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 направлению подготовки

231000.68 – Программная инженерия

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 231000.68 – Программная инженерия
(указывается название специальности (направления подготовки)

от «____» _____ 201__ г. № _____

Председатель УМСС _____ /_____ /
(подпись) (И.О. Фамилия)

1 РАБОЧАЯ ПРОГРАММА

## 1. ЦЕЛИ И ЗАДАЧИ ДИСЦИПЛИНЫ

Целями освоения дисциплины «Системное программное обеспечение» является 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

## 2. МЕСТО ДИСЦИПЛИНЫ В СТРУКТУРЕ ООП ВПО

Код УЦ ООП М.2.В.ОД.3. – Профессиональный цикл, вариативная часть.

Для изучения дисциплины «Системное программное обеспечение» студент должен обладать стартовыми навыками создания программ на языке высокого уровня, уметь анализировать и обобщать информацию, желательно обладать аналитическим складом мышления, что могло быть получено в результате изучения дисциплин «Информатика», «Программирование», в объеме основной образовательной программы данного направления, работать с современным программным обеспечением.

## 3. КОМПЕТЕНЦИИ ОБУЧАЮЩЕГОСЯ, ФОРМИРУЕМЫЕ В РЕЗУЛЬТАТЕ ОСВОЕНИЯ ДИСЦИПЛИНЫ (МОДУЛЯ)

В результате освоения дисциплины обучающийся должен демонстрировать следующие результаты образования:

***Знать*** методы планирования и управления ресурсами жизненного цикла программного обеспечения;

***Уметь*** осуществлять выбор технической и экономической моделей эволюции и сопровождения программного продукта;

***Владеть*** навыками управления версиями и релизами программного продукта, навыками поддержки целостности конфигурации в течении жизненного цикла программного продукта.

Обучение студентов данной дисциплине должно способствовать развитию следующих профессиональных компетенций:

способность к проектной деятельности в профессиональной сфере на основе системного подхода, умения строить и использовать модели для описания и прогнозирования различных явлений, осуществлять их качественный и количественный анализ (ПК-6);

умение формировать технические задания и способность руководить разработкой программного обеспечения (ПК-7).

## 4. СТРУКТУРА И СОДЕРЖАНИЕ ДИСЦИПЛИНЫ (МОДУЛЯ)

Общая трудоемкость дисциплины составляет 4 зачетных единицы, 144 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)<br>Форма промежуточной аттестации (по семестрам) |
|---|---|---|---|---|---|---|---|---|
| | | | | практич. | лаб. | сам. работа | экзамен | |
| 1 | Трансляторы и компиляторы | 3 | 1 – 2 | 2 | | 10 | | Отчеты по лаб. раб. |
| 2 | Формальные языки и | 3 | 3 – 6 | 4 | 4 | 10 | | Отчет по лаб. раб. |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | грамматики | | | | | | | |
| 3 | Методы построения таблиц идентификаторов | 3 | 7 – 8 | 2 | 4 | 10 | | Отчеты по лаб. раб.<br>Тест №1 |
| 4 | Лексические и синтаксические анализаторы | 3 | 9 – 10 | 2 | 8 | 10 | | Отчеты по лаб. раб. |
| 5 | Генерация и оптимизация кода | 3 | 11 – 12 | 2 | | 12 | | Отчеты по лаб. раб.<br>Тест №2 |
| 6 | Алгоритмы управления памятью в операционных системах | 3 | 13 – 18 | 6 | 2 | 20 | | Отчеты по лаб. раб. |
| | ИТОГО | | | 18 | 18 | 72 | 36 | |

## 5. СОДЕРЖАНИЕ РАЗДЕЛОВ И ТЕМ ДИСЦИПЛИНЫ

5.1 Практические занятия

5.1.1. Трансляторы и компиляторы: определения, общая схема работы, фазы компиляции их особенности.

5.1.2. Формальные языки и грамматики: способы задания, классификация, регулярные и автоматные грамматики, алгоритмы преобразования грамматик.

5.1.3. Организация таблиц идентификаторов: простейшие методы, метод бинарного дерева, хэш-адресация, комбинированные методы.

5.1.4. Лексические и синтаксические анализаторы: определение и общая схема работы распознавателя; конечные автоматы, автоматы с магазинной памятью.

5.1.5. Генерация и оптимизация кода: подготовка к генерации кода, распределение памяти, методы генерации кода.

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

5.2. Лабораторные работы

5.2.1. Формальные языки и грамматики. Способы преобразования.

5.2.2. Алгоритмы построения таблиц идентификаторов.

5.2.3. Реализация алгоритма работы конечного автомата.

5.2.4. Минимизация конечного автомата.

5.2.5. Автоматы с магазинной памятью.

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

## 6. САМОСТОЯТЕЛЬНАЯ РАБОТА

| № п/п | № раздела (темы) дисциплины | Форма (вид) самостоятельной работы | Трудоемкость в часах |
|---|---|---|---|
| 1 | Трансляторы и компиляторы | Изучение учебной литературы | 10 |
| 2 | Формальные языки и грамматики | Изучение учебной литературы<br>Подготовка отчетов по лабораторным работам | 10 |
| 3 | Методы построения таблиц идентификаторов | Изучение учебной литературы<br>Подготовка к контрольной работе<br>Подготовка отчета по лабораторной работе | 10 |

| | | | |
|---|---|---|---|
| 4 | Лексические и синтаксические анализаторы | Изучение учебной литературы<br>Подготовка отчета по лабораторной работе | 10 |
| 5 | Генерация и оптимизация кода | Изучение учебной литературы<br>Подготовка к тесту<br>Подготовка отчета по лабораторной работе | 12 |
| 6 | Алгоритмы управления памятью в операционных системах | Изучение учебной литературы | 20 |
| | **Итого** | | 72 |

## 7. МАТРИЦА КОМПЕТЕНЦИЙ УЧЕБНОЙ ДИСЦИПЛИНЫ

| Разделы | Компетенции | | Итого |
|---|---|---|---|
| | ПК-6 | ПК-7 | Общее число компетенций |
| 1 | + | + | 2 |
| 2 | + | + | 2 |
| 3 | + | + | 2 |
| 4 | + | + | 2 |
| 5 | + | + | 2 |

## 8. ОБРАЗОВАТЕЛЬНЫЕ ТЕХНОЛОГИИ

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

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

проблемный семинар, начинающийся с постановки проблемы, которую необходимо решить в ходе изложения материала,

На практических занятиях 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.

При оценивании результатов обучения используется балльно-рейтинговая технология.

| № п/п | № раздела (темы) дисциплины | Форма (вид) образовательных технологий | Кол-во часов |
|---|---|---|---|
| 1 | Трансляторы и компиляторы | Мультимедийная презентация | 2 |
| 2 | Формальные языки и грамматики | Мультимедийная презентация | 2 |
| 3 | Методы построения таблиц идентификаторов | Мультимедийная презентация | 2 |
| 4 | Лексические и синтаксические анализаторы | Мультимедийная презентация | 2 |
| 5 | Генерация и оптимизация кода | Мультимедийная презентация | 2 |
| 6 | Алгоритмы управления памятью в операционных системах | Мультимедийная презентация | 2 |

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

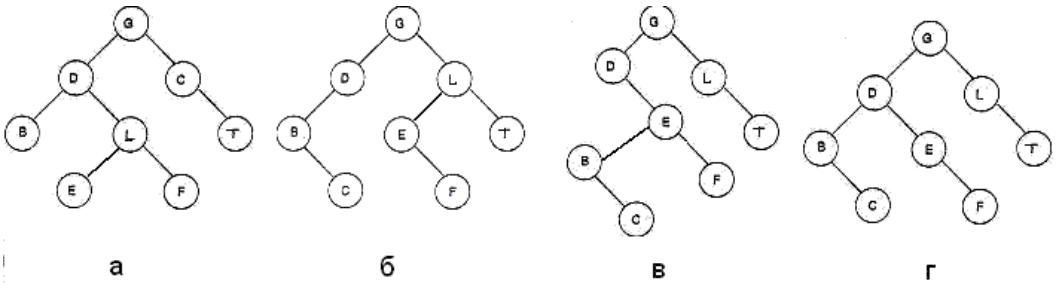
Для оценки текущей успеваемости в данной дисциплине относятся:
тестовые задания с закрытыми и открытыми видами вопросов;
тестирование (промежуточное);
отчеты по выполнению лабораторных работ;
экзамен.

Вопросы к экзамену

1. Цепочки символов. Операции над цепочками символов.
2. Формальное определение языка
3. Формальное определение грамматики. Форма Бэкуса-Наура
4. Общая схема работы распознавателя
5. Виды распознавателей
6. Четыре типа грамматик по Хомскому
7. Классификация языков
8. Определение транслятора. Однопроходные и многопроходные трансляторы
9. Этапы трансляции
10. Определение компилятора. Особенности построения и функционирования
11. Определение интерпретатора.
12. Организация таблиц идентификаторов. Простейшие способы построения
13. Построение таблиц идентификаторов по методу бинарного дерева
14. Принципы работы хэш-функций
15. Построение таблиц идентификаторов на основе хэш-функций
16. Построение таблиц идентификаторов по методу цепочек
17. Назначение лексического анализатора
18. Принципы построения лексических анализаторов
19. Конечные автоматы
20. Алгоритмы минимизации и преобразования конечных автоматов
21. Синтаксические анализаторы. Построение синтаксических анализаторов
22. Автоматы с магазинной памятью
23. Виды переменных
24. Виды и областей памяти
25. Статистическое и динамическое связывание
26. Стековая организация памяти.
27. Способы внутреннего представления программ
28. Принципы оптимизации кода
29. Структуризация виртуального адресного пространства
30. Функции ОС по управления памятью
31. Страничное распределение
32. Сегментное распределение
33. Сегментно-страничное распределение

Примеры тестовых заданий

1. К дереву какого вида приводит последовательность идентификаторов G, D, L, E, B, T, C, F ?




2. Как называется программа, переводящая исходную программу в эквивалентную ей программу на результирующем языке?
а)интерпретатором б) транслятором в) компилятором г)терминатором

3. Как называется процесс отображения области определения хэш-функции на множество значений?
а) хэш-адресацией б) хэш-функцией в)хэшированием г)рехэшированием

4. К какому типу относятся грамматики G(**VT,VN,P**,S), **V = VN** $\cup$ **VT** , которые могут иметь правила двух видов: A $\rightarrow$ B$\gamma$ или A $\rightarrow \gamma$ , где A, B$\in$ **VN**, $\gamma \in$ **VT* ?**
а)праволинейных грамматик б) леволинейных грамматик
в) контексно-свободных грамматик г) контексно-зависимых грамматик

5. Как называется автомат с магазинной памятью, если может заменять цепочку символов конечной длины в верхней части стека на другую цепочку символов конечной длины?
а) расширенным б) детерминированным в) недетерминированным г) обычным

6. Как называется следующий алгоритм преобразования конечного автомата (КА):
- из автомата исключаются все недостижимые состояния;
- строятся классы эквивалентности автомата;
- классы эквивалентности состояний исходного КА становятся состояниями результирующего минимизированного КА.
- функция переходов результирующего КА очевидным образом строится на основе функции переходов исходного КА.
а) детерминизации КА б) построения эквивалентных состояний
в) оптимизации КА г) минимизации КА

7. Какие из распознавателей просматривают входную цепочку символов слева направо и порождают левосторонний вывод?
а) восходящие б) линейные в) нисходящие г) универсальные

8. Как называется нетерминальный символ грамматики тогда и только тогда, когда из него нельзя вывести ни одной цепочки терминальных символов.
а) бесплодным б) циклическим в) недостижимым г) цепным

9. Алгоритм:
*Шаг 1*. Для всех символов $X \in VN$ повторить шаги 2-4, затем перейти к шагу 5.
*Шаг 2*. $N_0^X = \{X\},\ i = 1.$
*Шаг 3*. $N_1^X = N_{i-1}^X \cup \{B : (A \rightarrow B) \in P, B \in N_{i-1}^X\}.$
*Шаг 4*. Если $N_i^X \neq N_{i-1}^X$, то *i* = *i* +1, перейти к шагу 3, иначе $N_i^X - \{X\}$ и продолжить цикл по шагу 1.
*Шаг 5*. $VN' = VN,\ VT' = VT$ , в $P'$ входят все правила из *P,* кроме правил вида

$A \to B,\ S' = S$.

*Шаг 6*. Для всех правил $(A \to \alpha) \in P'$, если $B \in N^A$, $B \neq A$, то в $P'$ добавляются правила вида $B \to \alpha$.

преобразует КС-грамматику G(**VT,VN,P**,S), удаляя из нее

а) недостижимые символы б) цепные правила
в) бесплодные правила г) правила с пустыми цепочками

10. Выберите верные утверждения из нижеперечисленных:
а) компилятор может работать как с несколькими, так и с одной таблицей идентификаторов;
б) праволинейные и леволинейные грамматики не эквивалентны;
в) два конечных автомата эквивалентны, если задают один и тот же язык;
г) процесс построения лексического анализатора гораздо сложнее процесса построения синтаксического анализатора;
д) алфавиты терминальных и нетерминальных символов грамматики не пересекаются.

11. В качестве показателей эффективности оптимизации программы используют два критерия: _____ и _____ .

12. Для оптимизации чего применяют удаление бесполезных присваиваний, исключение избыточных вычислений, свертка объектного кода, перестановка операций, арифметические преобразования применяют?
а) логических выражений б) циклов
в) линейных участков программы г) вызовов функций

13. Распознавателями контекстно-свободных языков служат _____.

14. Неформальное описание работа автомата: «если на верхушке стека автомата находится цепочка символов γ, то ее можно заменить на нетерминальный символ А, не сдвигая считывающую головку, если в грамматике есть правило А→ γ. С другой стороны, если считывающая автомата обозревает некоторый символ входной цепочки *a,* то этот символ можно поместить в стек и передвинуть считывающую головку на одну позицию вправо» соответствует алгоритму.
а) «сдвиг-свертка» б) рекурсивного спуска
в) с подбором альтернативы г) множества альтернатив

15. Нисходящий распознаватель с подбором альтернатив моделирует работу МП-автомата с начальным состоянием _____и конечной конфигурацией _____.

16. Как называется фаза, на которой компилятором выполняются предварительные действия, непосредственно связанные с синтезом текста результирующей программы, но не ведущие к порождению текста на выходном языке?
а) генерация кода б) синтаксический разбор
в) семантический анализ г) подготовка к генерации кода

17. Структура правил _____грамматик такова, что при построении предложений заданного ими языка один и тот же нетерминальный символ может быть заменен на ту или иную цепочку символов в зависимости от того контекста, в котором он встречается.

## 10. **УЧЕБНО-МЕТОДИЧЕСКОЕ И ИНФОРМАЦИОННОЕ ОБЕСПЕЧЕНИЕ ДИСЦИП-**

**ЛИНЫ (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

2 Гордеев, А.В. Операционные системы. Учебник для вузов. (Допущено МинОбр РФ) – СПб: Питер, – 2009. – 416 с.

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. С/С++. Программирование на языке высокого уровня (Доп. Мин. образования РФ) – СПб.: Питер, 2009, 2010. – 461с.

4 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. |
| 3 | http://www.biblioclub.ru | Электронная библиотечная система «Университетская система -online» специализируется на учебных материалах для ВУЗов по научно-гуманитарной тематике, а также содержит материалы по точным и естественным наукам. |
| 4 | http://www.window.edu.ru | Единое окно доступа к образовательным ресурсам/ каталог/ профессиональное образование |

ПЕРИОДИЧЕСКИЕ ИЗДАНИЯ

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

## 11. МАТЕРИАЛЬНО-ТЕХНИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ДИСЦИПЛИНЫ (МОДУЛЯ)

Мультимедийная лекционная аудитория (331)

В качестве программного обеспечения используются свободно распространяемое про-

граммное обеспечение Dev C++.

## 12. РЕЙТИНГОВАЯ ОЦЕНКА ЗНАНИЙ СТУДЕНТОВ ПО ДИСЦИПЛИНЕ

Балльная структура оценки за 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 | Трансляторы и компиляторы | Тест №1 | 1 – 2 | 4 | 1 | 5 |
| 2 | Формальные языки и грамматики | Отчет по лаб. работе<br>Тест №1 | 3 – 6 | 14 | 2 | 16 |
| 3 | Методы построения таблиц идентификаторов | Отчет по лаб. работе<br>Тест №1 | 7 – 8 | 8 | 1 | 9 |
| 4 | Лексические и синтаксические анализаторы | Отчет по лаб. работе<br>Тест №2 | 9 – 10 | 8 | 1 | 9 |
| 5 | Генерация и оптимизация кода | Отчет по лаб. работе<br>Тест №2 | 11 – 12 | 14 | 1 | 15 |
| 6 | Алгоритмы управления памятью в операционных системах | Тест №2 | 13 – 16 | 4 | 2 | 6 |
| | Сдача экзамена | | | | | 40 |
| | Итого | | | | | 100 |

## 2 КРАТКОЕ ИЗЛОЖЕНИЕ ПРОГРАММНОГО МАТЕРИАЛА

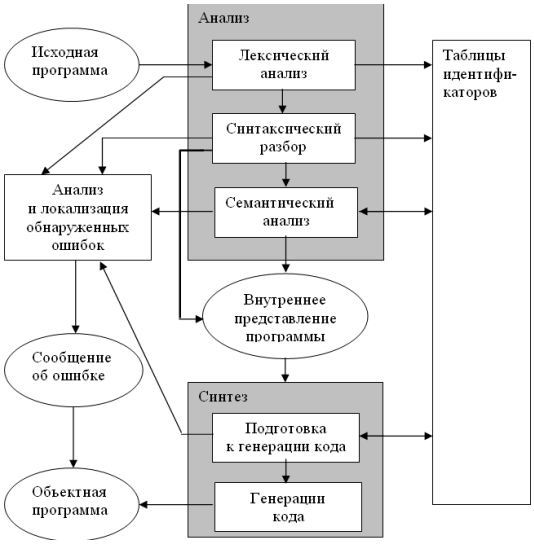
Практическое занятие 1

**Тема** Трансляторы и компиляторы

*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.




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.

Реальные компиляторы, как правило, выполняют трансляцию текста исходной программы за несколько проходов.

Практическое занятие 2

**Тема** Формальные языки и грамматики

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

Если V — некоторый алфавит, то:

$V^+$ — множество всех цепочек над алфавитом V без $\lambda$;

V* — множество всех цепочек над алфавитом V, включая $\lambda$.

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 $\alpha$ -> $\beta$, 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.

Практическое занятие 3

**Тема Р**егулярные и автоматные грамматики

1.Однозначность и эквивалентность грамматик

2. Понятие автоматной грамматики. Соотношение классов автоматной и регулярной грамматик.

3. 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→B$\gamma$ или A→$\gamma$, 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$\rightarrow \gamma$ B или A$\rightarrow \gamma$ , 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.

Практическое занятие 4

**Тема** 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.

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, 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: $F(r) = n$, $r \in R$, $n \in Z$. 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.

Практическое занятие 5

Тема Лексические и синтаксические анализаторы

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.

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.

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 $(q,\omega,n)$, 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, $\delta$, qo, Zo, F), где Q – множество состояний автомата; V – алфавит входных символов автомата; Z – специальный конечный алфавит магазинных символов автомата; $\delta$ – 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 помещать специальные (магазинные) символы. Обычно это терминальные и нетерминальные символы грамматики языка.

Практическое занятие 6

**Тема** Генерация и оптимизация кода

Генерация объектного кода – перевод компилятором внутреннего представления исходной программы в цепочку символов выходного языка. Генерацию кода можно считать функцией, определенной на синтаксическом дереве, построенном в результате синтаксического анализа, и на информации из таблицы идентификаторов. Чтобы компилятор мог построить код результирующей программы для синтаксической конструкции входного языка, часто используется метод, называемый синтаксически управляемым переводом – СУ-переводом. В нем каждому правилу входного языка компилятора сопоставляется одно или несколько правил, или не одного правила выходного языка в соответствии с семантикой входных и выходных правил.

С каждой вершиной дерева синтаксического разбора N связывается цепочка некоторого промежуточного кода C(N). Код для вершины N строится сцеплением (конкатенацией) в фиксированном порядке последовательности кода C(N) и последовательностей кодов, связанных со всеми вершинами, являющимися прямыми потомками N. Для построения последовательностей кода прямых потомков вершины N требуется найти последовательности кода их потомков. Таким образом, перевод идет снизу вверх, в строго установленном порядке, определенном структурой дерева. Кроме того схемы СУ-перевода могут выполнять следующие действия:

Помещение в выходной поток данных машинных кодов или команд ассемблера, представляющих результат работы компилятора;

Выдачу пользователю сообщений об обнаруженных ошибках и предупреждениях;

Порождение и выполнение команд, указывающих, что некоторые действия должны быть произведены самим компилятором.

Оптимизация программы – это обработка, связанная с переупорядочиванием операций в компилируемой программе с целью получения эффективной результирующей объектной программы. В качестве показателей эффективности используются два критерия: объем памяти, необходимый ля выполнения результирующей программы, и скорость ее выполнения.

Далеко не всегда удается выполнить оптимизацию так, чтобы удовлетворить обоим этим критериям. Зачастую сокращение необходимого программе объема данных ведет к уменьшению ее быстродействия и наоборот. Поэтому для оптимизации выбирается либо один из критериев, либо некий комплексный критерий, основанный на них.

Принципиально различаются два основных вида оптимизирующих преобразований:

- преобразования исходной программы (в форме ее внутреннего представления в компиляторе), независящие от результирующего объектного языка;
- преобразования результирующей объектной программы.

Первый вид преобразований не зависит от архитектуры целевой вычислительной системы. Он основан на выполнении хорошо известных и обоснованных математических и логических преобразований. Во вором типе – могут учитываться объем кэш-памяти и реализация компилятора.

Методы преобразования программ зависят от типов синтаксических конструкций исходного языка программы. Теоретически разработаны методы для следующих типовых конструкций: линейных участков программы, логических выражений, циклов, вызовов процедур и функций и др.

Практическое занятие 7

Оперативная память – важнейший ресурс вычислительной системы, требующий тщательного управления со стороны операционной системы.

Виртуальная память использует дисковую память для временного хранения не помещающихся в оперативную память данных и кодов выполняющихся процессов.

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,

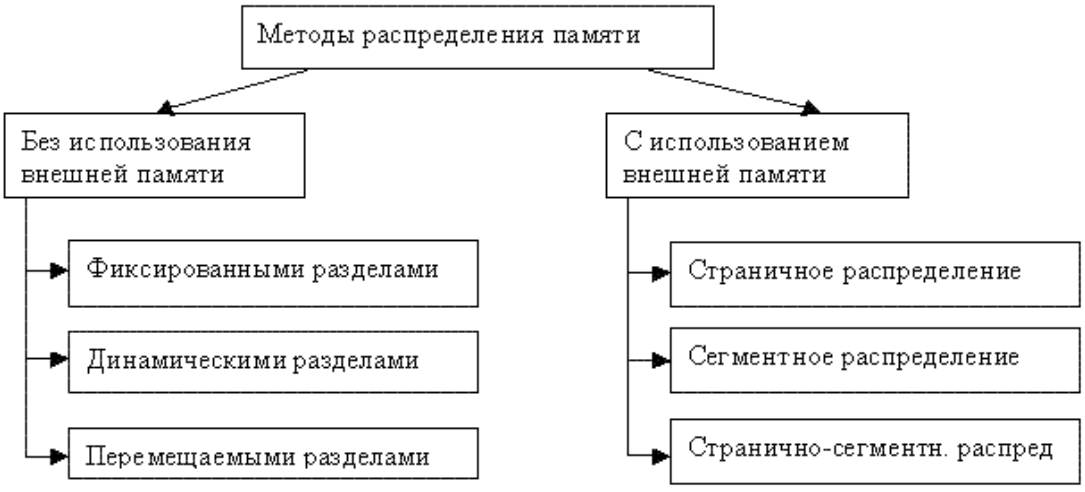
настройка адресов программы на конкретную область физической памяти.

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

Для идентификации переменных и программ на разных этапах жизненного цикла программы используются различные имена: символьные, виртуальные и физические. Совокупность виртуальных адресов процесса называется виртуальным адресным пространством. Каждый процесс имеет свое адресное виртуальное пространство.

Совпадение виртуальных адресов переменных и команд различных процессов не приводит к конфликтам, поскольку, когда переменные одновременно присутствуют в памяти, ОС отображает их на разные физические адреса.

Для обеспечения приемлемого уровня мультипрограммирования используется методы, в которых образы некоторых процессов целиком или полностью выгружаются на диск – свопинг и виртуальная память. Для временного хранения сегментов и страниц на диске отводится специальная область – страничный файл.




Общая классификация методов распределения памяти приведена на рисунке. Некоторые из методов представляют познавательный интерес,

другие встречаются в специализированных системах, третьи – сохранили свою актуальность.

Практическое занятие 8 Виртуальная память

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.

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 процесс из очереди процессов, находящихся в состоянии готовности. Параллельно программа обработки страничного прерывания находит на диске требуемую виртуальную страницу и пытается загрузить ее. Если в памяти имеется свободная физическая страница, то загрузка выполняется немедленно, иначе на основании принятой в данной системе стратегии замещения страниц решается вопрос о том, какую страницу следует выгрузить из оперативной памяти. Виртуальный и физический адреса представлены в виде пар – порядковый номер страницы и смещение в пределах страницы.

Деление виртуального адресного пространства на сегменты осуществляется компилятором на основе указаний программиста или, по умолчанию, в соответствии с принятыми в системе соглашениями. Размер сегмента определяется с учетом смыслового значения содержащейся в них информации. Отдельный сегмент может представлять собой подпрограмму, массив данных и т.п. Максимальный размер сегмента определяется разрядностью виртуального адреса.

Виртуальный адрес при сегментной организации памяти представлен парой, содержащей номер сегмента и смещение в сегменте. Физический адрес получается сложением базового адреса сегмента, который определяется по номеру сегмента из таблицы сегментов и смещения в сегменте, что замедляет процедуру преобразования адресов.

Сегментно-страничное распределение представляет собой комбинацию страничного и сегментного механизмов управления памятью и 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, что позволяет более эффективно использовать память, сократив до минимума фрагментацию.

## 3 МЕТОДИЧЕСКИЕ УКАЗАНИЯ

### Методические указания по изучению дисциплины

В результате освоения дисциплины у студентов развивается способность оценивать и выбирать современные операционные среды и информационно-коммуникационные технологии для информатизации и автоматизации решения прикладных задач и создания ИС. Для успеш-

ного 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. На первом занятие обязательно проводится инструктаж по выполнению техники безопасности.

Задания к лабораторным работам выполняются студентами парами на одном компьютере (работа в команде).

Желательно готовиться к лабораторным работам заранее по перечню тем, выданным преподавателем.

Выполняя задание, студенты пользуются материалом, изложенным в тексте лабораторной работы; готовят письменный отчет, включающий краткое изложение проделанных действий, ответы на контрольные вопросы, выводы.

Преподаватель, принимая лабораторную работу, проверяет навыки, полученные студентами при выполнении задания, отчет, задает дополнительные вопросы по отчету.

## Технология выполнения лабораторных работ

Тема 1. Работа с файловой системой с использованием JavaScript и Windows Scripting Host (WHS)

**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.

### 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 C.  В заключение он удаляется из обеих папок.

```
var  fso =  new ActiveXObject("Scripting. FileSystemObject")
var  f1 = 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.

**Тема 2** Формальные языки и грамматики. Способы преобразования.

Проанализировав материал первой и второй лекций, ответьте на следующие контрольные вопросы:

## Контрольные вопросы

1. Как выглядит описание грамматики в форме Бэкуса-Наура?
2. Каковы составляющие формального описания грамматики?
3. Как классифицируются языки? Как их классификация соотносится с классификацией грамматик?
4. Почему язык программирования нельзя не является чисто формальным языком?
5. Какой тип грамматики самый сложный?
6. Грамматики G(**VT,VN, P**, S), **V = VN** $\cup$ **VT** , которые могут иметь правила двух видов: $A \to B\gamma$ или $A \to \gamma$, где A, B$\in$ **VN**, $\gamma \in$ **VT*** относятся к типу:

а) праволинейных грамматик ;

б) леволинейных грамматик;

в) контексно-свободных грамматик;

г) контексно-зависимых грамматик.

7. Выберите верное утверждение:

а) неукорачивающие грамматики имеют такую структуру правил, что при построении предложений языка, заданного грамматикой, любая цепочка символов может быть заменена цепочкой символов меньшей длины;

б) целевой символ грамматики – это всегда терминальный символ;

в) языки классифицируются в соответствии с типами грамматик, с помощью которых они заданы;

г) языки программирования являются формальными языками.

8. Для любого языка, заданного какой грамматикой, можно построить неукорачивающую грамматику, которая будет задавать эквивалентный язык.

а) контекстно-свободной грамматикой

б) грамматикой с фразовой структурой

в) контексно-зависимой грамматикой

г) регулярной грамматикой

## Задание

1. Определить тип указанных грамматик

а) G1 ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, – }, {A, B}, P, A):

P:
A→ B | +B | –B
B→ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0B | 1B | 2B | 3B | 4B | 5B | 6B | 7B | 8B | 9B

б) G2 ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, – }, {S, B}, P, S):

P:
B→ + | – | $\lambda$
S→ B0 | B1 | B2 | B3 | B4 | B5 | B6 | B7 | B8 | B9 | S0 | S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8 | S9

в) G3 ({0, 1}, {A, S}, P, S)

P:
S→ 0A1
0A→00A1
A→ $\lambda$

г) G4 ({0, 1}, {A, S}, P, S)

P:
S→ 0A1 | 01
0A→00A1 | 001

A→ $\lambda$

д) G5 ({0, 1}, { S}, P, S)

P:
S→ 0S1 | 01

е) G5 ({f, g, h}, { G, H, E, S}, P, S)

P:
S→ GH
G → fGgH | fg
Hg →gH
HE → Hh
gEh → ghh
fgE → fgh

2. Определить язык грамматики G ( {+, – , *, /, (, ), x, y}, {S}, P, S):

P:
S → S+S | S–S | S*S | S/S | (S) | x | y

3. Поездом называется произвольная последовательность локомотивов и вагонов. Построить грамматику в форме Бэкуса-Наура для понятия «поезд», если

а) поезд всегда начинается с локомотива;
б) все локомотивы должны быть сосредоточены в начале поезда;
в) поезд начинается с локомотива и заканчивается локомотивом;
г) в поезде должны чередоваться через два локомотивы и вагоны;
д) поезд не должен содержать два локомотива или два вагона подряд;
е) поезд не должен содержать подряд два локомотива.

4. Дана грамматика G**({“ ”,** i, f, t, h, e, n, l, s, b, a**}**, {E}, P**,** E**)** с правилами:

P:
E → if b then E else E; | if b then E; | a
Не строя цепочек вывода, показать, что грамматика является неоднозначной. Проверить это, построив некоторую цепочку вывода.

5. Построить эквивалентную ей однозначную грамматику. Построить для нее дерево вывода.

**Тема 3.** Алгоритмы построения таблиц идентификаторов

Изучив материал соответствующей лекции, ответьте на 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 на основе метода цепочек. В качестве хэш-функции использовать варианты из предыдущего задания.

5. Сравнить реализованные методы построения таблиц идентификаторов.

**Тема 4.** Реализация алгоритма работы конечного автомата

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, называемого 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 отвергнутся. 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, $\delta(a,q)=R$, $a \in V$, $q \in Q$, $R \subseteq Q$,

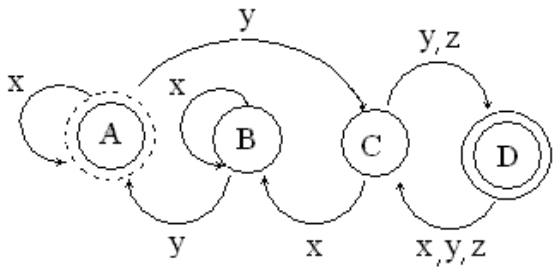
$q_0$ – начальное состояние автомата ($q_0 \in Q$),

F – непустое множество конечных состояний автомата.

Конфигурация автомата на каждом шаге работы определяется тройкой $(q,\omega,n)$, 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 оп-



ределена соответствующая функция перехода δ. Начальное и конечное состояние автомата помечаются специальным образом.

На рис. 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» – E. На него замыкают все неопределенные переходы, в том числе и само на себя.

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. 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. 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 $A \to Ba_1$, $A, B \in \mathbf{VN}$, $a_1 \in \mathbf{VT}$ или вида $A \to a_1$, $A \in \mathbf{VN}$, $a \in \mathbf{VT}$, то они переносятся во множество P' правил грамматики $G^1$ без изменений.

Если встречаются правила вида $A \to Ba_1, a_2, ..., a_n, n > 1, A, B \in \mathbf{VN}, \forall n > i > 0: a_i \in \mathbf{VT}$, 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 $A \to a_1 a_2 ..., a_n, n > 1, A, B \in \mathbf{VN}, \forall n > I > 0: a_i \in \mathbf{VT}$, 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, $\forall A, B, C \in \mathbf{VN'}$, $\forall a \in \mathbf{VT'}$ (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, $\forall A, B, C \in \mathbf{VN'}$, $\forall a \in \mathbf{VT'}$ (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 $A \to B$ (такие правила называются цепными) или вида $A \to \lambda$ (такие правила называются $\lambda$ - 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 C++: inline, function, class, protected, extern, delete, operator, struct.

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 |

**Тема 5**. Минимизация конечного автомата

Алгоритм преобразования произвольного конечного автомата $M(Q, V, \delta, q_0, F)$ к эквива-

лентному ему, детерминированному конечному автомату $M'(Q', V, \delta', q_0', F')$, 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 $\delta'$ 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 $[\ldots, f_i, \ldots]$.

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*. $P_{i+1}=\varnothing$.

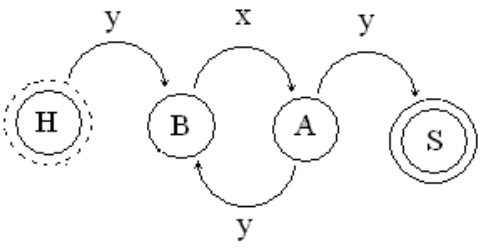
*Шаг 3*. $\forall\, a \in V,\ \forall\, q \in P_i\ \ P_{i+1}=P_i \cup \delta(a, q)$

*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.




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

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

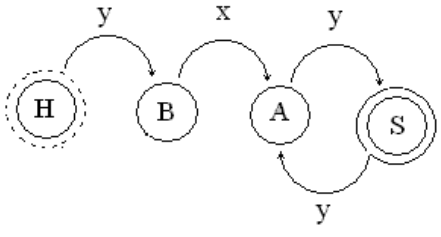
*Шаг 2*. Функция переходов эквивалентного конечного автомата

| | | |
|---|---|---|
| $\delta'([H], y)=[B]$ | $\delta'([A], y)=[BS]$ | $\delta'([B], x)=[A]$ |
| $\delta'([HA], y)=[BS]$ | $\delta'([HB], x)=[A]$ | $\delta'([HS], y)=[B]$ |
| $\delta'([AB], x)=[A]$ | $\delta'([AB], y)=[BS]$ | $\delta'([AS], y)=[BS]$ |
| $\delta'([BS], x)=[A]$ | $\delta'([HAB], x)=[A]$ | $\delta'([HAS], y)=[BS]$ |
| $\delta'([HBS], y)=[B]$ | $\delta'([HBS], x)=[A]$ | $\delta'([ABS], y)=[BS]$ |
| $\delta'([ABS], x)=[A]$ | $\delta'([HABS], x)=[A]$ | $\delta'([HABS], y)=[BS]$ |

*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 $M'(\{H, B, A, S\}, \{x, y\}, \delta', H, \{S\})$, $\delta'(H, y)=B$, $\delta'(B, x)=A$, $\delta'(A, y)=S$, $\delta'(S, x)=A$. Граф переходов автомата приведен на рис.




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 миними-

зировать. 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 |

Таблица переходов не минимизированного автомата

Поэтому можно объединить состояния K и L в одно состояние и назвать его X. Получается упрощенная таблица состояний:

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

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:

| | *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 |

Выявим его эквивалентные состояния. 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.

| | *y* | *z* |
|---|---|---|
| F, P | | |

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

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:

| | *y* | *z* |
|---|---|---|
| F, P | F, N | K |

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 y 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 | | |

Результаты промежуточного заполнения таблицы:

| | *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 | | |

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.

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:

| | *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 |

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 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:

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

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.

3. Для автоматов из предыдущего задания найти недостижимые состояния.

4. Найти недостижимые состояния автомата, представленного таблицей состояний

| | *X* | *Y* | |
|---|---|---|---|
| S1 | S1 | S3 | 1 |
| S2 | S7 | S4 | 0 |
| 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 |

**Тема 6**. Автоматы с магазинной памятью.

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, $\delta$ , qo, Zo, F), где Q – множество состояний автомата;

V – алфавит входных символов автомата;

Z – специальный конечный алфавит магазинных символов автомата;

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

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. 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.

| | a | b | c | ┤ |
|---|---|---|---|---|
| A | ВТОЛКНУТЬ (A) СДВИГ | ОТВЕРГНУТЬ | ВЫТОЛКНУТЬ СДВИГ | ОТВЕРГ-НУТЬ |
| B | ВТОЛКНУТЬ (C) СДВИГ | ВЫТОЛКНУТЬ ДЕРЖАТЬ | ВТОЛКНУТЬ (A) СДВИГ | ОТВЕРГ-НУТЬ |
| C | ВТОЛКНУТЬ (B) ДЕРЖАТЬ | ВТОЛКНУТЬ (C) СДВИГ | ВЫТОЛКНУТЬ СДВИГ | ОТВЕРГ-НУТЬ |
| Магазин-ный символ | ВТОЛКНУТЬ (A) СДВИГ | ОТВЕРГНУТЬ | ОТВЕРГНУТЬ | ДОПУС-ТИТЬ |

**Методические рекомендации по организации самостоятельной работы студентов**

Общая структура самостоятельной работы представлена в рабочей программе и включает 94 часа, часть из которых предназначены для подготовки к лабораторным работам, остальные для подготовки вопросов самостоятельного изучения. Вопросы даются в соответствии с лекционными занятиями.

**Контрольные вопросы для самоподготовки студентов**

1. Почему распределение памяти не может быть выполнено до выполнения семантического анализа?
2. Как работает процесс распределения памяти с таблицей идентификаторов?
3. Каковы принципы распределения памяти?
4. По каким параметрам классифицируют области памяти?
5. На чем основа стековая организация памяти?
6. Принципы организации дисплея памяти функции.
7. Каковы исторические и технические причины создания модулей в составе систем программирования?
8. История создания и эволюции систем программирования

9. Какие функции выполняет текстовый редактор в системе программирования?
10. Каковы особенности компилятора в составе системы программирования?
11. В чем преимущества и недостатки динамически загружаемых библиотек по сравнению со статическими?
12. Какие функции загрузчика выполняет операционная система, если он не входит в состав системы программирования?
13. Какие современные средства отладки существуют?
14. В чем отличие отладчика и интерпретатора?

# 4 КОНТРОЛЬ ЗНАНИЙ

К текущему контролю знаний относятся контрольные вопросы при выполнении лабораторных работ, вопросы для самоподготовки, тестовые задания.

Контрольные вопросы при выполнении лабораторных работ предложены в каждой теме раздела «Технология выполнения лабораторных работ», вопросы для самоподговки даны в предыдущем разделе, примеры тестовых заданий представлены в рабочей программе.

Итоговым контролем знаний является экзамен.

Примеры экзаменационных билетов:

Билет№1
1. Принципы работы транслятора, компилятора, интерпретатора
2. Регулярные и автоматные грамматики Преобразование регулярной грамматики к автоматному виду

Билет №2
1. Организация таблиц идентификаторов
2. Сегментная организация памяти

Билет №3
1. Метод логарифмического поиска построения таблиц идентификаторов
2. Страничная организация памяти

Билет №4
1. Построение таблиц идентификаторов по методу бинарного дерева
2. Понятие грамматики. Контекстно-свободные грамматики. Приведенные грамматики КС-грамматики

Билет №5
1. Этапы трансляции. Общая схема работы транслятора
2. Алгоритм удаления бесплодных символов КС-грамматики

СОДЕРЖАНИЕ