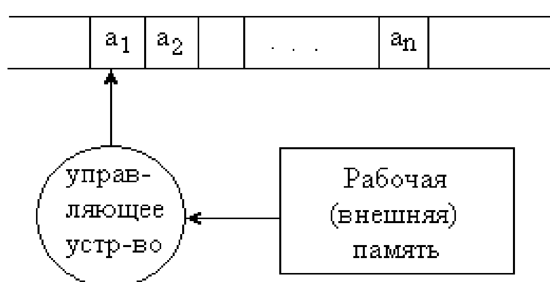
Задача – изучение основных принципов работы распознавателей

Ключевые вопросы – устройство управления, конфигурация распознавателя, такты работы, задача разбора.

План лекции

1. Компоненты распознавателя
2. Принцип работы распознавателя
3. Виды распознавателей

Распознаватель – это специальный автомат, который позволяет определить принадлежность цепочки символов некоторому языку. Задача распознавателя заключается в том, чтобы на основании исходной цепочки дать ответ на вопрос, принадлежит ли она заданному языку или нет. Распознаватель состоит из следующих компонентов:



- ленты, содержащей входную цепочку символов, и считывающей головки, обозревающей очередной символ в этой цепочке;

- устройства управления, координирующего работу распознавателя, имеющего некоторый набор состояний и конечную намять;

- внешней памяти, которая может хранить некоторую информацию в процессе работы распознавателя и имеет неограниченный объем.

Для распознавателя всегда задается определенная конфигурация, которая считается начальной. Кроме начального состояния для распознавателя задается одна или несколько конечных конфигураций. Распознаватель *допускает входную цепочку символов а*, если, находясь в начальной конфигурации и получив на вход эту цепочку, он может проделать последовательность шагов, заканчивающуюся одной из его конечных конфигураций.

В процессе своей работы распознаватель может выполнять некоторые элементарные операции;

- чтение очередного символа из входной цепочки;
- сдвиг входной цепочки на заданное количество символов (вправо или влево);
- доступ к рабочей памяти для чтения или записи информации;
- преобразование информации в памяти УУ, изменение состояния УУ.

Распознаватель работает по шагам, или тактам. В начале такта, как правило, считывается очередной символ из входной цепочки, и в зависимости от этого символа УУ определяет, какие действия необходимо выполнить. Вся работа распознавателя состоит из последовательности тактов. В начале каждого такта состояние распознавателя определяется его конфигурацией. В процессе работы конфигурация распознавателя меняется.

Для каждого из типов языков существует свой тип распознавателя.

Для языков с фразовой структурой (тип 0) – недетерминированный двусторонний автомат, имеющий неограниченную внешнюю память.

Для контекстно-зависимых языков (тип 1) распознавателями являются двусторонние недетерминированные автоматы с линейно-ограниченной внешней памятью. Такой алгоритм распознавателя уже может быть реализован в программном обеспечении компьютера.

Для контекстно-свободных языков (тип 2) распознавателями являются односторонние недетерминированные автоматы с магазинной (стековой) внешней намятые — МП-автоматы.

Для регулярных языков (тип 3) распознавателями являются односторонние недетерминированные автоматы без внешней памяти – конечные автоматы.

Литература – основная: п.1 , дополнительная: п.1 рабочей программы, интернет-ресурсы.

Лекция 8

Тема Конечные автоматы и автоматы с магазинной памятью

Цель – организация целенаправленной познавательной деятельности студентов по овладению программным материалом.

Задача – изучение основных принципов работы распознавателей

Ключевые вопросы – способы задания автоматов, принципы работы, начальное и конечные состояния, допускаемые цепочки.

План лекции

1. Компоненты распознавателя
2. Принцип работы распознавателя
3. Виды распознавателей

Под автоматом понимают не реально существующее устройство, а некоторую математическую модель, свойства и поведение которой можно изучать. Конечные автоматы применяются при построении компиляторов, благодаря следующим свойствам:

1. КА может решать простые задачи компиляции. В частности, лексический блок почти всегда строится на его основе.
2. Обработка одного входного символа требует небольшого числа операций, что обеспечивает быстроту работы.
3. Моделирование КА требует фиксированного объема памяти.
4. Существует ряд теорем и алгоритмов, позволяющих конструировать и упрощать КА.

Конечный автомат является простейшим из моделей теории автоматов и служит управляющим устройством для всех остальных автоматов. Он задается как M(Q, V, δ, q_0, F), где Q – конечное множество состояний автомата, **V** – алфавит входных символов, δ – функция переходов, $\delta(a,q)=R$, $a \in V$, $q \in Q$, $R \subseteq Q$, q_0 – начальное состояние автомата ($q_0 \in Q$), F – непустое множество конечных состояний автомата. Конфигурация автомата на каждом шаге работы определяется тройкой (q,ω,n), где q – текущее состояние автомата, ω – цепочка входных символов, n – положение указателя во входной цепочке. Конечный автомат часто представляют в виде диаграммы или графа переходов автоматов. Существуют алгоритмы приведения конечного автомата к детерминированному виду и алгоритм его минимизации.

МП-автомат определяют как R (Q, V, Z, δ , qo, Zo, F), где Q – множество состояний автомата; V – алфавит входных символов автомата; Z – специальный конечный алфавит магазинных символов автомата; δ – функция переходов автомата; $q_o \in Q$ – начальное состояние автомата; $z_0 \in Z$ – начальный символ магазина; $F \in Q$ – множество конечных состояний.

МП-автомат в отличие от обычного КА имеет стек (магазин), в который можно помещать специальные (магазинные) символы. Обычно это терминальные и нетерминальные символы грамматики языка.

Существует 2 варианта реализации. В первом варианте на каждом шаге алгоритм должен запоминать все возможные следующие состояния МП-автомата, выбирать один одно из них, переходить в это состояние и действовать до тех пор, пока либо не будет достигнуто конечное состояние автомата, либо автомат не перейдет в такую конфигурацию, когда следующее состояние будет не определено. Если будет достигнуто одно из конечных состояний – входная цепочка принята, работа алгоритма завершается. В противном случае алгоритм должен вернуть автомат на несколько шагов назад, когда еще был возможен выбор одного из набора следующих состояний. Алгоритм завершается с ошибкой, когда все возможные варианты работы автомата перебраны и при этом не было достигнуто ни одного из возможных конечных состояний.

Во втором варианте алгоритм должен на каждом шаге при возникновении неоднозначности с несколькими возможными следующими состояниями автомата запускать новую копию для обработки каждого из этих состояний. Алгоритм завершается, если хотя бы одна из выполняющихся его копий достигнет одного из конечных состояний. При этом работа всех остальных копий прекращается. Если ни одна из копий не достигла конечного состояния МП-автомата, алгоритм завершается с ошибкой. Этот вариант алгоритма требует параллельных вычислений и, следовательно, сложен в реализации.

Литература – основная: п.1 , дополнительная: п.1 рабочей программы, интернет-ресурсы.

Лекция 9

Тема Средства трассировки и отладки программ

Цель – организация целенаправленной познавательной деятельности студентов по овладению программным материалом.

Задача – изучение основных принципов отладки и трассировки

Ключевые вопросы – диагностика ошибок, способы их обнаружения и исправления.

План лекции

1. Основные понятия
2. Возможности трассировки
3. Мобильность программного обеспечения

Отладка – это процесс поиска и исправления ошибок в программе, препятствующих корректной работе программы. Правила, создания программ, препятствующие возникновению ошибок:

Выбор лучшего алгоритма позволяет кардинально повысить скорость вычислений и упростить программу.

Разделяйте задачу на малые части и оформляйте их в виде законченных программных модулей – прежде всего функций.

Не скупитесь на программные комментарии – чем их больше, тем понятнее программа (ясность программы в большинстве случаев важнее скорости ее работы).

Тщательно готовьте сообщения об ошибках и диагностические сообщения, а также наименования программных модулей и описания их назначения.

Тщательно производите диагностику программных модулей – хорошо спроектированный модуль должен диагностировать любые виды ошибочных ситуаций и реагировать на них адекватным образом.

Используйте имена переменных и констант с использованием понятных по смыслу обозначений, не используйте в именах зарегистрированные идентификаторы команд и функций.

Заменяйте циклы функциями обработки списков, например функциями суммирования и произведения. Применяйте эффективные варианты упрощенных операторов и функций.

В максимальной степени используйте функции ядра системы. Обращайтесь к пакетам расширений только в том случае, когда это действительно необходимо.

Обращайте особое внимание на реализацию механизма контекстов, позволяющего избежать грубых ошибок при модернизации различных объектов программ, прежде всего наборов функций.

Избегайте недокументированных приемов программирования. Такие программы в момент создания могут выглядеть удивительно эффектными и потрясающе оригинальными, но возможно, что в следующей версии системы они перестанут работать вообще. Основной смысл использования встроенного отладчика состоит в управляемом выполнении. Отслеживая выполнение каждой инструкции, вы можете легко определить, какая часть вашей программы вызывает проблемы. В отладчике предусмотрено пять основных механизмов управления выполнением программы, которые позволяют: выполнять инструкции по шагам;
трассировать инструкции; выполнять программу до заданной точки; находить определенную точку; останавливать выполнение программы.

Само по себе выполнение программы по шагам может быть недостаточно полезным, но управляемое выполнение дает возможность проверять состояние программы и ее данных, например, отслеживать вывод программы и значения ее переменных.

Выполнение по шагам – это простейший способ выполнения программы по элементарным фрагментам. Выбор команды Run|Step Over (клавиша F8) вызывает выполнение отладчиком всего кода в подсвеченной строке, включая любые вызываемые на ней процедуры или функции. После этого подсвеченная строка перемещается на следующую строку программы.

Литература – основная: п.1 , дополнительная: п.3, 4 рабочей программы, интернет-ресурсы.

Лекция 10

Тема Управление задачами

Цель – организация целенаправленной познавательной деятельности студентов по овладению программным материалом.

Задача – изучение основных принципов работы операционных систем

Ключевые вопросы – планирование и диспетчеризация процессов и задач.

План лекции

1.Функции операционной системы по управлению задачами

2. Стратегии планирование процессов

3. Дисциплины диспетчеризации

Функции операционной системы группируются в соответствии с типами ресурсов, которыми управляет ОС. Иногда такие группы функций называют подсистемами. Наиболее важными из них являются подсистемы управления памятью, файлами и внешними устройствами. Общими для всех ресурсов являются подсистемы пользовательского интерфейса, защиты данных и администрирования.

Основные функции ОС по управлению задачами включают в себя:

1) создание и удаление задач;

2) планирование процессов и диспетчеризация задач;

3) синхронизация задач.

Процессы, порождаемые пользователями и их приложениями, называются *пользовательскими*, а инициированные самой ОС для выполнения своих функций – *системными*. На протяжении своего существования процесс может быть многократно прерван и продолжен.

Планирование процессов осуществляется на основе, некоторой стратегии, например:

1) по возможности сохранить порядок окончания процессов таким, каков был порядок их запуска;

2) отдавать предпочтение более коротким процессам;

3) предоставлять всем пользователям одинаковые услуги (например, время ожидания).

Понятие стратегии обслуживания обычно применяется к процессу, а не к задаче (процесс может состоять из нескольких потоков или задач). Понятие же диспетчеризации применимо к задаче.

Дисциплина диспетчеризации − правило, по которому формируют очередь готовых к выполнению задач. Существует два больших класса этих правил: бесприоритетные и приоритетные. Первые бывают линейные и циклические, вторые – с динамическими и статическими.

Помимо деления согласно приведенной классификации дисциплины диспетчеризации делятся на:

1) вытесняющие, в которых решение о переключении процессора с одного процесса на другой принимается операционной системой в соответствии с принятой стратегией планирования;

2) невытесняющие, в которых текущий процесс занимает процессор в течение времени, необходимого для выполнения всех возможных вычислений, и только после этого передаёт его другому процессу.

Дисциплины диспетчеризации:

дисциплина FCFS (first come – first served – первым пришёл первым обслужен), в соответствии с которой задачи обслуживаются в «порядке очереди».

дисциплина обслуживания SJN (Shortest Job Next или следующим будет выполняться кратчайшее задание)

SRT (Shortest Remaining Time или следующим будет выполняться задание, требующие меньше всего времени).

дисциплина обслуживания RR (Round Robin − круговая или карусельная).

дисциплина LCFS (last come – first served – первым пришёл последним обслужен) - обслуживание по принципу стека.

Литература – дополнительная: п. 4 рабочей программы, интернет-ресурсы.

Лекция 11

Тема Управление файлами

Цель – организация целенаправленной познавательной деятельности студентов по овладению программным материалом.

Задача – изучение особенностей файловых систем

Ключевые вопросы – файловые системы, подсистемы управления файлами, файловая система FAT, NTFS.

План лекции

1. Соотнесение понятий «файловая система» и «система управления файлами»
2. Задачи системы управления файлами
3. Примеры систем управления файлами

Специальное системное программное обеспечение, реализующее работу с файлами по принятым спецификациям файловой системы, часто называют *системой управления файлами.* Она отвечает за создание, уничтожение, организацию, чтение, запись, модификацию и перемещение файловой информации, а также за управление доступом к файлам и за управление ресурсами, которые используются файлами. Как правило, все современные операционные системы имеют соответствующие системы управления файлами. А некоторые - возможность работы с несколькими файловыми системами (либо с одной из нескольких, либо сразу с несколькими одновременно). В этом случае говорят о *монтируемых файловых системах* (монтируемую систему управления файлами можно установить как дополнительную), и в этом смысле они самостоятельны.

Очевидно, что система управления файлами, будучи компонентом операционной системы, не является независимой от нее, поскольку активно использует соответствующие вызовы API. С другой стороны, системы управления файлами сами дополняют API новыми вызовами.
Если термин *файловая система* определяет, прежде всего, принципы доступа к данным, организованным в файлы. Тот же термин используют и по отношению к конкретным файлам, расположенным на том или ином носителе данных. А термин *система управления файлами* следует употреблять по отношению к конкретной реализации файловой системы, то есть это — комплекс программных модулей, обеспечивающих работу с файлами в конкретной операционной системе.

Файловая система FAT32 является полностью самостоятельной 32-разрядной файловой системой и содержит многочисленные усовершенствования и дополнения по сравнению с предыдущими реализациями FAT. Самое принципиальное отличие в том, что FAT32 намного эффективнее расходует дисковое пространство.

NTFS (New Technology File System) — файловая система новой технологии. Действительно, файловая система NTFS по сравнению FAT16 (и даже FAT32)содержит ряд значительных усовершенствований и изменений. С точки зрения пользователей файлы по-прежнему хранятся в каталогах (называемых *папками*). Однако в ней появилось много новых особенностей и возможностей: надежность, ограничения доступа к файлам и каталогам, расширенная функциональность, поддержка дисков большого объема.
Одним из основных понятий, используемых при работе с NTFS, является понятие *тома* (volume). Том означает логическое дисковое пространство, которое может быть воспринято как логический диск, то есть том может иметь букву (буквенный идентификатор) диска. Частным случаем тома является логический диск. Файловая система NTFS поддерживает размеры кластеров от 512 байт до 64 Кбайт; неким стандартом же считается кластер размером 2 или 4 Кбайт.

Все дисковое пространство в NTFS делится на две неравные части. Первые 12 % диска отводятся под так называемую зону MFT (главная таблица файлов). Остальные 88 % тома представляют собой обычное пространство для хранения файлов.

Литература – дополнительная: п. 4 рабочей программы, интернет-ресурсы.

Лекции 12, 13 Подсистема управлению памятью.

Цель - организация целенаправленной познавательной деятельности студентов по овладению программным материалом.

Задача – изучение функций операционной системы по управлению памятью и особенностей их реализации.

Ключевые вопросы – структуризация виртуального адресного пространства, свопинг и виртуальная память, страничный, сегментный и сегментно-страничный механизмы виртуальной памяти.

План лекции

1. Функции управления оперативной памятью в мультипрограммных операционных системах
2. Виртуальное адресное пространство процессов
3. Страничное, сегментное распределение и сегментно-страничная организация виртуальной памяти

Функциями ОС по управлению памятью в мультипрограммной системе являются:

отслеживание свободной и занятой памяти,

выделение памяти процессам и ее освобождение по завершении процессов,

вытеснение кодов и данных из оперативной памяти на диск, когда размеры основной памяти не достаточны для размещения в ней всех процессов, и возвращение их обратно, когда в них освобождается место,

настройка адресов программы на конкретную область физической памяти.

защита памяти, которая состоит в том, чтобы не позволить выполняемому процессу записывать и читать данные другого процесса.

Для обеспечения приемлемого уровня мультипрограммирования используется методы, в которых образы некоторых процессов целиком или полностью выгружаются на диск – свопинг и виртуальная память.

Ключевой проблемой виртуальной памяти является преобразование виртуальных адресов в физические. Ее решение зависит от способа структуризации виртуального адресного пространства, который реализуется методами: страничная виртуальная память; сегментная виртуальная память; сегментно-страничная организация памяти.

Для временного хранения сегментов и страниц на диске отводится специальная область – страничный файл.

При страничном распределении виртуальное адресное пространство каждого процесса делится на части одинакового фиксированного для данной системы размера, называемые *виртуальными страницами*. Вся оперативная память машины также делится на части того же размера, называемые *физическими страницами.* Для каждого процесса ОС создает *таблицу страниц.* При каждом обращении к памяти выполняется поиск номера виртуальной страницы, содержащей требуемый адрес. Если нужная виртуальная страница в данный момент выгружена на диск, то происходит так называемое *страничное прерывание*. Выполняющийся процесс переводится в состояние ожидание, и активизируется другой. Параллельно программа обработки страничного прерывания находит на диске требуемую виртуальную страницу и пытается загрузить ее.

Размер сегмента определяется с учетом смыслового значения содержащейся в них информации. Механизм перемещения данных из оперативной памяти на диск аналогичен предыдущему методу.

Сегментно-страничное распределение памяти направлено на реализацию достоинств обоих подходов. Виртуальное адресное пространство процесса разделено на сегменты так же как при сегментной организации памяти, что позволяет определять разные права доступа к разным частям кодов и данных программы. Перемещение же данных между памятью и диском осуществляется не сегментами, а страницами. Для этого каждый виртуальный сегмент и физическая память делятся на страницы равного размера.

Литература – дополнительная: п. 4 рабочей программы, интернет-ресурсы.

Лекции 14 Управление вводом-выводом.

Цель - организация целенаправленной познавательной деятельности студентов по овладению программным материалом.

Задача – изучение функций операционной системы по управлению вводом-выводом.

Ключевые вопросы – режим обмена с опросом готовности устройства ввода-вывода и режим обмена с прерываниями, виртуализация устройств ввода-вывода

План лекции

1. Функции ОС по управлению вводом-выводом
2. Режимы управления вводом-выводом
3. Разделяемые и неразделяемые устройства ввода-вывода

Управление вводом-выводом — это одна из основных функций любой операционной системы. Программирование ввода-вывода является наиболее сложным и трудоемким, требующим очень высокой квалификации. Поэтому программный код, реализующий операции ввода-вывода, сначала стали оформлять в виде системных библиотечных процедур, а потом включили в состав операционной системы. Организация ввода-вывода в различных операционных системах имеет много общего, но с другой стороны, реализация ввода-вывода в ОС сильно отличается. В большинстве ныне используемых систем эти моменты вообще, как правило, подробно не описаны (исключением являются только системы Linux и FreeBSD, для которых имеются комментированные исходные тексты), а детально описываются только функции API, реализующие ввод-вывод.

Существуют разделяемые и неразделяемые устройства ввода-вывода. Примерами разделяемого устройства могут служить накопитель на магнитных дисках, устройство чтения компакт-дисков. Это устройства с прямым доступом. Примеры неразделяемых устройств – принтер, накопитель на магнитных лентах. Это устройства с последовательным доступом. Многие устройства и, прежде всего, устройства с последовательным доступом не допускают совместного использования. Такие устройства могут стать *закрепленными* за процессом на все время его жизни. Однако это приводит к невозможности параллельного выполнения вычислительных процессов. Для организации совместное использование многими параллельно выполняющимися задачами неразделяемых устройств ввода-вывода вводится понятие *виртуальных устройств.*

Любые операции по управлению вводом-выводом объявляются привилегированными и могут выполняться только кодом самой операционной системы.

Существует два основных режима ввода-вывода: *режим обмена с опросом готовности* устройства ввода-вывода и *режим обмена с прерываниями.*

Центральный процессор посылает команду устройству управления, которое исполняет команду, транслируя сигналы, понятные ему и центральному устройству, в сигналы, понятные устройству ввода-вывода. После выполнения команды устройство ввода-вывода (или его устройство управления) выдает *сигнал готовности,* который сообщает процессору о том, что можно выполнять новую команду для продолжения обмена данными.

Однако поскольку быстродействие устройства ввода-вывода намного меньше быстродействия центрального процессора (порой на несколько порядков), сигнал готовности приходится очень долго ожидать, постоянно опрашивая соответствующую линию интерфейса на наличие или отсутствие нужного сигнала. Посылать новую команду, не дождавшись сигнала готовности, сообщающего об исполнении предыдущей команды, бессмысленно. В режиме опроса готовности драйвер, управляющий процессом обмена данными с внешним устройством, как раз и выполняет в цикле команду «проверки на наличие сигнала готовности». До тех пор пока сигнал готовности не появится, драйвер ничего другого не делает. При этом, естественно, нерационально используется время центрального процессора. Гораздо выгоднее, выдав команду ввода-вывода, на время забыть об устройстве ввода-вывода и перейти на выполнение другой программы. А появление сигнала готовности трактовать как запрос на прерывание от устройства ввода-вывода. Эти сигналы готовности и являются *сигналами запроса на прерывание.*

Литература – дополнительная: п. 4 рабочей программы, интернет-ресурсы.

Лекции 15 Системные таблицы ввода-вывода.

Цель - организация целенаправленной познавательной деятельности студентов по овладению программным материалом.

Задача – изучение функций операционной системы по управлению вводом-выводом.

Ключевые вопросы – назначение и содержание таблиц ввода-вывода

План лекции

1. Основные таблицы ввода-вывода, их назначение
2. Примерное содержание таблиц
3. Синхронный и асинхронный ввод-вывод

Для управления всеми операциями ввода-вывода и отслеживания состояния всех ресурсов, занятых в обмене данными, операционная система должна иметь соответствующие информационные структуры. Эти структуры часто называют таблицами ввода-вывода.

Каждая операционная система ведет свои таблицы ввода-вывода, их состав, количество и назначение может сильно отличаться. В некоторых операционных системах вместо таблиц создаются списки, хотя использование статистических структур данных для организации ввода-вывода, как правило, приводит к более высокому быстродействию. ОС создает по крайней мере три системные таблицы.

Первая таблица (или список) содержит информацию обо всех устройствах ввода вывода, подключенных к вычислительной системе – *таблица оборудования.* Каждый ее элемент содержит: тип устройства, его конкретная модель, символическое имя, характеристики устройства; способ подключения устройства; номер и адрес канала (и подканала), если такие используются для управления устройством; информацию о драйвере, который должен управлять этим устройством, адреса секции запуска и секции продолжения драйвера; информацию о том, используется или нет буферизация при обмене данными с устройством, «имя» (или просто адрес) буфера; установку тайм-аута и ячейки для счетчика тайм-аута; состояние устройства; поле указателя для связи задач, ожидающих устройство.

Вторая таблица предназначена для реализации принципа независимости от устройства. Ее назначение – установление связи между виртуальными (логическими) устройствами и реальными устройствами, описанными посредством первой таблицы (таблицы оборудования). Она позволяет супервизору перенаправить запрос на ввод-вывод па приложения в те программные модули и структуры данных, которые (или адреса которых) хранятся в соответствующем элементе первой таблицы. Во многих многопользовательских системах таких таблиц несколько: одна общая и по одной на каждого пользователя, что позволяет строить необходимые связи между логическими устройствами (символьными именами устройств) и реальными физическими устройствами, которые имеются в системе.

Третья таблица – *таблица прерываний* – необходима для организации обратной связи между центральной частью и устройствами ввода-вывода. Эта таблица указывает для каждого сигнала запроса на прерывание тот элемент UCB, который сопоставлен данному устройству.

В ряде сложных операционных систем, а к ним следует отнести все современные 32-разрядные системы для персональных компьютеров, имеется гораздо больше системных таблиц или списков, используемых для организации управления one рациями ввода-вывода.

Режим обмена с прерываниями по своей сути является режимом асинхронного управления. Для того чтобы не потерять связь с устройством (после выдачи процессором очередной команды по управлению обменом данными и переключения его на выполнение других программ), может быть запущен отсчет времени, в течение которого устройство обязательно должно выполнить команду и выдать сигнал запроса на прерывание.

Максимальный интервал времени, в течение которого устройство ввода-вывода или его контроллер должны выдать сигнал запроса на прерывание, называют *установкой тайм-аута.*

Литература – дополнительная: п. 4 рабочей программы, интернет-ресурсы.

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ

Методические указания по изучению дисциплины

Для успешного освоения данной дисциплины студентам предлагаются:

- содержание разделов и тем дисциплины, включающей лекционные и лабораторные занятия;
- контрольные вопросы для самостоятельной работы;
- список рекомендуемой основной и дополнительной литературы;
- вопросы к экзамену.

В течении семестра студент не только должен изучать материал лекций, но и готовить вопросы самостоятельной работы на основе рекомендуемой литературы.

Основными формами самостоятельной (внеаудиторной) работы студентов являются:

подготовка отдельных вопросов по темам программы;

участие в научных и научно-практических студенческих конференциях;

подготовке к лабораторной работе по определенной теме;

подготовка и написание отчетов к лабораторным работам;

подготовка к промежуточному и итоговому контролю.

Самостоятельная работа начинается до прихода студента на лекцию. Весьма эффективно использование «системы опережающего чтения», т.е. предварительного прочитывания лекционного материала, содержащегося в учебниках, учебных пособиях, в результате чего закладывается база для более глубокого восприятия лекции.

В процессе организации самостоятельной работы большое значение имеют консультации преподавателя, в ходе которых решаются многие проблемы изучаемого курса, уясняются наиболее сложные вопросы.

Контроль самостоятельной работы осуществляется тестированием, которое может проводиться в письменной форме, либо с использованием компьютерной системы тестирования. Примерные тестовые задания приведены в рабочей программе. Тест включает в себя вопросы с открытыми и закрытыми вариантами ответов. Результаты тестирования включены в балльно-рейтинговую систему оценки знаний (см. рабочую программу).

Экзамен является завершающим этапом учебного процесса, на котором проводится подведение итогов всей самостоятельной работы студентов.

Подготовку к экзамену требуется начинать с просмотра перечня всех вопросов с целью оценки требуемого объема учебного материала, логики и структуры построения курса. С учетом накопленных за семестр знаний студент должен запланировать распределение времени на подготовку. Желательно зарезервировать время для повторения материала. Работа над каждым из вопросов рекомендуется прочитать конспект лекции, дополнительно прочитать рекомендованный учебник, если материал трудно усваивается. Завершается работа восстановление в памяти прочитанного.

Экзаменационный билет включает в себя два теоретических вопроса из перечня.

При оценке на экзамене учитывается: полнота ответа на поставленный вопрос, точность формулировок, логичность ответа, умение делать выводы, выявлять закономерности, соблюдение норм литературной речи и использования специализированной терминологии. Высшего балла заслуживает ответ, удовлетворяющий всем этим требованиям.

Методические указания к лабораторным работам

Лабораторные работы проводятся по подгруппам в компьютерном классе. Каждый студент получает индивидуальное задание в соответствии с вариантом.

Выполняя задание, студент пользуется материалом, изученным в тексте лабораторной работы.

Перед созданием любой программы требуется точно продумать алгоритм. Записать его блок-схемой или словесно. Надо четко определить, что в нее требуется ввести и что получить в результате, в какой последовательности выполнять действия. В случае необходимости выде-

лить циклические структуры и подпрограммы. В циклах четко определить параметры, задать их начальные значения, определить условия повторения и завершения цикла. В функциях определить количество передаваемых и возвращаемых значений.

При кодировании программы нужно определить тип используемых данных в зависимости от возможного диапазона принимаемых значений. При вводе величины не забывать осведомить об этом пользователя, а иногда сообщить и о типе, диапазоне или порядке ввода значений. Такое сообщение должно быть информативно и коротко. Вывод данных лучше сопровождать текстом и форматированием. Формат вывода можно уточнить при помощи модификаторов.

В именах переменных необходимо отражать их назначение, что повышает читаемость и понимание программы.

При записи сложных выражений нужно обращать внимание на приоритет операций. Текст программы лучше сопровождать краткими и информативными комментариями, что облегчает как понимание программы, так и ее отладку.

Объявление локальных переменных предпочтительнее по сравнению с глобальными.

Для отладки программы нужно запустить ее на выполнение несколько раз, задавая различные значения вводимых величин. Перед запуском необходимо иметь заранее подготовленные тестовые примеры, содержащие исходные данные и ожидаемые результаты. Их количество зависит от алгоритма. Проверьте реакцию программы на заведомо неверные исходные данные.

Для быстрого поиска ошибки в алгоритме рекомендуется выводить промежуточные данные.

При сдаче лабораторной работы студент должен продемонстрировать преподавателю созданную программу, правильно работающую, отлаженную.

Преподаватель, принимая лабораторную работу, тестирует программу студента и задает ему вопросы по конструкциям, используемым в программе и теоретическим основам программирования.

Технология выполнения лабораторных работ

РАБОТА С ФАЙЛОВОЙ СИСТЕМОЙ СРЕДСТВАМИ JAVASCRIPT

Лабораторная работа №1

Создание объекта файловой системы

Доступ к файловой системе с помощью языков на основе сценариев, таких как JavaScript и VBScript, в Windows обеспечивается через объект
FileSystemObject (FSO - объект файловой системы). Программы на JavaScript VBScript, использующие этот объект, могут интерпретироваться браузером IE5+, а также системой Windows Scripting Host (WHS), встроенной в Windows 98 и более поздние версии.

Операциям с файловой системой, выполняемые браузером пользователя с помощью сценариев будут предшествовать предупреждающие сообщения об опасности даже при работе с локальным компьютером. Поэтому рекомендуется использовать FSO не на клиентском компьютере, а на сервере.

Технология WHS позволяет свободно пользоваться FSO на локальном компьютере. Для этого создается программа на JavaScript в текстовом файле с расширением js и затем выполняется с помощью сервера сценариев Windows (файл wscript.exe, расположенный в папке Windows). В MS DOS аналогичная программа представлена файлом cscript.exe.

Программы JavaScript, написанные для выполнения браузером и WHS, во многом похожи, но и имеют ряд отличий. Сценарии для браузера размещаются в HTML-документе (обычно в контейнере <SCRIPT>) или в js-файле. Программы для WSH размещаются только в js-файлах. Для вывода сообщений в браузере используется, например, метод alert(). В WSH такого метода нет. Вместо него применяется метод WScript.Echo(), отсутствующий в браузере.

При работе с файловой системой и реестром Windows следует соблюдать осторожность, поскольку можно нечаянно не только потерять ценные данные, но и повредить операционную систему.

Чтобы получить доступ к файловой системе, необходимо создать для нее объект FileSystemObject (экземпляр FSO). Если ваша программа на JavaScript будет выполняться браузером как сценарий в HTML-документе, то для создания FSO можно использовать только следующее выражение:

```
var  fso = new ActiveXObject("Scripting.FileSystemObject”)
```

Если программа предназначена для выполнения с помощью WSH, то кроме указанного выше выражения можно использовать и такое:

```
var fso = WScript.CreateObject("Scripting.FileSystemODject")
```

Здесь fso — переменная (ее имя может быть произвольным), содержащая ссылку па объект файловой системы. Эта ссылка будет использоваться для применения методов и свойств объекта файловой системы. В дальнейшем будет применяться первый вариант создания FSO, поскольку он подходит и для браузера, и для WSH. Первый вариант соответствует вызову объекта FSO как элемента управления ActiveX, а второй — как объекта приложения Wscript.

После того как объект файловой системы создан, можно применить методы для создания и удаления папок и файлов, копирования и перемещения файлов, а также получения информации о дисках, папках и файлах. Существуют и другие методы, такие как открытие и закрытие файла, запись данных в файл и т. п.

Общий синтаксис:

Ссылка на объект файловой системы (FSO) для доступа к ее объектам:

```
var fso = new ActiveXObject("Scripting.FileSystemObject")
```

Методы доступа к существующим объектам и методы создания новых объектов:

```
var x = fso.методОбъекта(параметры)
```

Свойства и методы конкретного объекта — диска, папки или файла:

```
var y = x.свойство(параметры)
var z  =  x.метод(параметры)
```

Например, для получения информации о свободном пространстве на диске С следует выполнить следующий код:

```
var  fso = ActiveXObject("Scripting.FileSystemODject")   //  ссыпка  на  FSO

var d =  fso.GetDrive("C")
//  ссылка  на объект с  характеристиками диска  С

var  freespace  = d.FreeSpace //  значение  свободного пространства  в байтах

Wscript.Echo(freespace)          //  вывод сообщения
```

При этом ряд методов FSO разбивается на две группы: Get-методы и Create-методы. Названия методов из той или иной группы начинаются либо с Get, либо с Create. Get-методы предназначены для получения ссылок на уже существующие объекты (get — получить). Create-методы предназначены для создания объектов (create — создать). Однако методы создания возвращают ссылку на созданный объект. Поэтому при создании объекта Create-методом и необходимости ссылки на него, лучше сохранять ссылку, возвращаемую Create-методом в переменной для дальнейшего использования, а не применять Get-метод. Get-методы работают для уже существующих объектов, а Create-методы, кроме создания новых объектов, обеспечивают и доступ к этим новым объектам, такой же как и Get-методы.

Задание

В приведенных фрагментах программного кода содержатся ошибки. Исправьте их и создайте программно объект файловой системы различными способами. Продемонстрируйте вызов методов для созданного объекта.

ЛАБОРАТОРНАЯ РАБОТА №2
Тема **Работа с дисками**

Работа с дисками заключается в получении информации о них (объем свободного пространства, тип, готовность и т. п.). Информация о дисках важна, например, при создании новых папок и файлов. При создании, копировании или перемещении файла полезно сначала убедиться, что указанный диск существует, готов к работе и имеет достаточно свободного пространства. Знание серийного номера диска может использоваться, например, при решении задачи защиты программных продуктов от несанкционированного копирования.

Сначала создается объект FSO для доступа к файловой системе, и ссылка на него сохраняется в переменной, например fso. Далее используются методы и свойства для получения информации о диске. Пусть переменная dpath содержит букву, которой обозначен диск, или путь к какой-нибудь папке, начинающийся с буквы диска. Тогда метод fso.GetDrive(path) возвращает ссылку на объект, содержащий информацию о диске, указанном в dpath. Пусть эта ссылка сохранена в переменной d. Получить значения свойств этого объекта, которые являются характеристиками диска можно например так, свойство d.IsReady равно true, если диск готов, и false — в противном случае. Свойство d.FreeSpace содержит величину свободного пространства на диске в байтах. Чтобы определить, существует ли указанный в dpath диск, необходимо применить метод fso.DriveExists(dpath). Этот метод возвращает 0, если диск не существует, в противном случае — 1. Ниже приведен код функции driveinfo(dpath), которая возвращает массив всех характеристик диска, указанного в качестве строкового параметра:

```
function driveInfo(dpath) {                          //  информация  о диске
var  fso = ActiveXOb]ect("Scripting.FileSystemObject")
var disk =  fso.GetDriveName(dpath)             //  имя   (буква)  диска
var dinfo  =  new Array(7)
if   (fso.DriveExists(disk)){                      //  еспи диск существует
var  d  =  fso.GetDrive(disk)                      //объект с информацией о диске
dinfo[0]   =  d.DriveLetter                        // буква с диска
dinfo[l]   =  d.IsReady                                       // готовность диска
dinfo[2]   =  d.DriveType                          // тип диска
if   (d.IsReady){
dinfo[3] = d.VolumeName                            // имя диска
dinfo[4] = d.SerialNumber                          // серийный номер диска
dinfo[5] = d.TotalSize                                        //   полный объем в байтах
dinfo[6]  = d.FreeSpace                                       //   свободно в байтах
}
return  dinfo                               //возвращение массива характеристик диска
}
```

Некоторые характеристики диска получают только после проверки готовности диска (например, гибкий диск установлен в дисковод). Используется выражение var disk = fso.GetDriveName(dpath) на тот случай, когда параметр функции содержит не просто букву диска, а путь к папке или файлу.

Для тестирования функции driveinfo(dpath), используйте следующий код:

```
function  driveInfo(dpath){
//  код функции
}

WScript.Ecno(driveInfo("A"))
WScript.Ecno(driveInfo("C"))
WScript.Ecno(driveInfo("E"))
WScript.Ecno(driveInfo("D"))

WScript.Echo(driveInfo("C:\\Мои документы"))
var  x = driveInfo("C")
 WScript.Echo ("Свободно:   "  +  x[5])
```

Сохраните этот код в файле с расширением js и выполните его (дважды щелкнув на этом файле в Проводнике).

Функция driveTotallnfo(dpath), код которой приведен ниже, ничего не возвращает, а выводит диалоговое окно с характеристиками одного или всех дисков. В качестве строкового параметра можно указать диск. Однако если параметр не указан или пуст, выводится информация обо всех дисках.

```
function  driveTotalInfo(dpath){                    //  Информация  о дисках
var  fso =  new ActiveXObject("Scripting.FUeSystem0bject")
var  disk
if  (!dpath)                              //  если  нет  параметра,  то  все диски  disk = new Array("
A" ,"B","C","D" ,"E","F")
else
disk = new Array(fso.GetDriveName(dpath)) // одноэлементный массив

var s = " ", t, d
for (i  =  0;   i   <  disk.length;   i++){
if   (fso.DriveExists(disk[i])){                          //если диск не существует
d =  fso.GetDrive(disk[i])                         // ссылка на диск
switch  (d.DriveType) {                            /          / тип диска
case   0:    t = " - неизвестный";       break
case   1:   t = "  - съемный";             break
case   2:   t   = "   - несъемный";       break
case   3:   t   = "   - сетевой";            break
case   4:   t   = "   - CD-ROM";          break
case   5:   t  = "   - виртуальный";    break
}

s+= "Диск  "  + d.DriveLetter + ":"              // буква диска

if   (d.IsReady) {                                         // если диск готов

        s+= d.VolumeName + t                          // имя диска
        s+=  "\n  SN:  "  + d.SerialNumber    // серийный номер диска
        s+=  "\л Объем: "  + d.TotalSize       // полный объем в байтах
        s+=  "\n Свободно: "  + d.freeSpace// свободно в байтах
}else
        s+=  t+  "  не  готов"
```

```
s+=  "\n\n"
}
}
WScript.Echo(s)        // вывод строки с характеристиками
}
```

Чтобы использовать функцию driveTotalInfo(dpath) в сценарии, выполняемом браузером, необходимо лишь заменить в ней выражение WScript.Echo(s) на alert(s).

Задание

1. Воспользовавшись приведенными листингами программ получите информацию о всех дисках компьютера. Как и в предыдущей работе приведенные фрагменты кода содержат незначительные ошибки.
2. Выведите диалоговое окно о дисках C и F.

ЛАБОРАТОРНАЯ РАБОТА №3
Тема **Работа с папками**

Создание папки

Для создания папки можно использовать следующий код:

```
var  fso =  new AciiveXObject(“Scripting.FileSystemObject")
 fso.CreateFolder(folderpath)
```

Здесь folderpath — строка, содержащая полный путь к создаваемой папке, например "C:\\Мои документы\\моя папка". Обратите внимание, что при указании пути требуется использовать двойной слэш.

Для выполнения этих же действий с помощью WSH вместо первого выражения можно использовать:

```
var fso = WScript.Create0bject("Scripting.FileSystemObject")
```

Второе выражение, fso.CreateFolder(folderpath), создает указанную папку и возвращает ссылку на нее, если операция прошла успешно. В противном случае выводится диалоговое окно с сообщением об ошибке.

При создании папки необходимо, чтобы существовали все папки более высокого уровня, лежащие на пути к создаваемой папке, указанные в параметре folderpath и сам диск.

Нельзя создать папку с уже используемым именем. Таким образом, чтобы создать папку, предварительно следует выполнить целый ряд проверок.

В листинге приведен код функции createFolder(folderpath), которая выполняет все необходимые проверки. Более того, если какие-нибудь папки на пути к конечной (целевой) папке не существуют, то функция создаст их.

```
function createFolder(folderpath){
 /*  создание папки  Возвращает:  -1.  если папка  создана или  существует,   и  0 - в  противном случае  */

var fso = new ActiveXObject("Scripting.FileSystemObject")
var disk = fso.GetDriveName(folderpath)              // имя (буква) диска
/* Проверка характеристик диска: */

if (!fso.DriveExists(disk)) return 0                                // если диск не существует
if (!fso.GetDrive(disk). IsReady) return 0              // если диск не готов
```

```
// Если не подходит тип диска:
if (fso.GetDrive(disk).DriveType = = 0 || fso.GetDrive(disk).DriveType = = 4)
        return 0
if (fso.GetDrive(disk).FreeSpace < 1024)
        return 0                    // если мало места
if (fso.FolderExists(folderpath))
        return -1          //если  папка  уже существует, не  создаем ее
var  apath  =  folderpath.split("\\")               // преобразуем в массив имен папок
for (i =1;  i  <  apath. length;   i++)
{
        disk+=  "\\"  +  apath[i]
        if  (!fso.FolderExists(disk))   // если папка не существует, создаем ее
        fso.CreateFolder(disk)

}
return  fso.FolderExists(folderpath)
//  возвращает результат  проверки существования  созданной  папки
```

Дополнительно проверяется наличие свободного пространства на диске. Для создания папки диск должен иметь минимум 1 Кбайт свободного места. Можно задать и другую пороговую величину.

Копирование, перемещение и удаление папки

Для копирования, перемещения и удаления папки используются следующие методы объекта файловой системы.

CopyFolder(folderpathl, folderpath2 [, переписать]) — копирует папку, указанную в строке folderpathl, в папку, указанную в строке folderpath2; если третий необязательный параметр имеет значение true, то уже существующая папка folderpath2 с тем же именем переписывается.

MoveFolder(folderpathl, folderpath2) — перемещает папку, указанную в строке folderpathl, в папку, указанную в строке folderpath2.

DeleteFolder(folderpath [, force]) — удаляет папку, указанную в строке folderpath; если второй необязательный параметр имеет значение true, то удаляется и папка, предназначенная только для чтения.

Примеры

```
var  folderpathl  =  "C:\\Мои документы \\Test1"
var  folderpath2  =  "C:\\Test2"
var  fso=new ActiveXObject(“Scripting. FileSystemObject")       //  объект  FSO
/* Создаем папку C:\Test2\ Мои документы \\Test1  */
fso.CopyFolder(folderpathl, folderpath2)
/*  Удаляем  папку  C:\Test2  */
fso.DeleteFolder(folderpath2)
/*  Создаем папку  C:\Program  Files\  Мои документы\ Test1  */
fso.MoveFolder(folderpathl,   "C:\\Program  Files")
```

Как и в случае создания папки, при ее копировании, перемещении или удалении папки необходимо сначала убедиться в том, что это действительно можно сделать. Следующая функция выполняет ряд проверок и в случае положительного результата удаляет папку:

```
function deleteFolder(folderpath){                    //  удаление  папки
```

```
// Возвращает 0,   еспи  папка удалена  или не  существует,   и  -1 – иначе
var fso = new ActiveXObject("Scripting. FileSystemObject")
 if   (!fso.FolderExists(folderpath))   return  0      //  если  папка  не существует
var disk =  fso.GetDriveName(folderpath)              //  имя   (буква)  диска
//  Если  не  подходит  тип диска:
if (fso. GetDrive(disk). DriveType = = 0 || fso. GetDrive(disk). DriveType = = 4)
return -1
fso.DeleteFolder(folderpath) //удаление папки
return fso. FolderExists(folderpath)
//возвращает результат проверки существования созданной папки
}
```

Задание

Создайте функции для создания, копирования, перемещения и удаления папок.

ЛАБОРАТОРНАЯ РАБОТА №4
Тема **Работа с файлами**

Создание текстового файла

Для создания текстового файла на диске, следует выполнить следующие:

```
var  fso =  new ActiveXObject("Scripting.FileSystemObject")
fso.CreateTextFile(filepath)
```

Здесь filepath — строка, содержащая полный путь к создаваемому файлу, например "C:\\Мои документы\\testfile.txt". При указании пути требуется использовать двойной слэш. Выражение, fso.CreateTextFile(filepath), создает указанный файл, открывает с доступом для записи и возвращает ссылку на него, если операция прошла успешно. В противном случае выводится диалоговое окно с сообщением об ошибке. Обратите внимание, что созданный файл остается не доступным для записи.

Для выполнения с помощью WSH можно использовать и:

```
var  fso = WScript.CreateObject("Scripting.FileSystemObject")
```

Второй способ создания текстового файла основан на применении метода OpenTextFile (открыть текстовый файл) с параметром режима открытия ForWriting (для записи). Этот параметр имеет значение 2. Это делается следующим образом:

```
var  fso =  new ActiveXObject("Scripting.FileSystemObject")
var  f  =  fso.OpenTextFile(filepath,   2)
```

Новый текстовый файл, созданный описанными выше методами, ничего не содержит. Создаваемый или открываемый для чтения или записи текстовый файл совсем не обязательно должен иметь расширение txt. Он может иметь расширение htm, html, shtml, js, asp, prg или др. Т.е, он должен быть текстовым. При создании файла требуется убедиться в возможности это сделать, во избежание появления сообщений об ошибках. Так, если хотя бы одна из папок в пути к файлу не существует, то попытка применить метод CreateTextFile( ) приведет к ошибке. Ошибка также возникнет и в случае неготовности диска, отсутствия указанного дисковода, а также в случае, если это устройство для чтения компакт-дисков. В листинге приведен код функции createFile(filepath), выполняющей все необходимые проверки. Кроме того, создаются папки, указанные в filepath, но не существующие.

Листинг

```
function createFile(filepath){
//  Возвращает  ссылку  на  созданный файл или 0, если файл не создан
var  fso =  new ActiveXObject("Scripting. FileSystemObject")
var  i   = filepath. lastIndexOf("\\")
```

```
if  (i  >= 0)  file =  filepath.substr(i + 1)  //  выделяем имя файла из filepath
var  folder = filepath.slice(0, i)        //выделяем путь  к файлу без  имени файла

if  (!createFolder(folder))  return  0 //  проверка и создание недостающих  папок
if  (fso.FileExists(file)) // если файл существует,  то открываем его для  записи
return fso.OpenTextFile(filepath, 2)
return  fso.CreateTextFile(folder +  "\ \"  +  file)
// создаем файл и возвращаем ссылку на  него
}
```

Функция createFolder() создания папки производит все проверки и при необходимости создает недостающие папки. Если указанный в параметре filepath файл уже существует на диске, то он открывается для записи.

В рассмотренной выше функции createFile(filepath) для выделения имени файла из его полного имени filepath использовались встроенные функции JavaScript: lastIndexOf( ), substr( ) и slice( ). Однако вместо этого можно было использовать и специальные методы объекта файловой системы:

GetBaseName(filepath) – возвращает последний элемент в filepath без расширения;

GetExtensionName(filepath) – возвращает расширение последнего элемента в filepath.

Примеры

```
var  fso = new ActiveXObject("Scripting.FileSystemObject")
f so.GetBaseName("C: \\Мои документы\\testf i le. txt")          //  testfile
fso. GetExtensionName("C:\\Мои документы\\testfile.txt")          //  txt
```

Не менее полезным является и метод BuildPath(path, name), который к пути path дописывает элемент name:

```
var  fso =  new ActiveXObject("Scripting.FileSystemObject")
fso.BuildPath("C:\\  Мои документы",   "testfile.txt")
```

Если файл был создан, то он остается открытым. Чтобы закрыть файл, используется метод Close().

Примеры

```
// Создание и закрытие файла:
var fso = new ActiveXObject("Scripting.FileSystemObject")
var myfile = fso.CreateTextFile("C:\\Мои документы\\testfile.txt")
var myfile .Close( )
// Открытие и закрытие файла:
var fso = new ActiveXObject("Scripting.FileSystemObject")
var f = fso.OpenTextFile("C: \\Мои документы \\ testfile. txt ", 2)
var myfile.Close( )
// Создание/открытие и закрытие файла:
var myfile = createFi le( "C: \\Мои документы \\ testfile.txt ")
var myfile.Close( )
```

Копирование, перемещение и удаление файла

Для операций копирования, перемещения (переименования) и удаления файлов имеются методы объекта файловой системы (FSO) и методы объекта файла.

```
var  fso = new ActiveXObject("Scripting.FileSystemObject")  //  объект  FSO
var  file =  fso.GetFile(filepathl)                 //  объект файла
/*  Методы копирования  filepathl  в  filepath2  */
file.Copy(filepath2)
```

fso.CopyFile(filepathl, fiIepath2)
/* Методы перемещения filepathl в filepath2   */
file.Move(filepath2)
fso.MoveFile(filepathl, filepath2)
/* Методы удаления filepathl */
 file.Delete(filepathl)
 fso.DeleteFile(filepathl)

Так же как и в случае с папками, методы копирования и удаления имеют еще один необязательный параметр. Так, значение true этого параметра в методах копирования обеспечивает перезапись уже существующего файла с тем же именем, а в методах удаления — удаление файлов, предназначенных только для чтения. Перечисленные выше операции могут применяться к любым файлам, а не только к текстовым.

В следующем примере создается текстовый файл testfile.txt в папке C:\Мои документы, затем этот файл перемещается в папку C:\Windows\Temp и копируется в корневую папку на диске С. В заключение он удаляется из обеих папок.

var fso = new ActiveXObject("Scripting. FileSystemObject")
var fl = fso.CreateTextFile("C: \\Мои документы\\testfile.txt")
fl.Close( )   // закрываем файл
fl = fso.GetFileC'C: \\Мои документы\\testfile. txt")   //Получаем ссылку на файл   fl.Move("C: \\Windows\\Temp\\testfile. txt")
 // Перемещаем файл в папку C:\Windows\Temp
var f2=fso.GetFile("C:\\Windows\\Temp\\testfile.txt")   // получаем ссылку f2.Copy   ("C:\\testfile. txt")   // копируем файл в папку C:\
// Получаем ссылки на файлы:
fl = fso.GetFile("C:\\Windows\\Temp\Uestfile. txt")
f2 = fso.GetFile("C:\\testfile. txt")
fl.Delete( )   //   удаляем файл C:\Windows\Temp\testfile.txt f3.Delete()
// удаляем файл C:\testfile.txt

Копировать, перемещать или удалять можно только закрытые файлы. Поскольку после операции создания файла он остается открытым, использовался метод Close().

Прежде чем копировать и перемещать файлы, необходимо проверить, возможны ли данные операции. Так, следует проверить, существует и готов ли диск, достаточно ли свободного места на нем, существуют ли все папки, указанные в пути к файлу. Требуется также решить, что делать, если копируемый или перемещаемый файл уже создан в месте назначения.

Некоторые из этих проверок следует выполнять и перед удалением файла. Попробуйте в качестве упражнения написать код функции, выполняющий все эти операции. Для ее решения дополнительно потребуется информация о такой характеристике файла, как его объем.

Значение объема (размера) файла в байтах содержится в свойстве Size объекта файла. Например,

var fso = new ActiveXObject("Scripting.FileSystemObject")
var fl = fso.GetFile("C: \\autoexec.bat")   // ссылка на объект файла
var size = fl.Size   // объем файла
C:\autoexec.bat
WScript.Echo(size)   // вывод окна с сообщением об объеме файла

Значение свойства Size объекта файла нужно сравнить со значением свойства FreeSpace объекта диска, чтобы выяснить, достаточно ли места для записи файла.

Чтение данных из файла и запись данных в файл

Открытие текстового файла производится с помощью метода OpenTextFile объекта FileSystemObject либо с помощью метода OpenAsTextStream объекта файла. При этом файл может быть открыт в трех режимах: только для чтения (for reading only), для записи (for writing) и для добавления (for appending) данных. Режимы для записи и добавления данных не допускают чтения. В режиме добавления записываемые данные добавляются к уже существующим. В режиме записи старые данные теряются, а новые записываются, поэтому режим записи лучше называть режимом перезаписи. Открытие файла:

```
var fso = new ActiveXObject("Scripting.FileSystemObject")
var myfile = fso.OpenTextFile(filepath, mode)
```

либо

```
var  fso  =  new ActiveXObject("Scripting.FileSystemObject")
var  fileobj   =  fso.GetFile(filepath)
var myfile = fileobj.OpenAsTextStream(mode)
```

Здесь filepath — имя файла, возможно, с указанием пути к нему (например, "C:\\Мои документы\\testfile.txt"); mode — режим открытия файла:

1 — только для чтения (for read only);
2 — для записи (for writing);
8 — для добавления (for appending).

В результате создания нового текстового файла он остается открытым. Чтобы он был сразу же доступен для записи, необходимо передать методу CreateTextFile( ) второй параметр со значением true:

```
var fso =  new ActiveXObject("Scripting.FileSystemObject")
fso.CreateTextFile(filepath", true)
```

Для чтения данных из открытого текстового файла используются следующие методы объекта файла:

Read(количество_байтов) — применяется для чтения заданного количества байтов (символов), которое указывается в качестве параметра;

ReadLine( ) — применяется для чтения строки, при этом исключается символ перехода на новую строку;

ReadAll() — применяется для чтения всего содержимого текстового файла.

При использование методов Read(количество_байтов) или Readline() и желание пропустить заданное количество байтов или строку, можно использовать методы Skip(количество_байтов) и ScipLine( ) соответственно. Эти методы перемещения по файлу изменяют положение указателя, которое характеризуется значениями свойств Column (позиция в строке) и Line (номер строки) объекта файла. При первоначальном открытии файла эти свойства, доступные только для чтения, имеют значения 1. Каждое применение методов ReadLine( ) и SkipLine( ) увеличивает значение свойства Line на 1.

Пример

```
var  filepath =   "C : \\autoexec.bat"
var  fso =  new ActiveXObject("Scripting.FileSystemObject")
var myfile =  fso.OpenTextFile(filepath,   1)              //  объект файла
WScript.Echo(myfile.Line +  ",   "  + myfile.Column)   //   1,   1
myfile.SkipLine( )                                      //   пропустить  строку
WScript.Echo(myfile.Line +  ",   "  + myfile.Column)   //   2,   1
myfile.Skip(14)                                         //   пропустить  14 байтов
WScript.Echo(myfile . Line +  ",   "  + myfile.Column)  //   2,   15
```

Заметим, что применение метода ReadAll( ) после методов перемещения даст в результате содержимое файла, начиная с текущего положения указателя и до конца файла.

Для записи данных в открытый текстовый файл используются следующие методы объекта файла:

Write(строка) — применяется для записи строки символов без символа перехода на новую строку;

WriteLine(строка) — применяется для записи строки символов с добавлением символа перехода на новую строку;
WriteBlankLine(количество) — применяется для добавления пустых строк, количество которых указывается в качестве параметра; по существу, этот метод просто записывает заданное количество символов перехода на новую строку.

Для проведения экспериментов с файловыми операциями полезно выражения с методами чтения данных передать в качестве параметра методу отображения сообщений WScript.Echo( ).
Например, WScript.Echo(myfile.ReadLine( )).

Задание

Создать файл. Записать в него текст. Дополнить файл новым текстом. Открыв файл, читать из файла каждую третью строку.

ЛАБОРАТОРНАЯ РАБОТА № 5
Тема: **Создание ярлыков**

Ярлык (значок) представляет собой файл с расширением lnk, содержащий ссылку на некоторое приложение или документ, а также параметры его открытия в окне.

Аналогичный ссылочный файл с расширением url содержит URL-адрес документа (веб-страницы). Создать ярлык на рабочем столе компьютера, в меню Пуск, в папке Автозагрузка, Избранное или в любой другой папке можно с помощью объекта Wscript.Shell, входящего в состав WSH. Этот объект (точнее, его экземпляр) создается двумя способами, так же, как и FSO.

- Первый способ:

```
var Myshell = new ActiveXObject("WScript.Shell")
```

- Второй способ:

```
var  Myshell  = WScript.CreateObject ("WScript.Shell")
```

С помощью метода SpecialFolders( ) можно узнать местоположение специальных папок, таких как Рабочий стол, Автозагрузка, Избранное, Мои документы, Программы и др. За этими папками закреплены специальные идентификаторы. Например, папке Рабочий стол соответствует идентификатор Desktop, папке Мои документы — MyDocuments, папке Избранное — Favorites.

Список всех идентификаторов специальных папок: AllUsersDesktop, AllUsersStartMenu, AllUsersPrograms, AllUsersStartup, Desktop, Favorites, Fonts, MyDocuments, NetHood, PrintHood, Programs, Recent, SendTo, StartMenu, Startup, Templates.

Для получения местоположения папки, например - Главное меню, открываемой при щелчке на кнопке Пуск, следует выполнить следующие:

```
 var Myshell = new ActiveXObject ("WScript. Shell")
 var mypath = Myshell. SpecialFolders("StartMenu")
//  Значение:   C:\Windows\ Главное меню
```

В листинге приводится пример программы создания ярлыка для Блокнота Windows (notepad.exe) и расположения его на рабочем столе.

```
var  Myshell  =  new ActiveXObject ("WScript. Shell")
var mypath = Myshell. SpecialFolders("Desktop")    //  путь  к папке  Рабочий  стол
/*  Создание  ярлыка  и  подписи  к  нему:   */
var myshortcut  = Myshell. CreateShortcut(mypath +  "\\Мой  Блокнот.lnk")
/*  Папка  расположения Windows:   */
var mywindir  = Myshell. ExpandEnvironmentStrings ("%windir%")
 /*  Параметры ярлыка:   */
```

```
// расположение файла:
 myshortcut.TargetPath = Myshell. ExpandEnvironmentStrings (mywindir  +
"\\notepad.exe")
myshortcut.WorkingDirectory = Myshell.ExpandEnvironmentStrings(mypath)
  // рабочая папка
myshortcut.WindowStyle = 4                          // тип окна  (стандартное)
/* файл, содержащий графическое изображение ярлыка: */
myshortcut. IconLocation = Myshell. ExpandEnvironmentStrings (mywindir  +
"\\notepad.exe")
myshortcut.Save( )                                  // сохранить на диске
```

Здесь Myshell.ExpandEnvironmentStrings("%windir%") возвращает строку, содержащую значение переменной среды, в данном случае %windir%. По умолчанию файл notepad.exe находится в папке расположения операционной системы Windows, но папка не обязательно называется C:\Windows.
Свойство WindowStyle может при-ать три значения: 3 — развернуть окно на весь экран, 4 — стандартное окно, 7— свернуть в значок на панели задач.

В следующем примере создается ярлык в главном меню кнопки Пуск для некоторой программы afpwin.exe:

```
var Myshell  = new ActiveXObject ("WScript. Shell")
var mypath = Myshell.SpecialFolders("StartMenu")
var myshortcut  = Myshell.CreateShortcut(mypath +  "\\АФП.lnk")
myshortcut.TargetPath =
        Myshell.ExpandEnvironmentStrings ("C:\\AFP\\afpwin.exe")
myshortcut.WorkingDirectory =
        Myshell. ExpandEnvironmentStrings("C: \\AFP")
myshortcut.WindowStyle =3           // тип окна  (развернуть во весь экран)
myshortcut.IconLocation =
        Myshell.ExpandEnvironmentStrings(“C: \\AFP\\afp.ico")
myshortcut.Save( )
```

Приведенный ниже код создает в папке Избранное ссылку (url-файл) на главную страницу веб-сайта автора книги:

```
var Myshell = new ActiveXObject ("WScript.Shell")
 var mypath = Myshell.SpecialFolders ("Favorites")
var myshortcut  = Myshell.CreateShortcut (mypath  +
"\\Сам себе Web-дизайнер.url")
myshortcut.TargetPath = Myshell.ExpandEnvironmentStrings ("http://
www.admiral.ru/~dunaev")
myshortcut. Save( )
```

Основное отличие этого примера от предыдущих состоит в том, что свойству TargetPath (путь к цели) присваивается URL-адрес документа.

Запуск приложений

Для запуска приложений служит метод Run( ) объекта Wscript.Shell. Командная строка запуска приложения (обычно это просто полное имя файла программы) передается методу в качестве строкового параметра.

Примеры

```
var  MysheU  =  new ActiveXObject("WScript.Shell")
Myshell.Run("winword.exe C:\\My\\mydocument.dopc")
```

```
Myshell.Run("C:\\MyFolder\\myprogram.exe")
```

Задание

Создать ярлык для своей рабочей папки на рабочем столе компьютера, в меню Пуск, в папке Автозагрузка, Избранное и в любой другой папке.

Лабораторная работа №6
Методы организация таблиц идентификаторов

Выделение идентификаторов и других элементов исходной программы происходит на фазе лексического анализа. Для хранения найденных идентификаторов и их характеристик используются специальные хранилища данных, называемые *таблицами символов,* или *таблицами идентификаторов.*

Любая таблица идентификаторов состоит из набора полей, количество которых равно числу различных идентификаторов, найденных в исходной программе. Каждое поле содержит в себе полную информацию о данном элементе таблицы. Компилятор может работать как с одной, так и с несколькими таблицам идентификаторов.

Состав информации, хранимой в таблице идентификаторов для каждого элемента исходной программы, зависит от семантики входного языка и типа элемента. Так для переменных могут храниться: имя переменной, тип данных и область памяти, связанная с переменной; для функций – имя функции, количество и типы формальных аргументов функции, тип возвращаемого результата; адрес кода функции.

Не вся информация, хранимая в таблице идентификаторов, заполняется компилятором одновременно. Имена переменных могут быть выделены на фазе лексического анализа, типы данных для них – на фазе синтаксического разбора, а область памяти – на фазе подготовки к генерации кода. Таким образом, на разных фазах компиляции компилятор многократно обращается к таблице для поиска информации и записи новых данных.

Таблицы идентификаторов организуются таким образом, чтобы компилятор имел возможность максимально быстрого поиска требуемого ему элемента.

Простейший способ ее организации состоит в добавлении новых элементов в порядке их поступления. В этом случае таблица идентификаторов представляет собой неупорядоченный массив информации. Поиск нужного элемента заключается в последовательном сравнении искомого элемента с каждым элементом таблицы. Тогда для поиска в таблице, содержащей n элементов, в среднем будет выполнено n/2 сравнений. Такой способ организации таблиц идентификаторов является неэффективным.

Поиск более эффективен в таблице, элементы которой упорядочены (отсортированы) согласно некоторому порядку. Методом поиска в упорядоченном списке является *бинарный* (или *логарифмический*) *поиск.*

Его алгоритм состоит в следующем: искомый символ сравнивается с элементом в середине таблицы (с порядковым номером $(N + 1)/2$). Если этот элемент не является искомым, то просматривается только блок элементов, пронумерованных от 1 до $(N + 1)/2 – 1$, или блок элементов от $(N + 1)/2 + 1$ до N в зависимости от того, меньше или больше искомый элемент по сравнению с ранее найденным. Так продолжается до тех пор, пока либо искомый элемент не будет найден, либо алгоритм дойдет до очередного блока, содержащего один или два элемента, с которыми можно выполнить прямое сравнение искомого элемента.

Так как на каждом шаге число элементов, которые могут содержать искомый элемент, сокращается в 2 раза, максимальное число сравнений равно $1+\log_2(N)$.

Недостатком данного метода является требование упорядочивания элементов таблицы идентификаторов. Время упорядочивания напрямую зависит от числа элементов в массиве. Таблица идентификаторов зачастую просматривается компилятором еще до того, как она за-

полнена полностью, поэтому для построения такой таблицы можно пользоваться только алгоритмом прямого упорядоченного включения элементов.

Для сокращения времени поиска искомого элемента в таблице идентификаторов без значительного увеличения времени ее заполнения, надо отказаться от организации таблицы в виде непрерывного массива данных.

Например, существует метод построения таблиц в форме бинарного дерева. Каждый узел такого дерева представляет собой элемент таблицы, причем корневой узел является первым элементом, встреченным при заполнении таблицы. Дерево называется бинарным, так как каждая вершина в нем может иметь не более двух ветвей. Для определенности их называют «правая» и «левая».

Первый идентификатор помещается в вершину дерева. Все дальнейшие идентификаторы попадают в дерево по следующему алгоритму:

Шаг 1. Выбрать очередной идентификатор из входного потока данных. Если его нет – построение дерева закончено.

Шаг 2. Сделать текущим узлом дерева корневую вершину.

Шаг 3. Сравнить очередной идентификатор с идентификатором, содержащимся в текущем узле дерева.

Шаг 4. Если очередной идентификатор меньше, то перейти к шагу 5, если равен – сообщить об ошибке и прекратить выполнение алгоритма (двух одинаковых идентификаторов быть не должно!), иначе перейти к шагу 7.

Шаг 5. Если у текущего узла существует левая вершина, сделать ее текущим узлом и вернуться к шагу 3, иначе перейти к шагу 6.

Шаг 6. Создать новую вершину, поместить в нее очередной идентификатор, сделать эту новую вершину левой вершиной текущего узла и вернуться к шагу 1.

Шаг 7. Если у текущего узла существует правая вершина, сделать ее текущим узлом и вернуться к шагу 3, иначе перейти к шагу 8.

Шаг 8. Создать новую вершину, поместить в нее очередной идентификатор, сделать эту новую вершину правой вершиной текущего узла и вернуться к шагу 1.

Поиск нужного элемента в дереве выполняется по алгоритму, схожему с алгоритмом заполнения дерева:

Шаг 1. Сделать текущим узлом дерева корневую вершину.

Шаг 2. Сравнить искомый идентификатор с идентификатором, содержащимся в текущем узле дерева.

Шаг 3. Если идентификаторы совпадают, искомый идентификатор найден, алгоритм завершен, иначе надо перейти к шагу 4.

Шаг 4. Если очередной идентификатор меньше, перейти к шагу 5, иначе перейти к шагу 6.

Шаг 5. Если у текущего узла существует левая вершина, сделать ее текущим узлом и вернуться к шагу 2, иначе искомый идентификатор не найден, алгоритм завершен.

Шаг 6. Если у текущего узла существует правая вершина, то сделать ее текущим узлом и вернуться к шагу 2, иначе искомый идентификатор не найден, алгоритм завершен.

Для данного метода число требуемых сравнений и форма дерева зависят от порядка, в котором поступают идентификаторы. Недостатком метода является необходимость работы с динамическим выделением памяти при построении дерева.

В целом метод бинарного дерева является довольно удачным механизмом для организации таблиц идентификаторов. Он нашел свое применение в ряде компиляторов. Иногда компиляторы строят несколько различных деревьев для идентификаторов разных типов и разной длины.

Лучших результатов можно достичь, если применить методы, связанные с использованием хэш-функций и хэш-адресации.

Хэш-функцией F называется некоторое отображение множества входных элементов R на множество целых неотрицательных чисел Z: $F(r) = n$, $r \in R$, $n \in Z$. Множество допустимых вход-

ных элементов R называется областью определения хэш-функции. Множеством значений хэш-функции F называется подмножество M из множества целых неотрицательных чисел Z: $M \subseteq Z$, содержащее все возможные значения, возвращаемые функцией F: $\forall r \in R$: $F(r) \in M$ и $\forall m \in M$: $\exists r \in R$: $F(r) = m$. Процесс отображения области определения хэш-функции на множество значений называется «хэшированием».

При работе с таблицей идентификаторов хэш-функция выполняет отображение имен идентификаторов на множество целых неотрицательных чисел. Областью определения такой хэш-функции будет множество всех возможных имен идентификаторов.

Хэш-адресация заключается в использовании значения, возвращаемого хэш-функцией, в качестве адреса ячейки из некоторого массива данных. Тогда размер массива данных должен соответствовать области значений используемой хэш-функции. Следовательно, в реальном компиляторе область значений хэш-функции не должна превышать размер доступного адресного пространства компьютера.

Метод организации таблиц идентификаторов, основанный на использовании хэш-адресации, заключается в размещении каждого элемента таблицы в ячейке, адрес которой возвращает хэш-функция, вычисленная для этого элемента. Для помещения каждого элемента в таблицу идентификаторов требуется вычисление его хэш-функции и размещения его по вычисленному адресу. Первоначально таблица идентификаторов должна содержать пустые ячейки.

Для поиска требуемого элемента в таблице также вычисляется хэш-функция и проверяется содержимое соответствующей ячейки. Если она не пуста – элемент найден, иначе – не найден.

Время размещения элемента в таблице и время его поиска определяются только временем, затрачиваемым на вычисление хэш-функции, которое в общем случае несопоставимо меньше времени, необходимого для выполнения многократных сравнений элементов таблицы. Но метод имеет два очевидных недостатка. Первый из них неэффективное использование объема памяти под таблицу идентификаторов: размер массива для ее хранения должен соответствовать области значений хэш-функции, в то время как реально хранимых в таблице идентификаторов может быть существенно меньше. Второй – необходимость соответствующего разумного выбора хэш-функции.

Если двум или более идентификаторам соответствует одно и то же значение функции, такая ситуация называется *коллизией.* Хэш-функция, допускающая хотя бы единичную коллизию, не может быть напрямую использована для хэш-адресации в таблице идентификаторов.

Для полного исключения коллизий хэш-функция должна быть взаимно однозначной, т.е. каждому элементу из области определения хэш-функции должно соответствовать только одно значение из ее множества значений, и каждому значению из множества значений этой функции должен соответствовать только один элемент из области ее определения.

В реальности область значений любой хэш-функции ограничена размером доступного адресного пространства в данной архитектуре компьютера. Множество адресов любого компьютера с традиционной архитектурой может быть велико, но всегда конечно. Организовать взаимно однозначное отображение бесконечного множества имен идентификаторов на конечное множество невозможно.

Существует несколько способов для разрешения проблемы коллизии. Одним из них является метод *рехэширования* (расстановки). В нем, если для элемента A адрес h(A), вычисленный с помощью хэш-функции h, указывает на уже занятую ячейку, то необходимо вычислить новое значение $n_1 = h_1(A)$ и проверить занятость ячейки по адресу n_1. Если и она занята, то вычисляется значение $h_2(A)$ и так до тех пор, пока либо не будет найдена свободная ячейка, либо очередное значение $h_i(A)$ совпадет с h(A). В последнем случае считается, что таблица идентификаторов заполнена, и места в ней больше нет.

Тогда таблицу идентификаторов можно организовать по следующему алгоритму размещения элемента:

Шаг 1. Вычислить значение хэш-функции n = h(A) для нового элемента A.

Шаг 2. Если ячейка по адресу n пустая, поместить в нее элемент A и завершить алго-

ритм, иначе i = 1 и перейти к шагу 3.

Шаг 3. Вычислить $n_i = h_i$ (A). Если ячейка по адресу n_i пустая, то поместить в нее элемент A и завершить алгоритм, иначе перейти к шагу 4.

Шаг 4. Если $n = n_i$ то сообщить об ошибке и завершить алгоритм, иначе i = i+1 и вернуться к шагу 3.

Поиск элемента A в таблице идентификаторов, организованной таким образом, будет выполняться по следующему алгоритму:

Шаг 1. Вычислить значение хэш-функции n= h(A) для искомого элемента A.

Шаг 2. Если ячейка по адресу n пуста, то элемент не найден, алгоритм завершен. Иначе сравнить имя элемента в ячейке n с именем искомого элемента A. Если они совпадают – элемент найден и алгоритм завершен, иначе i = 1, перейти к шагу 3.

Шаг 3. Вычислить $n_i = h_i(A)$. Если ячейка по адресу n_i пустая или $n = n_i$ то элемент не найден и алгоритм завершен, иначе сравнить имя элемента в ячейке n_i с именем искомого элемента A. Если они совпадают, то элемент найден и алгоритм завершен, иначе i = i + 1 и повторить шаг 3.

Алгоритмы размещения и поиска элемента схожи по выполняемым операциям и имеют одинаковые оценки времени, необходимого для их выполнения.

При такой организации таблиц идентификаторов при возникновении коллизии алгоритм пытается поместить элемент в пустую ячейку, что может привести к возникновению новой, дополнительной коллизии. Поэтому количество операций, необходимых для поиска или размещения в таблице элемента, зависит от степени заполнения таблицы.

Важно определить хэш-функцию h_i для каждого i. Чаще всего функции h_i определяют как некоторые модификации первоначальной хэш-функции h. Например, самым простым методом вычисления функции $h_i(A)$ является ее организация в виде $h_i(A) = (h(A) + p_i) \bmod N_m$, где p_i – некоторое вычисляемое целое число, а N_m – максимальное значение из области значений хэш-функции h. Простейшим случаем будет задать $p_i = i$. Тогда при совпадении значений хэш-функции для каких-либо элементов поиск свободной ячейки в таблице начинается последовательно от текущей позиции, заданной значением хэш-функции h(A). В этом случае при совпадении хэш-адресов элементы в таблице начинают группироваться вокруг них, что увеличивает число необходимых сравнений при их поиске и размещении.

Но даже такой примитивный метод рехэширования является достаточно эффективным средством организации таблиц идентификаторов при частном заполнении таблицы. Имея, например, заполненную на 90% таблицу для 1024 идентификаторов, в среднем необходимо выполнить 5.5 сравнений для поиска одного идентификатора, в то время как даже логарифмический поиск дает в среднем от 9 до 10 сравнений.

Лучшие результаты дает использование в качестве p_i последовательности псевдослучайных целых чисел p1, p_2, ..., p_k или при вычислении по формуле $h_i(A) = (h(A)*i) \bmod N_m$, если N_m – простое число. В целом, рехэширование позволяет добиться неплохих результатов, но требование частичного заполнения таблицы ведет к неэффективному использованию объема доступной памяти.

Частичное заполнение таблицы идентификаторов при применении хэш-функций ведет к неэффективному использованию всего объема памяти, доступного компилятору. Этого недостатка можно избежать, дополнив таблицу идентификаторов специальной промежуточной хэш-таблицей. В ее ячейках может храниться либо пустое значение, либо значение указателя на некоторую область памяти из основной таблицы идентификаторов. Тогда после вычисления значения хэш-функции определяется адрес, по которому происходит обращение сначала к промежуточной хэш-таблице, а через нее – к самой таблице идентификаторов. Тогда иметь в самой таблице идентификаторов ячейку для каждого возможного значения хэш-функции не обязательно и таблицу можно сделать динамической. Количество ячеек в ней будет равно числу идентификаторов. Пустые ячейки будут только в хэш-таблице. Способ реализации такой схемы называется «метод цепочек». Он работает по следующему алгоритму:

Шаг 1. Во все ячейки хэш-таблицы поместить пустое значение, таблица иденти-

фикаторов пуста, переменная FreePtr (указатель первой свободной ячейки) указывает на начало таблицы идентификаторов; i = 1.

Шаг 2. Вычислить значение хэш-функции n_i для нового элемента A_i. Если ячейка хэш-таблицы по адресу n_i пустая, поместить в нее значение переменной FreePtr и перейти к шагу 5; иначе перейти к шагу 3.

Шаг 3. Положить j=1, выбрать из хэш-таблицы адрес ячейки таблицы идентификаторов m_j и перейти к шагу 4.

Шаг 4. Для ячейки таблицы идентификаторов по адресу m_j проверить значение поля ссылки. Если оно пустое, записать в него адрес из переменной FreePtr и перейти к шагу 5; иначе j = j + 1, выбрать из поля ссылки адрес m_j и повторить шаг 4.

Шаг 5. Добавить в таблицу идентификаторов новую ячейку, записать в нее информацию для элемента A_i (поле ссылки должно быть пустым), в переменную FreePtr поместить адрес, следующий за добавленной ячейкой. Если больше нет идентификаторов для размещения в таблице, алгоритм завершен, иначе i = i + 1 и перейти к шагу 2.

Поиск элемента в таблице идентификаторов, организованной таким образом, будет выполняться по следующему алгоритму:

Шаг 1. Вычислить значение хэш-функции n для искомого элемента A. Если ячейка хэш-таблицы по адресу n пустая, то элемент не найден и алгоритм завершен, иначе j = 1, выбрать из хэш-таблицы адрес ячейки таблицы идентификаторов m_j.

Шаг 2. Сравнить имя элемента в ячейке таблицы идентификаторов по адресу m_j с именем искомого элемента A. Если они совпадают, искомый элемент найден и алгоритм завершен, иначе перейти к шагу 3.

Шаг 3. Проверить значение поля ссылки в ячейке таблицы идентификаторов по адресу m_j. Если оно пустое, то искомый элемент не найден и алгоритм завершен; иначе j = j + 1, выбрать из поля ссылки адрес m_j и перейти к шагу 2.

В случае возникновения коллизии алгоритм размещает элементы в ячейках таблицы, связывая их друг с другом последовательно через поле ссылки. При этом элементы не могут попадать в ячейки с адресами, которые потом будут совпадать со значениями хэш-функции. Таким образом, дополнительные коллизии не возникают. В итоге в таблице возникают своеобразные цепочки связанных элементов, откуда происходит и название данного метода.

В реальных компиляторах практически всегда, так или иначе, используется хэш-адресация. Часто применяются комбинированные методы. В этом случае, как и для метода цепочек, в таблице идентификаторов организуется специальное дополнительное поле ссылки. Но в отличие от метода цепочек оно имеет несколько иное значение. При отсутствии коллизий для выборки информации из таблицы используется хэш-функция, поле ссылки остается пустым. Если же возникает коллизия, го с помощью поля ссылки организуется поиск идентификаторов, для которых значения хэш-функции совпадают, по одному из рассмотренных выше методов: неупорядоченный список, упорядоченный список или же бинарное дерево. При хорошо построенной хэш-функции коллизии будут возникать редко.

Хэш-адресация – метод, который применяется не только для организации таблиц идентификаторов в компиляторах, но и нашел свое применение в операционных системах, и в системах управления базами данных.

Контрольные вопросы

1. Какая информация хранится в таблице идентификаторов?
2. Какие способы организации таблиц идентификаторов существуют?
3. Когда и по какой причине возникает коллизия при организации таблиц идентификаторов с использованием хэш-функции?
4. Каковы требования к списку идентификаторов при использовании метода логарифмического поиска в таблице идентификаторов?
5. В чем заключается преимущество метода цепочек по сравнению с методом рехэширования?

6. Выберите неверное утверждение:

а) область значений любой хэш-функции ограничена размером доступного адресного пространства в данной архитектуре компьютера;

б) вся информация, хранимая в таблице идентификаторов, заполняется компилятором одновременно;

в) для полного исключения коллизий хэш-функция должна быть взаимно однозначной;

г) состав информации, хранимой в таблице идентификаторов для каждого элемента исходной программы, зависит от семантики входного языка и типа элемента.

7. Использование значения, возвращаемого хэш-функцией, в качестве адреса ячейки из некоторого массива данных называется

а) хэш-адресацией б) хэш-функцией

в) хэшированием г) рехэшированием

Задание

1. Написать программу, реализующую создание таблицы идентификаторов по методу логарифмического поиска. В качестве исходных данных использовать файл, считая в нем все исходные слова идентификаторами.

2. Написать программу, реализующую метод бинарного дерева для построения таблицы идентификаторов. Для организации дерева использовать динамический массив. В качестве исходных данных использовать файл, считая в нем все исходные слова идентификаторами.

3. Написать программу, реализующую создание таблицы идентификаторов на основе метода рехэширования. В качестве исходных данных использовать файл, считая в нем все исходные слова идентификаторами. В качестве хэш-функции использовать:

а) коды первых двух букв идентификаторов;

б) коды последних двух букв идентификаторов;

в) некоторое случайное число в диапазоне от -10 до 10;

г) номер текущего вычисления хэш-функции.

4. Написать программу, реализующую создание таблицы идентификаторов методом цепочек с хэш-функцией из предыдущего задания.

5. Сравнить реализованные методы построения таблиц идентификаторов.

Лабораторная работа №7, 8
Конечные автоматы

Распознавателем языка называется программа, которая, получая на вход цепочку символов входного алфавита, принимает ее, если она представляет собой предложение языка и не принимает иначе.

Теория автоматов лежит в основе теории построения компиляторов. Под автоматом понимают не реально существующее устройство, а некоторую математическую модель, свойства и поведение которой можно изучать.

Конечный автомат является простейшим из моделей теории автоматов и служит управляющим устройством для всех остальных автоматов.

Преимуществами конечного автомата являются следующие факты:

моделирование конечного автомата требует фиксированного объема памяти;

существует ряд теорем и алгоритмов, позволяющих конструировать и упрощать конечные автоматы;

обработка одного входного символа требует небольшого числа операций, что обеспечивает быстроту работы.

Конечный автомат может решать простые задачи компиляции (в частности, лексический блок почти всегда строится на его основе).

На вход конечного автомата подается цепочка символов из конечного множества, назы-

ваемого входным алфавитом. Как допускаемые, так и отвергаемые автоматом цепочки состоят из символов входного алфавита. Символы, не принадлежащие входному алфавиту, нельзя подавать на вход автомата.

Работа конечного автомата представляет последовательность шагов (тактов). На каждом шаге автомат находится в одном из своих состояний. Одно из этих состояний называется начальным. На каждом следующем шаге автомат может либо остаться в текущем состоянии, либо перейти в другое. То, в какое состояние перейдет автомат, определяет функция переходов. Она зависит не только от текущего состояния, но и от символа на входе автомата. Если функция переходов допускает несколько состояний, то автомат может перейти в любое из них.

Некоторые состояния выбираются в качестве допускающих (или заключительных) состояний. Если автомат, начав работу в начальном состоянии, при прочтении всей входной цепочки переходит в одно из допускающих состояний, говорят, что входная цепочка допускается автоматом. Если последнее состояние автомата не является допускающим – цепочка отвергнута.

Таким образом, конечный автомат задается пятеркой вида M(Q, V, δ, q_0, F), где

Q – конечное множество состояний автомата,

V – алфавит входных символов,

δ – функция переходов, $\delta(a,q)=R$, $a \in V$, $q \in Q$, $R \subseteq Q$,

q_0 – начальное состояние автомата ($q_0 \in Q$),

F – непустое множество конечных состояний автомата.

Конфигурация автомата на каждом шаге работы определяется тройкой (q,ω,n), где q – текущее состояние автомата, ω – цепочка входных символов, n – положение указателя во входной цепочке. Тогда начальная конфигурация автомата $(q_0,\omega,0)$.

Языком автомата называют множество всех цепочек, которые принимаются этим автоматом. Два конечных автомата эквивалентны, если они задают один и тот же язык.

Конечный автомат часто представляют в виде диаграммы или графа переходов автоматов. Граф переходов конечного автомата – это направленный граф, вершины которого помечены символами состояний конечного автомата, а дуга, помечена некоторым символом, если определена соответствующая функция перехода δ. Начальное и конечное состояние автомата помечаются специальным образом.

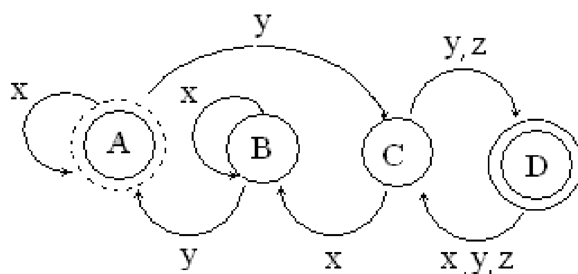


Рис. 5. Граф перехода конечного автомата

На рис. 5 задан конечный автомат *M({ A, B, C, D}, {x, y, z}, δ, A,, {D});*
δ: δ(A, x)=A, δ(A, y)=C), δ(B, x)=B, δ(B, y)=A, δ(C, x)=B, δ(C, y)=D, δ(C, z)=D, δ(D, x)=C, δ(D, y)=C, δ(D, z)=C

Конечный автомат называют полностью определенным, если в каждом его состоянии существует функция перехода для всех возможных входных символов.

Для моделирования работы конечного автомата его удобно привести к полностью определенному виду, чтобы исключить ситуации, из которых нет переходов по входным символам. Для этого добавляют еще одно состояние, которое условно называют «ошибка» – Е. На него замыкают все неопределенные переходы, в том числе и само на себя.

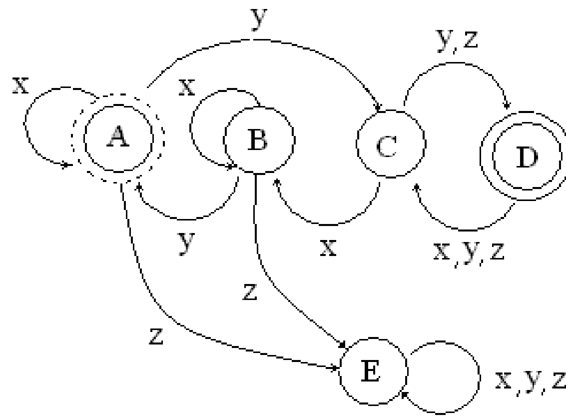


Рис. 6. Граф переходов полностью определенного конечного автомата

Другой способ представления конечного автомата – таблица переходов, в которой информация размещается в соответствии со следующими соглашениями:

столбцы помечены входными символами,

строки помечены символами состояний,

элементами таблицы являются символы новых состояний, соответствующих входным символам столбцов и состояниям строк,

первая строка помечена символом начального состояния,

строки, соответствующие допускающим (заключительным) состояниям помечены справа единицами, а строки, соответствующие отвергающим состояниям, помечены нулями.

Таблица переходов полностью определенного конечного автомата, представленного на рис. 6 задается таблицей:

	x	*y*	*z*	
A	A	C	E	0
B	B	A	E	0
C	B	D	D	1
D	C	C	C	0
E	E	E	E	0

Конечный автомат называют детерминированным конечным автоматом, если в каждом из его состояний для любого входного символа функция перехода содержит не более одного состояния. В противном случае автомат называют недетерминированным. Доказано, что для любого конечного автомата можно построить эквивалентный ему детерминированный конечный автомат. Моделировать работу детерминированного конечного автомата существенно проще, поэтому всегда стремятся сделать это преобразование. При построении компиляторов чаще всего используют полностью определенный детерминированный конечный автомат.

Конечные автоматы являются распознавателями для регулярных языков. Поскольку язык констант и идентификаторов является регулярным, то для реализации лексических анализаторов служат регулярные грамматики и конечные автоматы.

Среди всех регулярных грамматик выделяют отдельный класс – автоматные грамматики. Они также могут быть леволинейными и праволинейными.

Разница между автоматными и обычными регулярными грамматиками заключается в том, что где в правилах обычных регулярных грамматик может присутствовать цепочка терминальных символов, в автоматных грамматиках может присутствовать только один терминальный символ. Любая автоматная грамматика является регулярной, но не наоборот – не всякая регулярная грамматика является автоматной.

Доказано, что классы обычных регулярных грамматик и автоматных грамматик почти эквивалентны.

Существует алгоритм, который позволяет преобразовать произвольную регулярную

грамматику к автоматному виду – то есть построить эквивалентную ей автоматную грамматику:

Шаг 1. Все нетерминальные символы из множества **VN** исходной грамматики *G* переносятся во множество **VN'** грамматики G'.

Шаг 2. Необходимо просматривать все множество правил P грамматики G. Если встречаются правила вида $A \to Ba_1$, $A, B \in$ **VN**, $a_1 \in$ **VT** или вида $A \to a_1$, $A \in$ **VN**, $a \in$ **VT**, то они переносятся во множество P' правил грамматики G^1 без изменений.

Если встречаются правила вида $A \to Ba_1, a_2, ..., a_n, n > 1, A, B \in$ **VN**, $\forall n > i > 0: a_i \in$ **VT**, то во множество нетерминальных символов **VN'** грамматики G' добавляются символы $A_1, A_2, ..., A_{n-1}$, а во множество правил P' грамматики G' добавляются правила:

$A \to A_{n-1} a_n$

$A_{n-1} \to A_{n-2} a_{n-1}$

...

$A_2 \to A_1 a_2$

$A_1 \to Ba_1$

Если встречаются правила вида $A \text{ -> } a_1, a_2, ..., a_n, n > 1, A, B \in$ **VN**, $\forall n > I > 0: a_i \in$ **VT**, то во множество нетерминальных символов **VN'** грамматики G' добавляются символы $A_1, A_2, ..., A_{n-1}$, а во множество правил P′ грамматики G' добавляются правила:

$A \to A_{n-1} a_n$

$A_{n-1} \to A_{n-2} a_{n-1}$

...

$A_2 \to A_1 a_2$

$A_1 \to a_1$

Если встречаются правила вида $A \to B$ или вида $A \to \lambda$, то они переносятся во множество правил P' грамматики G' без изменений.

Шаг 3. Просматривается множество правил P' грамматики G' с целью поиска правил вида $A \to B$ или вида $A \to \lambda$.

Если находится правило первого вида, то просматривается множество правил P’ грамматики G'. Если в нем присутствуют правила вида $B \to C$, $B \to Ca$, $B \to a$ или $B \to \lambda$, то в него добавляются правила вида $A \to C$, $A \to Ca$, $A \to a$ и $A \to \lambda$ соответственно, $\forall A, B, C \in$ **VN'**, $\forall a \in$ **VT'** (при этом учитывается, что в грамматике не должно быть совпадающих правил). Правило $A \to B$ удаляется из множества правил P'.

Если находится правило вида $A \to \lambda$ (и символ A не является целевым символом S), то просматривается множество правил P' грамматики G'. Если в нем присутствуют правила вида $B \to A$ или $B \to Aa$, то в него добавляются правил: вида $B \to \lambda$ и $B \to a$ соответственно, $\forall A, B, C \in$ **VN'**, $\forall a \in$ **VT'** (при этом также учитывается, что в грамматике не должно быть совпадающих правил). Правило $A \to \lambda$ удаляется из множества правил P'.

Шаг 4. Если на шаге 3 было найдено хотя бы одно правило вида $A \to B$ или $A \to \lambda$ во множестве правил P' грамматики G', то надо повторить шаг 3, иначе перейти к шагу 5.

Шаг 5. Целевым символом *S′* грамматики *G′* становится символ *S*.

Шаги 3 и 4 алгоритма можно не выполнять, если грамматика не содержит правил вида A → B (такие правила называются цепными) или вида $A \to \lambda$ (такие правила называются λ - правилами). Реальные регулярные грамматики обычно не содержат правил такого вида.

В алгоритме рассмотрен случай леволинейной грамматики. Для праволинейной легко построить аналогичный алгоритм.

Регулярные языки являются удобным типом языков. Для них разрешимы многие проблемы: эквивалентности двух языков, принадлежности языку заданной цепочки символов, пустоты языка.

Эти проблемы разрешимы вне зависимости от того, каким из трех способов задан язык:

4) регулярными грамматиками (праволинейным или леволинейными);
5) конечным автоматом;
6) регулярным множеством.

Все три способа равноправны. Существуют алгоритмы, которые позволяют для регулярного

языка, заданного одним из способов, построить другой способ задания того же самого языка.

Регулярные множества – это множества цепочек символов над заданным алфавитом, построенные с использованием операций объединения, конкатенации и итерации.

Контрольные вопросы

1. Может ли граф переходов конечного автомата использоваться для однозначного определения автомата? Почему?
2. От каких параметров зависит функция переходов конечного автомата?
3. В каком случае конечный автомат называется полностью определенным?
4. Всегда ли недетерминированный конечный автомат может быть приведен к детерминированному?
5. Сколькими параметрами определяется конфигурация конечного автомата?
6. Распознавателями какого типа языков являются конечные автоматы?

Задание

1. Построить конечный автомат для следующих видов цепочек, состоящих из нулей и единиц:

а) между вхождениями единиц четное число нулей;
б) за каждым вхождением пары единиц следует нуль;
в) каждый пятый символ единица;
г) все цепочки начинаются на нуль и оканчиваются единицей;
д) в цепочке перед каждой единицей стоит нуль;
е) цепочка должна содержать ровно три единицы.

Для реализации конечного состояния построить граф или таблицу переходов, составить программу на языке высокого уровня.

2. Построить конечный автомат, распознающий зарезервированные слова языка С++:
а) inline, int, if
б) continue, class, const
в) private, protected, public
г) union, using, union
д) double, delete, default
е) virtual, void, volatile
3. Построить регулярное выражение для представления десятичных чисел, описания переменных в алгоритмических языках.
4. Описать словами множество цепочек, распознаваемых каждым из конечных автоматов, заданных таблицами переходов:

а)

	0	*1*	
A	B	C	0
B	D	B	1
C	C	D	1
D	D	D	0

б)

	0	*1*	
A	B	A	0
B	D	C	0
C	C	D	1
D	D	D	0

в)

	x	*y*	*z*	
A	A	C	C	0
B	C	D	C	0
C	C	C	C	0
D	C	C	A	1

Лабораторная работа №9
Преобразования конечного автомата

Алгоритм преобразования произвольного конечного автомата M(Q, V, δ, q_0, F) к эквивалентному ему, детерминированному конечному автомату M′(Q′, V, δ′, $q_0′$, F′), заключается в следующем:

Шаг 1. Множество состояний Q′ автомата M′ строится комбинацией всех состояний множества Q автомата M. Их возможное число $2^n – 1$, где n – количество состояний.

Шаг 2. Функция переходов δ′ автомата M′ строится как $\delta'(a, [q_1, q_2, ..., q_m]) = [r_1, r_2, ..., r_k]$, где $\forall\ 0< i \le m\ \exists\ 0< j \le k$ такое, что $\delta(a, q_i) = r_j$.

Шаг 3. Обозначим $q'_0 = [q_0]$.

Шаг 4. Если $f_1, f_2, ..., f_l$ $(l > 0)$ – конечные состояния автомата M ($f_i \in F$), тогда множество конечных состояний F′ автомата M′ строится из всех состояний имеющих вид [..., f_i, ...].

Затем требуется из полученного автомата удалить недостижимые символы по следующему алгоритму:

Шаг 1. Обозначим множество достижимых состояний R, $R=\{q_0\}$, а множество текущих активных состояний на каждом шаге алгоритма P_i. i=0, $P_0=\{q_0\}$.

Шаг 2. $P_{i+1}=\varnothing$.

Шаг 3. $\forall\ a \in V,\ \forall\ q \in P_i\ \ P_{i+1}=P_i \cup \delta(a, q)$

Шаг 4. Если $P_{i+1} - R = \varnothing$ алгоритм завершен, иначе $R=R \cup P_{i+1}$, i = i + 1, перейти к шагу 3.

После этого можно исключить все состояния, не вошедшие во множество R.

Пример.

Преобразовать конечный автомат M ({H, A, B, S}, {x, y}, δ, H, {S}), δ(H, y)=B, δ(B, x)=A, δ(A, y)={B, S}. Граф переходов такого автомата изображен на рис. 7.

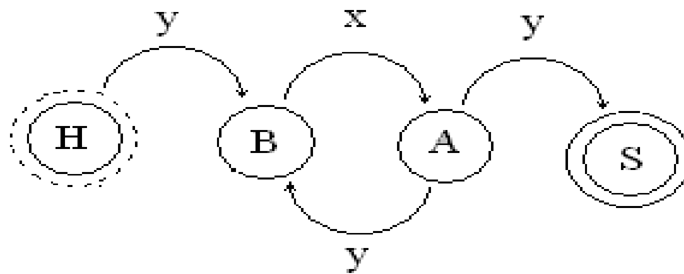


Рис. Граф переходов недетерминированного конечного автомата

Данный автомат недетерминированный, поскольку из состояния A возможны два различных перехода по символу y.

Шаг 1. Построим множество состояний эквивалентного автомата Q′ = {[H], [A], [B], [S], [HA], [HB], [HS], [AB], [AS], [BS], [HAB], [HAS], [HBS], [ABS], [HABS]}.

Шаг 2. Функция переходов эквивалентного конечного автомата

δ′ ([H], y)=[B]

δ′ ([A], y)=[BS]

δ′ ([B], x)=[A]

δ′ ([HA], y)=[BS]

δ′ ([HB], x)=[A]

δ' ([HS], y)=[B]
δ' ([AB], x)=[A]
δ'([AB], y)=[BS]
δ' ([AS], y)=[BS]
δ' ([BS], x)=[A]
δ' ([HAB], x)=[A]
δ' ([HAS], y)=[BS]
δ' ([HBS], y)=[B]
δ'([HBS], x)=[A]
δ' ([ABS], y)=[BS]
δ' ([ABS], x)=[A]
δ' ([HABS], x)=[A]
δ' ([HABS], y)=[BS]

Шаг 3. Начальное состояние M′ q'_0=[H]

Шаг 4. Множество конечных состояний эквивалентного детерминированного конечного состояния F′ = {[S], [HS], [AS], [BS], [HAS], [HBS], [ABS], [HABS]}

Исключим недостижимые состояния, в итоге получаем M′({H, B, A, S}, {x, y}, δ', H, {S}), δ'(H, y)=B, δ'(B, x)=A, δ'(A, y)=S, δ'(S, x)=A. Граф переходов этого автомата приведен на рис.

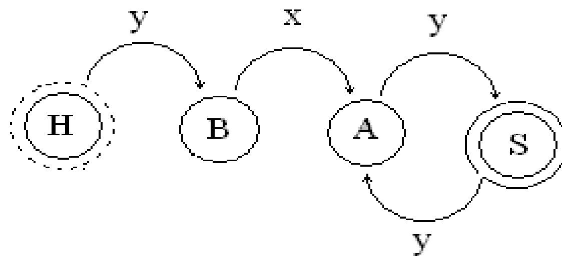


Рис. Граф переходов детерминированного конечного автомата

Моделировать работу детерминированного конечного автомата существенно проще, чем произвольного конечного автомата. Но при выполнении преобразований число состояний автомата может значительно вырасти. Тогда затраты на моделирование возрастают, и преобразование не оправдывается.

Многие конечные автоматы можно минимизировать. Минимизация конечного автомата заключается в построении эквивалентного конечного автомата с меньшим числом состояний.

Для минимизации автомата используется алгоритм построения эквивалентных состояний конечного автомата. Два различных состояния q и q’ в конечном автомате M(Q, V, δ, q_0, F) называются n-эквивалентными (n-неразличимыми) $n \geq 0$, если, находясь в одном из этих состояний и получив на вход любую цепочку символов, автомат может перейти в одно и то же множество конечных состояний. Очевидно, что 0-эквивалентными состояниями автомата M(Q, V, δ, q_0, F) являются два множества его состояний F и F–Q.

Множества эквивалентных состояний автомата называют классами эквивалентности, а всю совокупность – множеством классов эквивалентности R(n), причем R(0) = {F , F–Q}.

Алгоритм минимизации конечного автомата заключается в следующем:

Шаг 1. Из автомата исключаются все недостижимые состояния.

Шаг 2. Строятся классы эквивалентности автомата.

Шаг 3. Классы эквивалентности состояний исходного конечного автомата становятся состояниями результирующего минимизированного конечного автомата.

Шаг 4. Функции переходов результирующего конечного автомата очевидным образом строится на основе функции переходов исходного конечного автомата.

Для этого алгоритма доказано: во-первых, что он строит минимизированный конечный автомат, эквивалентный заданному конечному автомату; во-вторых, что он строит конечный

автомат с минимально возможным числом состояний (минимальный конечный автомат).

Если два состояния q_1 и q_2 одного автомата эквивалентны, то автомат можно упростить, заменяя в графе переходов (таблице переходов) все вхождения имен этих состояний каким-нибудь новым именем, а затем, удаляя двух строк, соответствующих q_1 и q_2. Например, состояния K и L конечного автомата, заданного таблицей переходов, представленной на рис. 9, явно имеют одинаковые функции, так как оба являются допускающими, оба переходят в состояние B при чтении входного символа *a* и оба переходят в состояние C при чтении *b*.

	a	*b*	
A	A	K	0
B	C	L	1
C	L	A	0
K	B	C	1
L	B	C	1

Рис. Таблица переходов не минимизированного автомата

Поэтому можно объединить состояния K и L в одно состояние и назвать его X. Получается упрощенная таблица состояний, представленная на рис.

	a	*b*	
A	A	4	0
B	C	X	1
C	X	A	0
X	B	C	1

Рис. Таблица переходов минимизированного автомата

Обычно эквивалентность состояний менее очевидна. Два состояния эквивалентны тогда и только тогда, когда не существует различающей их цепочки. Метод проверки эквивалентности состояний основывается на следующем. Состояния q_1 и q_2 эквивалентны тогда и только тогда, когда выполняются два условия:

1) условие подобия – состояния q_1 и q_2 должны быть либо оба допускающие, либо оба отвергающие;

2) условие преемственности – для всех входных символов состояния q_1 и q_2 должны переходить в эквивалентные состояния, т.е. их преемники эквивалентны.

Эти условия выполняются тогда и только тогда, когда q_1 и q_2 не имеют различающей цепочки. Если нарушено одно из условий, существует цепочка, различающая эти два состояния. А если не выполняется условие подобия, различающей является пустая цепочка. Если нарушено условие преемственности, то некоторый входной символ x переводит из состояния q_1 и q_2 в неэквивалентные. Поэтому x с приписанной к нему цепочкой, различающей эти новые состояния, образует цепочку, различающую q_1 и q_2.

Рассмотренные условия можно использовать в общем методе проверки на эквивалентность произвольной пары состояний. Для этого строятся таблицы эквивалентности состояний. Рассмотрим конечный автомат, таблица переходов которого изображена на рис. 11. Выявим его эквивалентные состояния.

	y	*z*	
F	F	K	0
G	H	M	0
H	H	P	0
K	N	P	0
L	G	N	1
M	N	M	0
N	N	K	1
P	N	K	0

Рис. Таблица переходов конечного автомата до минимизации

Сначала проверяем на эквивалентность состояния F и P. Таблица эквивалентности состояний содержит по одному столбцу для каждого входного символа, для *y* и *z*. Следующие строки будут добавляться в ходе проверки. Первоначально такая таблица имеет вид, представленный на рис.: имеется одна строка, которая помечена парой состояний, подвергаемых проверке, т.е. F, P.

Условие подобия для этих строк выполняется, так как оба состояния являются отвергающими.

	y	*z*
F, P		

Рис. Первоначальный вид таблицы эквивалентности состояний

Для проверки условия преемственности результат действия на отдельную пару состояний каждого входного символа запишем в соответствующую ячейку таблицы эквивалентности. Так как состояние F, P под действием входного символа *y* переходит в состояние F и N соответственно, то они записываются в столбец таблицы, соответствующий символу *y*. Так как оба состояния F и P переводятся символом *z* в состояние K, соответственно K помещается в столбец для z. Таблица эквивалентности символов F, P примет вид, представленный на рис:.

	y	*z*
F, P	F, N	K

Рис. Строка таблицы эквивалентности состояний F, P

Чтобы нарушалось условие преемственности, должны быть неэквивалентны либо состояние F и N, либо состояния K и K. Так как каждое состояние эквивалентно само себе, состояния K и K эквивалентны автоматически.

Для исследования на эквивалентность состояния F и N, к таблице эквивалентности состояний добавляется новая строка, помеченная новой парой F, N. Для нее повторяется весь процесс, описанный для пары F, P. Условие подобия для этой пары не выполняется, так как N – допускающее, а F – отвергающее состояние. Следовательно, состояния F и N неэквивалентны. Таблицу эквивалентности состояний можно использовать для построения различающей цепочки. Строка F, N появилась как результат применения входного символа у к паре F, P, поэтому *y* является различающей цепочкой.

Строим таблицу эквивалентности пары F, G. Эти состояния подобны, поэтому требуется вычислить результат применения к ним каждого входного символа, полученные состояния размещаются в таблице. Надежды на эквивалентность пары F, G оправдаются, если будет установлена эквивалентность пар, помещенных в таблицу – F, H и K, M. В таблицу добавляется строка для каждой из этих пар.

	y	*z*
F, G	F, H	K, M

	y	*z*
F, G	F, H	K, M
F, H		
K, M		

Рис. Таблица эквивалентности состояний F, G

Результаты промежуточного заполнения таблицы представлены на рис. Один из двух новых элементов дает новую строку, а именно пара K, P. Другой элемент уже имеется в таблице

– проверять его не следует. Далее осуществляется проверка следующей по списку пары: K, M. Состояния этой пары подобны. Вычисляем пары N, N и M, P. Так как состояние N эквивалентно самому себе, единственной новой строкой в таблице будет M, P.

	y	z
F, G	F, H	K, M
F, H	F, H	K, P
K, M		

	y	z
F, G	F, H	K, M
F, H	F, H	K, P
K, M	N	M, P
K, P		
M, P		

Рис. Заполнение таблица эквивалентности состояний F, G

Описанные процедуры повторяются для оставшихся пар – K, P и M, P. По их окончании не удается получить ни одной новой пары неподобных состояний и ни одной пары, которую надо проверять на подобие. Окончательная таблица представлена на рис. 16.

	y	z
F, G	F, H	K, M
F, H	F, H	K, P
K, M	N	M, P
K, P	N	K, P
M, P	N	K, M

Рис. Итоговая таблица эквивалентности состояний F, G

Различающие цепочки для состояний F, G отсутствуют, поэтому эти состояния являются эквивалентными.

Алгоритм проверки эквивалентности двух состояний можно описать следующим образом.

Шаг 1. Начать построение таблицы эквивалентности состояний с отведения столбца для каждого входного символа. Пометить первую строку парой проверяемых состояний.

Шаг 2. Выбрать в таблице эквивалентности состояний строку, ячейки которой еще не заполнены, и проверить, подобны ли состояния, которыми она помечена. Если они не подобны, то два исходных состояния неэквивалентны. Перейти к шагу 3. Иначе вычислить результат применения каждого входного символа к этой паре состояний и записать полученные пары состояний в соответствующие ячейки рассматриваемой строки.

Шаг 3. Если элементом таблицы является пара одинаковых состояний или пара состояний, которые уже использовались как метки строк – дополнительные действия не требуются. Если же элемент таблицы – пара различных состояний, до этого не используемая как метка, добавляется новая строка. Порядок состояний в паре не важен, и пары q_1, q_2 и q_2, q_1 считаются одинаковыми.

Шаг 4. Если все строки таблицы эквивалентности заполнены, исходная пара состояний и все пары, порожденные в ходе проверки, эквивалентны, проверка закончена. Если же таблица не заполнена, нужно обработать еще, по крайней мере, одну ее строку и применить шаг 2.

Так как каждая пара, появившаяся в заполненной таблице, содержит эквивалентные состояния, этот метод проверки дает обычно больше информации, чем предполагалось сначала. Из итоговой таблицы эквивалентности (рис. 16) следует, что, кроме эквивалентности пары (F, G), которая подвергалась проверке, доказана эквивалентность пар (F, H), (K, M), (K, P), (M, P).

По свойству транзитивности из эквивалентности пар состояний F, H и F, G и следует эквивалентность пары G, H. Таким образом, состояния F, G, H эквивалентны друг другу. Аналогично, эквивалентны друг другу состояния K, M, P.

Автомат можно упростить, объединив состояния F, G, H в состояние A, а K, M, P – в состояние B. Новые имена подставляются в таблицу переходов, лишние строки удаляются и получается более простой – эквивалентный автомат

	y	*z*	
A	A	B	0
B	N	B	0
L	A	N	1
N	N	B	1

Рис. Таблица переходов минимального конечного автомата

Чтобы упростить автомат необходимо, также удалить из него состояния, недостижимые из начального состояния ни для какой входной цепочки.

Минимизация конечных автоматов позволяет уменьшить количество их состояний, что в дальнейшем упрощает функционирование распознавателя.

Контрольные вопросы

1.В чем заключается алгоритм преобразования конечного автомата к детерминированному виду?

2.В каких случаях преобразовывать конечный автомат к детерминированному виду нецелесообразно?

3.Какие два состояния автомата называются эквивалентными?

4.Что собой представляет таблица эквивалентных состояний? Каким образом она заполняется?

5.Как определяется недостижимое состояние?

6.Какое из преобразований приводит к уменьшению количества состояний конечного автомата, а какое к их уменьшению?

Задание

1. Найти различающую цепочку для пары автоматов:

	a	*b*	
A	A	B	1
B	C	D	0
C	D	A	1
D	A	B	0

	a	*b*	
A	A	D	1
B	A	D	0
C	B	A	1
D	C	B	0

2. Найти минимальную эквивалентную таблицу для каждого из ниже расположенных автоматов.

	0	*1*	
S1	S1	S3	0
S2	S7	S4	1
S3	S6	S5	0
S4	S1	S4	1
S5	S1	S4	0
S6	S7	S6	1
S7	S7	S3	0

	X	*Y*	
1	4	1	1
2	5	1	1
3	4	5	0
4	2	6	0
5	1	7	0
6	1	4	1
7	2	5	1

3. Для автоматов из предыдущего задания найти недостижимые состояния.

4. Найти недостижимые состояния автомата, представленного таблицей состояний

	0	1	2	
A	C	E	G	0
B	J	E	G	1
C	J	A	H	0
D	F	A	G	1
E	E	J	H	0
F	D	I	A	1
G	H	A	J	0
H	G	J	B	1
I	D	F	G	0
J	B	H	G	1

Лабораторная работа №10

Автоматы с магазинной память

Синтаксический анализатор (синтаксический разбор) – часть компилятора, которая отвечает за выявление и проверку синтаксических конструкций входного языка. В задачи синтаксического анализа входят:

поиск и выделение синтаксических конструкций в тексте исходной программы;

установка типа и проверка правильности каждой из найденных синтаксических конструкций;

представление синтаксических конструкций в виде, удобном для дальнейшей генерации текста результирующей программы.

Синтаксический анализатор является основной частью компилятора на этапе анализа, поскольку без него работа компилятора бессмысленна. В то время как лексический разбор является необязательной фазой. Синтаксический анализатор воспринимает выход лексического анализатора, если он имеется в наличие, либо сам распознает их. На практике не существует жесткого правила, определяющего, какие конструкции должны распознаваться на лексическом уровне, а какие – на синтаксическом. Обычно это определяется разработчиком компилятора, исходя из технологических аспектов программирования, а также синтаксиса и семантики входного языка.

В основе синтаксического анализатора лежит распознаватель текста программы, построенный на основе грамматики входного языка. Как правило, синтаксические конструкции языков программирования могут быть описаны с помощью контекстно-свободных грамматик, реже регулярных грамматик.

Чаще всего регулярные грамматики применимы к языкам ассемблера, а синтаксис языков высокого уровня основан на грамматике КС-языков.

Распознавателями для КС-языков являются автоматы с магазинной памятью (МП-автоматы). Это односторонние недетерминированные распознаватели с линейно ограниченной магазинной памятью.

В общем виде МП-автомат можно определить как R (Q, V, Z, δ , qo, Zo, F), где Q – множество состояний автомата;

V – алфавит входных символов автомата;

Z – специальный конечный алфавит магазинных символов автомата;

δ – функция переходов автомата;

$q_o \in Q$ – начальное состояние автомата;

$z_o \in Z$ – начальный символ магазина;

$F \in Q$ – множество конечных состояний.

МП-автомат в отличие от обычного КА имеет стек (магазин), в который можно поме-

щать специальные (магазинные) символы. Обычно это терминальные и нетерминальные символы грамматики языка. Переход МП-автомата из одного состояния в другое зависит не только от входного символа, но и от символа на верхушке стека. Таким образом, конфигурация автомата определяется тремя параметрами: состоянием автомата, текущим символом входной цепочки (положением указателя в цепочке) и содержимым стека.

МП-автомат условно можно представить в виде схемы, показанной на рис.

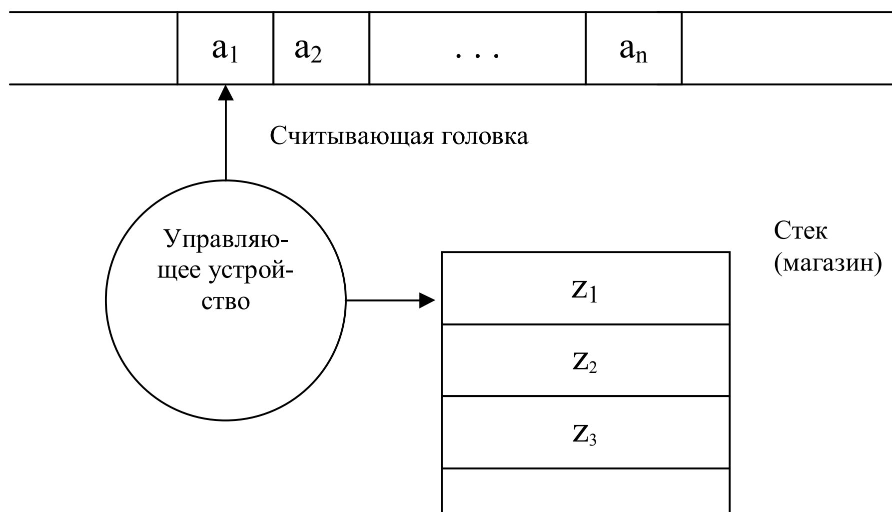


Рис. Схема работы МП-автомата

Каждый шаг процесса обработки задается множеством правил, использующих информацию трех видов: состояние, верхний символ магазина, текущий входной символ.

Это множество правил называется управляющим устройством, или механизмом управления.

В зависимости от получаемой информации управляющее устройство выбирает либо выход из процесса (т. е. прекращает обработку), либо переход в новое состояние. Переход состоит из трех операций: над магазином, над состоянием и над входом. Возможные операции над магазином могут быть следующими:

1) втолкнуть в магазин определенный магазинный символ;
2) вытолкнуть верхний символ магазина;
3) оставить магазин без изменений.

Операция над состоянием единственна – перейти в заданное новое состояние.

Возможные операции над входом:

1) перейти к следующему входному символу и сделать его текущим входным символом;
2) оставить данный входной символ текущим, иначе говоря, держать его до следующего шага.

Обработку входной цепочки МП-автомат начинает в некотором выделенном состоянии при определенном содержимом магазина, а текущим входным символом является первый символ входной цепочки. Затем автомат выполняет операции, задаваемые его управляющим устройством. Если происходит выход из процесса, обработка прекращается; если происходит переход, то он дает новый верхний магазинный символ, новый текущий символ – автомат переходит в новое состояние, и управляющее устройство определяет новое действие, которое нужно произвести.

Чтобы управляющие правила имели смысл, автомат не должен требовать следующего входного символа, если текущим символом является концевой маркер, и не должен выталкивать символ из магазина, если это маркер дна. Поскольку маркер дна может находиться исключительно на дне магазина, автомат не должен также вталкивать его в магазин.

При выполнении такта (перехода) из стека удаляется верхний символ, соответствующий условию перехода, и добавляется цепочка, соответствующая правилу перехода. Первый символ цепочки становится верхушкой стека.

МП-автомат допускает (принимает) цепочку символов, если, получив эту цепочку ход, он может перейти в одну из конечных конфигураций – когда при окончании цепочки автомат находится в одном из конечных состояний, а стек содержит некоторую определенную цепочку. Иначе цепочка символов не принимается.

МП-автомат допускает цепочку символов с опустошением магазина, если при окончании разбора цепочки автомат находится в одном из конечных состояний, а стек пуст.

Кроме обычного МП-автомата существует понятия расширенного и детерминированного МП-автомата.

Расширенный МП-автомат может заменять цепочку символов конечной длины в верхней части стека на другую цепочку символов конечной длины.
В отличие от обычного МП-автомата, который на каждом такте работы может изымать из стека только один символ, расширенный МП-автомат может изымать за один такт цепочку символов, находящуюся на вершине стека.

МП-автомат называется *детерминированным,* если из каждой его конфигурации возможно не более одного перехода в следующую конфигурацию. В противном случае МП-автомат называется недетерминированным

Стандартным представлением МП-автомата с одним состоянием является таблица со столбцами для входных символов и строками для символов магазина.

Контрольные вопросы

1. Каковы задачи, решаемые на этапе синтаксического анализа?
2. Если из каждой конфигурации автомата с магазинной памятью возможно не более одного перехода в следующую конфигурацию, то он называется

   а) расширенным б) детерминированным
   в) недетерминированным г) обычным
3. Автоматы с магазинной памятью служат для распознавания цепочек языков на основе

   а) контекстно-свободной грамматики
   б) грамматики с фразовой структурой
   в) контексно-зависимой грамматики
   г) регулярной грамматики
4. Каковы отличия автомата с магазинной памятью от конечного автомата? Какой из них является более интеллектуальным?
5. Объяснить понятие «управляющее устройство».
6. В каких случаях входная цепочка допускается МП-автоматом?

Задание

1. Построить МП-распознаватель для каждого из следующих множеств цепочек:

a) $\{1^n 0^m \mid n > m > 0\}$;
b) $\{1^n 0^m \mid n \geq m > 0\}$;
c) $\{1^n 0^n 1^m 0^m \mid n, m \geq 0\}$;
d) $\{1^n 0^m 1^n 0^m \mid n, m \geq 0\}$;
e) $\{1^n 0^m \mid m > n > 0\}$.

2. Для каждого из множеств первого задания указать цепочку длины, большей трех. Показать последовательность конфигураций соответствующих автоматов, построенных в первом задании при распознавании каждой из цепочек.

3. Написать три цепочки, принадлежащие множеству, распознаваемому МП-автоматом с одним состоянием, представленным следующей таблицей:

	a	b	c	⊣
A	ВТОЛКНУТЬ (A) СДВИГ	ОТВЕРГНУТЬ	ВЫТОЛКНУТЬ СДВИГ	ОТВЕРГНУТЬ
B	ВТОЛКНУТЬ (C) СДВИГ	ВЫТОЛКНУТЬ ДЕРЖАТЬ	ВТОЛКНУТЬ (A) СДВИГ	ОТВЕРГНУТЬ
C	ВТОЛКНУТЬ (B) ДЕРЖАТЬ	ВТОЛКНУТЬ (C) СДВИГ	ВЫТОЛКНУТЬ СДВИГ	ОТВЕРГНУТЬ
Магазинный символ	ВТОЛКНУТЬ (A) СДВИГ	ОТВЕРГНУТЬ	ОТВЕРГНУТЬ	ДОПУСТИТЬ

Для каждой из этих цепочек указать соответствующие последовательности конфигураций, допускающие эти цепочки.

4. Привести пример такого множество цепочек, которое может распознать МП-автомат с магазинным алфавитом, содержащим маркер дна магазина и еще два символа, но не может распознать никакой МП-автомат, у которого магазинный алфавит состоит из маркера дна магазина и еще одного символа.
5. Составить три допускаемые цепочки и соответствующие последовательности конфигураций для МП-автомата с одним состоянием, представленного на рис.

	(	)	⊣
A	ВТОЛКНУТЬ (A) СДВИГ	ВЫТОЛКНУТЬ СДВИГ	ОТВЕРГНУТЬ
Магазинный символ	ВТОЛКНУТЬ (A) СДВИГ	ОТВЕРГНУТЬ	ДОПУСТИТЬ

Лабораторная работа №11
Администрирование Linux

Задание

Зарегистрироваться в графической подсистеме пользователем user и запустить программу konsole. Из графической консоли переключиться на пользователя root.

Просмотреть список пользователей в файле /etc/passwd

Просмотреть с помощью программы getent подробные сведения об учетной записи пользователей root и user

Из графической подсистемы запустить программу Change Password

Изменить текущему пользователю user пароль.

Переключиться с помощью клавиш ctrl-alt-f1 в текстовый режим и зарегистрироваться пользователем root

Посмотреть конфигурацию первичного загрузчика lilo

Изменить уровень загрузки на 3 runlevel.
Просмотреть список всех установленных служб
Вывести список активных процессов системы
Исследовать в текстовом редакторе основной системный журнал
Настроить принтер
Создать новый раздел диска
Установить в него файловую систему ext3
Установить имя компьютера,
Настроить сетевой интерфейс и клиентскую часть NFS, DNS.4.2.6. Работа в Windows, Linux

Лабораторная работа №12
Команды Linux: мониторинг работы и просмотр логов

Команды Linux необходимые для мониторинга работы операционной системы. Все показания выводятся на экран в реальном времени. Число, стоящее после команды означает интервал между выводом информации.

top

Информация в реальном времени о загруженных процессах, потребление ОЗУ;

htop

Более расширенная on line-статистика о загруженных процессах (разработчик http://htop.sourceforge.net);

dmesg

Показывает log-файл загрузки ОС и нахождения новых устройств;

mpstat 1

Показывает расширенную статистику потребления ресурсов системы в процентах (для некоторых дистрибутивов необходима установка пакета sysstat);

vmstat 2

Показать расширенную статистику по использованию виртуальной памяти;

iostat 2

Показать расширенную статистику прерываний по устройствам;

6. Команды Linux: информация об устройствах.

Наверх

lsdev

информация об уже установленных устройствах (в некоторых дистрибутивах требует доставить пакет procinfo);

cat /proc/cpuinfo

Показать полную информацию о модели процессора (частота, поддерживаемые инструкции и т.д.);

cat /proc/meminfo

Показать расширенную информацию о занимаемой оперативной памяти (MemTotal, MemFree, Buffers, Cached, SwapCached,HighTotal, HighFree, LowTotal и т. д.);

grep SwapTotal /proc/meminfo

Показать размер раздела выделенного под swap;

watch -n1 'cat /proc/interrupts'

Показать информацию о прерываниях;

free -m

Информация о используемой и свободной ОЗУ и Swap-файле (-m указывает, что отображать нужно в Мб);

lshal

Показать список всех устройств и их параметров;

cat /proc/devices
Показать все устройства в системе (названия взяты из директории /proc/devices);
lspci -tv
Показать обнаруженные PCI-устройства;
lsusb -tv
Показать обнаруженные USB-устройства;
[sudo] dmidecode
Показать информацию о версии BIOS компьютера;
gtf 1024 768 75
Выводит строку ModeLine для Вашего монитора на параметрах экрана 1024x768x75Hz;

Лабораторная работа №13
Команды Linux: жесткие диски и файловая система

1 Информация о файловой системе и жестком диске
fdisk -l
Информация о всех подключенных жестких и сменных дисках;
[sudo] hdparm -I /dev/sda
Полная информация о IDE/ATA жестких дисках;
smartctl -a /dev/sda1
Выводит SMART-информацию о разделе жесткого диска /dev/sda1 (необходима установка пакета smartmontools);
[sudo] blkid
Выводит UUID всех доступных накопителей информации в системе;

2 Производительность жесткого диска
[sudo] hdparm -tT /dev/sda
Показывает производительность жесткого диска;

3 Монтирование разделов жесткого диска
mount | column -t
Показывает полную информацию о смонтированных устройствах;
cat /proc/partitions
Показывает только смонтированные разделы жесткого диска;
df
Показывает свободное место на разделах;
[sudo] mount /dev/sda1 /mnt
Монтирует раздел /dev/sda1 к точке монтирования /mnt;
[sudo] mount -t auto /dev/cdrom /mnt/cdrom
Монтирует большинство CD-ROM`ов;
[sudo] mount /dev/hdc -t iso9660 -r /cdrom
Монтирует IDE CD-ROM;
[sudo] mount /dev/scd0 -t iso9660 -r /cdrom
Монтирует SCSI CD-ROM;
[sudo] mount -t ufs -o ufstype=ufs2,ro /dev/sda3 /mnt
Монтирование FreeBSD разделов в Linux;
[sudo] mount -t smbfs -o username=vasja,password=pupkin //pup/Video
Монтирование сетевых ресурсов (SMB);
[sudo] mount -t iso9660 -o loop /home/file.iso /home/iso
Монтирование ISO-образов;
[sudo] mount /dev/sdb1 -t vfat -o rw /mnt
Монтирование раздел с файловой системой FAT 16/32 (к примеру USB-накопитель) к

точки монтирования /mnt с возможностью записи;
[sudo] umount /mnt
Отмонтирует раздел от точки монтирования /mnt;

Лабораторная работа №14
Команды Linux: настройка сети

1 Конфигурация сети
ifconfig
Показать параметры всех сетевых;
ifconfig eth0
Показать параметры сетевого интерфейса eth0;
[sudo] ethtool eth0
Показывает состояние сетевого интерфейса eth0 (для некоторых дистрибутивов требуется установка пакета ethtool). Команда ethtool применяется только для проводных подключений, не работает с беспроводными интерфейсами;
[sudo] ethtool -s eth0 speed 100 duplex full autoneg off
Принудительно задать скорость сетевому интерфейсу 100Mbit и режим Full duplex и отключить автоматическое определение;
ifconfig eth0 192.168.50.254 netmask 255.255.255.0
Задать основной IP адрес сетевому интерфейсу eth0;
ip addr add 192.168.50.254/24 dev eth0
Задать основной IP адрес сетевому интерфейсу eth0;
ifconfig eth0:0 192.168.51.254 netmask 255.255.255.0
Задать дополнительный IP адрес сетевому интерфейсу eth0;
ip addr add 192.168.51.254/24 dev eth0 label eth0:1
Задать дополнительный IP адрес сетевому интерфейсу eth0;
[sudo] ifconfig eth0 up
Запустить сетевой интерфейс eth0;
[sudo] ifconfig eth0 down
Отключить сетевой интерфейс eth0;
ifconfig eth0 hw ether 00:01:02:03:04:05
Смена MAC адреса;
[sudo] /etc/init.d/dhcpd restart
Перезагрузка DHCP клиента;
ping 192.168.0.2
Проверка сетевого соединения. Пингуется IP адрес 192.168.0.2 (пинговать можно ya.ru);

2 Маршрутизация
route -n
Выводит на экран таблицу маршрутизации;
netstat -rn
Выводит на экран таблицу маршрутизации;

3 Управление портами (брандмауэр)
netstat -an | grep LISTEN
Показывает список всех открытых портов;
lsof -i
Показывает список всех открытых портов в сеть Internet;
[sudo] netstat -tup
Активные соединения с интернетом;

socklist
Показывает все открытые сокеты;
[sudo] netstat -anp --udp --tcp | grep LISTEN
Список приложений, которые открывают порты;
[sudo] iptables -L -n -v
Показывает статус firewall (статус iptables);
[sudo] iptables -P INPUT ACCEPT
Открывает доступ ко всем портам;
[sudo] iptables -P FORWARD ACCEPT
Открывает доступ ко всем портам;
[sudo] iptables -P OUTPUT ACCEPT
Открывает доступ ко всем портам;
[sudo] iptables -X
Удаляет все цепочки;

Лабораторная работа №15
Команды Linux: пользователи и группы и работа с процессами

1.Команды Linux необходимые для работы с пользователями и группами пользователей.
id
Показывает сводную информацию по текущему пользователю (логин, UID, GID, группы);
finger Mut@NT
Показать информацию о пользователе Mut@NT;
last
Показывает последних зарегистрированных пользователей;
who
Показывает имя текущего пользователя и время входа;
useradd Mut@NT
Добавление нового пользователя Mut@NT;
groupadd ITShaman
Добавление группы ITShaman;
usermod -a -G ITShaman Mut@NT
Добавляет пользователя Mut@NT в группу ITShaman (для Debian-подобных дистрибутивов);
groupmod -A Mut@NT ITShaman
Добавляет пользователя Mut@NT в группу ITShaman (SuSE);
userdel Mut@NT
Удаление пользователя Mut@NT;
groupdel ITShaman
Удаление группы ITShaman;
Все запущенные процессы имеют уникальные номера - PID.
ps axjf
Показать все загруженные процессы;
pgrep -l sshd
Показать PID определенного процесса – sshd;
echo $$
Показать PID вашей оболочки;
fuser -va 22/tcp
Показать PID процесса использующий порт 22;
fuser -va /home
Показывает PID процесса имеющего доступ к /home;

lsof /home
Показывает список процессы, которые используют /home;
killall 0 httpd
Выводит на экран текущее состояние процесса httpd;
kil 4712
«Убить» процесс с PID 4712;
[sudo] killall TERM 4712
Посылает процессу с PID`ом 4712 сигнал TERM - завершить процесс;
[sudo] killall HUP httpd
Посылает процессу с именем httpd сигнал HUP - остановить процесс;
[sudo] fuser -k -TERM -m /home
“Убить” все процессы имеющие доступ к /home;

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО САМОСТОЯТЕЛЬНОЙ РАБОТЕ СТУДЕНТОВ

Самостоятельная работа – планируемая учебная, учебно-исследовательская, научно-исследовательская работа студентов, выполняемая во внеаудиторное время по заданию и при методическом руководстве преподавателя, но без его непосредственного участия (при частичном непосредственном участии преподавателя, оставляющем ведущую роль за работой студентов).

Самостоятельная работа является важным видом учебной и научной деятельности студента и играет значительную роль в рейтинговой технологии обучения. Государственным стандартом предусматривается, как правило, 50% часов из общей трудоемкости дисциплины на самостоятельную работу студентов. Количество часов самостоятельной работы и ее распределение между дидактическими единицами приведено в рабочей программе.

Задачами самостоятельной работы являются:

систематизация и закрепление полученных теоретических знаний и практических умений студентов;

углубление и расширение теоретических знаний;

формирование умений использовать нормативную, правовую, справочную документацию и специальную литературу;

развитие познавательных способностей и активности студентов: творческой инициативы, самостоятельности, ответственности и организованности;

формирование самостоятельности мышления, способностей к саморазвитию, самосовершенствованию и самореализации;

развитие исследовательских умений;

использование собранного и полученного в ходе самостоятельных занятий для эффективной подготовки к итоговым зачетам и экзаменам.

В процессе самостоятельной работы студент приобретает навыки самоорганизации, самоконтроля, самоуправления, становится активным самостоятельным субъектом учебной деятельности.

Выполняя самостоятельную работу под контролем преподавателя, студент должен:

– освоить минимум содержания, выносимый на самостоятельную работу студентов и предложенный преподавателем в соответствии с Государственными образовательными стандартами высшего профессионального образования по данной дисциплине.

– планировать самостоятельную работу в соответствии с графиком самостоятельной работы, предложенным преподавателем.

– самостоятельную работу студент должен осуществлять в организационных формах, предусмотренных учебным планом и рабочей программой преподавателя.

– выполнять самостоятельную работу и отчитываться по ее результатам в соответствии с графиком представления результатов, видами и сроками отчетности по самостоятельной работе студентов.

Основной формой самостоятельной работы студента является изучение конспекта лекций, их дополнение, рекомендованной литературы, подготовка отчетов к лабораторным работам.

КОНТРОЛЬ ЗНАНИЙ

Текущий контроль знаний предполагает защиту лабораторных работ и выполнение тестовых заданий. В тестовые задания входят вопросы по лекционному материалу и вопросы самостоятельной работы, приведенные в рабочей программе.

СОДЕРЖАНИЕ

Рабочая программа	3
Краткое изложение программного материала	10
Методические указания по изучению дисциплины	23
Методические указания к лабораторным работам	23
Технология выполнения лабораторных работ	24
Методические рекомендации по организации самостоятельной работы студентов	60
Контроль знаний	61