Федеральное агентство по образованию РФ
АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
(ГОУВПО «АмГУ»)

УТВЕРЖДАЮ
Зав. кафедрой ИУС
_____А.В.Бушманов
«_____»_____

УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС

по дисциплине «Системное программное обеспечение»

для студентов специальности 230102 – Автоматизированные системы обработки информации и управления

Составитель доцент кафедры ИУС Галаган Т.А.

Факультет математики и информатики

Кафедра информационных и управляющих систем

2007

Печатается по решению
редакционно-издательского совета
факультета математики и информатики
Амурского государственного
университета

Т.А. Галаган

Учебно-методический комплекс по дисциплине «Системное программное обеспечение». Для студентов специальности 230102 «Автоматизированные системы обработки информации и управления» очной формы обучения.- Благовещенск: Амурский гос. ун-т, 2007.

Пособие содержит рабочую программу, курс лекций, методические рекомендации по проведению и выполнению лабораторных работ. Составлено в соответствии с требования государственного образовательного стандарта.

I. ПРИМЕРНАЯ ПРОГРАММА УЧЕБНОЙ ДИСЦИПЛИНЫ, УТВЕРЖДЕННАЯ МИНОБРАЗОВАНИЯ РФ

Государственный образовательный стандарт высшего профессионального образования

Направление подготовки дипломированного специалиста 654600 - Информатика и вычислительная техника

Образовательная программа – 230201 Автоматизированные системы обработки информации и управления

наименование дисциплины – Системное программное обеспечение

блок специальных дисциплин, индекс СД.11.

Всего часов - 100

Содержание разделов:

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

II. РАБОЧАЯ ПРОГРАММА

Семестр – 8

Лекции – 30 часов

Лабораторные работы – 30 часов

Самостоятельная работа – 40 часов

Экзамен – 8 семестр

1. ЦЕЛИ И ЗАДАЧИ ДИСЦИПЛИНЫ, ЕЕ МЕСТО В УЧЕБНОМ ПРОЦЕССЕ

1.1. Традиционная архитектура компьютера остается неизменной, неизменны и базовые принципы построения программного обеспечения – трансляторы, компиляторы и интерпретаторы. В курсе лекций излагаются теоретические принципы и технологии, лежащие в основе современных средств разработки программного обеспечения.

1.2. По завершению курса «Системное программное обеспечение» студент должен:

- владеть такими понятиями, как распознаватели и преобразователи, формальные языки и грамматики;
- знать назначение и функции компиляторов, трансляторов, интерпретаторов, современное состояние теории операционных систем и методы, используемые при их разработке;
- знать теоретические основы разработки синтаксических и семантических анализаторов;
- иметь устойчивые практические навыки работы с файловыми системами, поддерживаемыми MS DOS, Windows98/2000/XP;
- уметь создавать программы, расширяющие возможности операционных систем.

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

2. СОДЕРЖАНИЕ ДИСЦИПЛИНЫ

2.1.ФЕДЕРАЛЬНЫЙ КОМПОНЕНТ

Программа курса «Системное программное обеспечение» составлена в соответствие с требованиями государственного образовательного стандарта специализации – Интегрированные системы автоматизированного управления, специализации 230201, блок специальных дисциплин СД.11.

2.2. ЛЕКЦИИ (30 часов)

2.2.1. Пользовательский интерфейс операционной системы; управление задачами, управление памятью, управление вводом-выводом, управление файлами. Ассемблеры. Макроязыки (4 часа)

2.2.2. Формальные языки и грамматики. Способы задания языков. Классификация языков и грамматик (2 часа).

2.2.3. Трансляторы, компиляторы, интерпретаторы: общая схема работы, особенности построения (2 часа).

2.2.4. Организация таблиц идентификаторов: простейшие методы, метод бинарного дерева, хэш-адресация, комбинированные методы. (4 часа).

2.2.5. Лексические анализаторы: регулярные и автоматные грамматики, конечные автоматы, примеры построения лексических анализаторов (8 часов)

2.2.6. Синтаксические анализаторы: автоматы с магазинной памятью, контекстно-свободные грамматики и их преобразование, распознаватели с возвратом, нисходящие распознаватели (6 часов)

2.2.7.Общие принципы генерации кода. Синтаксически управляемый перевод, способы внутреннего представления программ. Принципы оптимизации кода. (4 часа)

2.3.–

2.4.ЛАБОРАТОРНЫЕ РАБОТЫ (30 часов)

2.4.1. Работа с файловой системой с использованием библиотеки <dos.h> языка C++. (2 часа)

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

Работа с папками (2 часа)

Работа с файлами (2 часа)

Работа с дисками (2 часа)

Работа с ярлыками (2 часа)

Работа с реестром (2 часа)

2.4.3. Структура компилятора (2 часа)

2.4.4. Конечный автомат (4 часа)

2.4.5. Получение минимального автомата (4 часа)

2.4.6. Построение таблиц идентификаторов (2 часа)

2.4.7. Автоматы с магазинной памятью. (2 часа)

2.4.8. Нисходящие методы обработки языков с помощью МП-автомата. (4 часа).

2.5. -

2.6. САМОСТОЯТЕЛЬНАЯ РАБОТА

Для самостоятельного изучения студентам рекомендованы темы:

2.6.1. Распределение памяти: виды переменных и областей памяти

2.6.2. Интерактивные системы

2.6.3. Средства трассировки и отладки программ

2.6.4. Принципы функционирования систем программирования.
2.6. 5. Современные операционные системы
2.6. 6. Мобильность программного обеспечения
Перечисленные темы включены в экзаменационные вопросы.

2.7. ВОПРОСЫ К ЭКЗАМЕНУ

1. Пользовательский интерфейс операционной системы
2. Управление задачами
3. Управление памятью
4. Управление вводом-выводом
5. Управление файлами.
6. Ассемблеры.
7. Макроязыки
8. Цепочки символов. Операции над цепочками символов.
9. Формальное определение языка
10. Способы задания языков. Синтаксис и семантика языка.
11. Формальное определение грамматики. Форма Бэкуса-Наура
12. Запись грамматик с использованием метасимволов
13. Запись грамматик в графическом виде
14. Общая схема распознавателя
15. Виды распознавателей
16. Четыре типа грамматик по Хомскому
17. Классификация языков
18. Определение транслятора. Однопроходные и многопроходные трансляторы
19. Этапы трансляции
20. Определение компилятора. Особенности построения и функционирования
21. Определение интерпретатора. Особенности построения и функционирования
22. Организация таблиц идентификаторов. Простейшие способы построения
23. Построение таблиц идентификаторов по методу бинарного дерева
24. Принципы работы хэш-функций
25. Построение таблиц идентификаторов на основе хэш-функций
26. Построение таблиц идентификаторов по методу цепочек
27. Назначение лексического анализатора
28. Принципы построения лексических анализаторов
29. Синтаксические анализаторы. Построение синтаксических анализаторов
30. Автоматы с магазинной памятью
31. Нисходящий распознаватель с возвратом
32. Виды переменных
33. Виды и областей памяти

34. Средства трассировки и отладки программ.
35. Интерактивные системы.
36.Принципы функционирования систем программирования.
37.Мобильность программного обеспечения
38.Средства трассировки и отладки программ
39. Особенности операционных системы Windows NT/2000/XP
40. Операционная система Linux

КРИТЕРИИ ОЦЕНОК ЗНАНИЙ СТУДЕНТОВ

Отлично

Студент дает полные ответы на теоретические вопросы билета, показывая глубокое знание учебного материала, свободное владение основными понятиями и терминологией; ответ на дополнительный вопрос.

Хорошо

Студент дает ответы на теоретические вопросы билета, показывая прочное знание учебного материала, владение основными понятиями и терминологией; ответ на дополнительный вопрос.

Удовлетворительно

Студент дает неполные ответы на теоретические вопросы билета, показывая поверхностное знание учебного материала, владение основными понятиями и терминологией; при неверном ответе на билет ответы на наводящие вопрос.

Неудовлетворительно

Студент не дает полные ответы на теоретические вопросы билета, показывая лишь фрагментарное знание учебного материала, незнание основных понятий и терминологии; наводящие вопросы остаются без ответа.

3. УЧЕБНО_МЕТОДИЧЕСКИЕ МАТЕРИАЛЫ ПО ДИСЦИПЛИНЕ

3.1. ОСНОВНАЯ ЛИТЕРАТУРА

3.1.1. Молчанов А. Ю. Системное программное обеспечение. СПб.: Питер, - 2003. – 396 с. (Допущено Министерством образования РФ)

3.1.2. Павловская Т.А. С/С++: Учебник. СПб.: Питер, - 2003. – 456 с.

3.1.3 Галаган Т.А., Соловцова Т.А. Практикум по лингвистическим основам информатики. Благовещенск: Изд-во АмГУ. – 2005. – 98 с.

3.1.4. Гордеев А.В. Операционные системы. Учебник для вузов. 2-е издание. СПб: Питер, 2004. 416 с. (Допущено Министерством образования РФ)

3.1.5. Дунаев В JavaScript. СПб.: Питер. – 2005. - 394 с.

3.2. ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

3. 2. 1.Федоров В.В. Основы построения трансляторов: Учебное пособие. Обнинск: ИАТЭ, - 1995. – 105 с.

3.2.2. Олифер В.Г., Олифер Н.А. Сетевые операционные системы. СПб.: Питер, 2001. 544 с.

3.2.3. Данкин Р. Профессиональная работа с MS DOS: М.: Мир, 1993

4. УЧЕБНО-МЕТОДИЧЕСКАЯ (ТЕХНОЛОГИЧЕСКАЯ) КАРТА ДИСЦИПЛИНЫ

Номер недели	Вопросы, изучаемые на лекции	Занятия (номера)		Используемые нагляд. и метод. пособия	Самостоятельная работа студентов		Форма контроля
		(семин.)Практич	Лаборат.		Содержание	часы	
1	2.2.1		2.4.1	карточки	2.6.1	2	отчет
2	2.2.1		2.4.2	карточки	2.6.1	2	отчет
3	2.2.2		2.4.2	карточки	2.6.2	3	отчет
4	2.2.3		2.4.2	карточки	2.6.2	3	отчет
5	2.2.4		2.4.2	карточки	2.6.3	2	отчет
6	2.2.4		2.4.3	карточки	2.6.3	3	отчет
7	2.2.5		2.4.3	3.1.3.	2.6.3	3	отчет
8	2.2.5		2.4.4	3.1.3.	2.6.4	2	отчет
9	2.2.5		2.4.5	3.1.3.	2.6.4	3	отчет
10	2.2.5		2.4.6	3.1.3.	2.6.4	3	отчет
11	2.2.6		2.4.7	3.1.3.	2.6.5	4	отчет
12	2.2.6		2.4.7	3.1.3.	2.6.5	3	отчет
13	2.2.6		2.4.8	3.1.3.	2.6.5	3	отчет
14	2.2.7		2.4.8	3.1.3.	2.6.7	2	отчет
15	2.2.7		2.4.9	3.1.3.	2.6.7	2	отчет

III ГРАФИК САМОСТОЯТЕЛЬНОЙ РАБОТЫ СТУДЕНТОВ

Для самостоятельного изучения студентам рекомендованы темы:

	Тема	Количество часов
1	Распределение памяти: виды переменных и областей памяти	4
2	Интерактивные системы	6
3	Средства трассировки и отладки программ	8
4	Принципы функционирования систем программирования	8
5	Современные операционные системы	10

6	Мобильность программного обеспечения	4

Объем самостоятельной работы составляет 40 часов.

Контрольные вопросы по самостоятельной проверке изученного материала:

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. Каковы особенности систем программирования Borland Delphi, Borland C++ Builder, Microsoft Visual C++?
13. Почему компоновщик носит также название «редактор связей»?
14. Какие функции загрузчика выполняет операционная система, если он не входит в состав системы программирования?
15. Какие современные средства отладки существуют?
16. В чем отличие отладчика и интерпретатора?
17. Каковы особенности ОС Windows NT?
18. Какие методы диспетчеризации используются в Windows NT?
19. Опишите основные архитектурные особенности семейства Windows NT
20. Каковы особенности ОС Linux?
21. Перечислите и поясните основные понятия, относящиеся к Unix-системам

Проверка знаний перечисленных тем осуществляется на экзамене.

IV. МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ПРОВЕДЕНИЮ И ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ. ЗАДАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ

Лабораторные работы проводятся в компьютерных классах по подгруппам. В тексте каждой лабораторной работы содержатся краткие теоретические сведения, примеры фрагментов кодов программ с комментариями, задания для выполнения. Студенты выполняют задания индивидуально, демонстрируя преподавателю работающий программный продукт и отвечая на вопросы.

ЛАБОРАТОРНАЯ РАБОТА №1 (2 часа)
Тема **Работа с файловой системой с использованием библиотеки <dos.h> языка C++**

С использованием справочной информации из библиотеки <dos.h> языка C++ составьте следующие программы:

1. Измените программно значение текущей даты и времени на компьютере.
2. Создайте программным способом файл. Выполните программно его редактирование, переименование, копирование, перемещение.
3. Создайте программным способом папку. Выполните программно ее редактирование, переименование, копирование, перемещение.

РАБОТА С ФАЙЛОВОЙ СИСТЕМОЙ И РЕЕСТРОМ WINDOWS С ИСПОЛЬЗОВАНИЕМ JAVASCRIPT

Создание объекта файловой системы

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

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

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

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

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

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

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

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

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

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

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

Общий синтаксис:

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

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

Методы доступа к существующим объектам и методы создания новых объектов:

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

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

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

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

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

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

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

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

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

Задание

Создайте различными способами объект файловой системы. Продемонстрируйте вызов методов для созданного объекта.

ЛАБОРАТОРНАЯ РАБОТА №2 (2 часа)
Тема **Работа с дисками**

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

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

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

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

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

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

WScript.Ecno(driveInfo("A"))
```

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

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

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

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

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

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

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

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

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

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

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

Задание

1. Воспользовавшись приведенными листингами программ получите информацию о всех дисках компьютера.
2. Выведите диалоговое окно о дисках C и F.

ЛАБОРАТОРНАЯ РАБОТА №3 (2 часа)
Тема **Работа с папками**

Создание папки

Для создания папки можно использовать следующий код:

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

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

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

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

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

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

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

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

```
function createFolder(folderpath){
```

```
/* создание папки Возвращает: -1. если папка создана или существует, и
0 - в противном случае */

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

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

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

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

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

Копирование, перемещение и удаление папки

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

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

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

Примеры

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

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

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

Задание

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

ЛАБОРАТОРНАЯ РАБОТА №4 (2 часа)
Тема **Работа с файлами**

Создание текстового файла

Для создания текстового файла на диске, следует выполнить следующие:

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

fso.CreateTextFile(filepath)

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

Для выполнения с помощью WSH можно использовать и:

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

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

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

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

Листинг

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

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

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

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

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

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

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

Примеры

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

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

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

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

Примеры

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

Копирование, перемещение и удаление файла

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

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

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

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

В следующем примере создается текстовый файл testfile.txt в папке C:\Мои документы, затем этот файл перемещается в папку C:\Windows\Temp и копируется в корневую папку на диске 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:
f1 = 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

Чтение и запись данных производится с помощью следующих трех этапов:

- открытие файла;
- чтение или запись данных;
- закрытие файла.

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

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

либо

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

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

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

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

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

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

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

ReadLine( ) — применяется для чтения строки, при этом исключается символ перехода на новую строку;
ReadAll() — применяется для чтения всего содержимого текстового файла.

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

Пример

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

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

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

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

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

Задание

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

ЛАБОРАТОРНАЯ РАБОТА № 5 (2 часа)
Тема: **Создание ярлыков**

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

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

■ Первый способ:

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

■ Второй способ:

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

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

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

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

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

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

```
var  Myshell  =  new ActiveXObject ("WScript. Shell")
var mypath = Myshell. SpecialFolders("Desktop")    // путь  к папке  Рабочий стол
/* Создание ярлыка и подписи к нему:  */
var myshortcut  = Myshell. CreateShortcut(mypath +  "\\Мой  Блокнот.lnk")
/* Папка  расположения Windows:  */
var mywindir  = Myshell. ExpandEnvironmentStrings ("%windir%")
 /* Параметры ярлыка:  */
// расположение файла:
 myshortcut.TargetPath = Myshell. ExpandEnvironmentStrings (mywindir  +
"\\notepad.exe")
myshortcut.WorkingDirectory = Myshell.ExpandEnvironmentStrings(mypath)
  // рабочая  папка
myshortcut.WindowStyle = 4                                  // тип окна  (стандартное)
/* файл,  содержащий  графическое изображение ярлыка:  */
```

```
myshortcut. IconLocation = Myshell. ExpandEnvironmentStrings (mywindir +
"\\notepad.exe")
myshortcut.Save( )                                    // сохранить на диске
```

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

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

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

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

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

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

Запуск приложений

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

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

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

Задание

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

ЛАБОРАТОРНАЯ РАБОТА № 6 (2 часа)
Тема **Работа с реестром**

Реестр Windows представляет собой базу данных, содержащую сведения об ее настраиваемых параметрах, или, как еще говорят, о конфигурации операционной системы. Кроме того, в реестре хранится информация о настройках аппаратных средств компьютера и программ. Реестры различных систем и версий семейства Windows частично различаются. Однако между ними много общего. В Windows 3.1 (Windows for WorkGroups) реестр хранится в файле reg.dat в папке Windows. В более поздних версиях, Windows 9x/Me и т. д., реестр размещается в двух файлах, расположенных в папке Windows: system.-dat и user.dat. Реестр задумывался для замены настроечных ini-файлов. Записи в файле реестра имеют более удобную древовидную структуру. Хотя ini -файлы также поддерживаются Windows, разработчикам программного обеспечения рекомендуется хранить настроечную информацию в реестре.

Чтобы работать с реестром, необходимо понимать его структуру. Древовидная структура реестра представляет собой иерархически упорядоченное множество разделов (папок), содержащее следующие шесть разделов самого верхнего (корневого) уровня (табл.).

Таблица. Разделы реестра

Имя раздела реестра верхнего уровня	**Сокращенное имя раздела реестра**	**Описание**
HKEY_CLASSES_ROOT	HKCR	Содержит информацию о зарегистрированных типах файлов, OLE и др.
HKEY CURRENTJJSER	HKCU	Содержит параметры настройки оболочки Windows для пользователя, вошедшего в Windows. Например, настройки Рабочего стола, меню кнопки Пуск. Если на компьютере работает единственный пользователь и используется обычный вход в Windows, то со-

		держимое этого раздела берется из подраздела HKEY_USERS\DEFAULT
HKEY_LOCAL_MACHINE	HKLM	Содержит информацию об установленных драйверах и программном обеспечении
HKEY_USERS	HKU	Содержит параметры настройки оболочки Windows для всех пользователей. Информация этого раздела копируется в раздел HKEY_CURRENT_USER. С другой стороны, все изменения в HKEY_CURRENT_USER автоматически переносятся в раздел HKEY_USER
HKEY_CURRENT_CONFIG	HKCC	Содержит информацию о настройках устройств Plug&Play, а также сведения о настройках компьютера с переменным составом аппаратных средств
HHEY_DYN_DATA	HKDD	Содержит изменяющиеся данные о состоянии устройств, установленных на компьютере пользователя. Эти сведения отображаются в окне Панель управления> Система> Устройства> Свойства. Данные этого раздела обновляются операционной системой Windows, и поэтому не рекомендуется редактировать его самостоятельно

Перечисленные коренные разделы имеют подразделы, а те — свои подразделы и т. д. В конечном разделе ветви дерева реестра определяются параметры. Каждый параметр имеет имя и значение. Работа с реестром заключается в просмотре, создании и удалении его записей. Так, можно создать или удалить раздел реестра, создать или удалить параметр в каком-либо разделе. Однако делать это без четкого понимания целей и правил не рекомендуется.

Не следует открывать и редактировать файлы реестра system.dat и user.dat в текстовом редакторе.

Рассмотрим, как можно изменять записи в реестре, если такая необходимость возникла. Для работы с реестром используются три основных способа:

С помощью редактора реестра — программы regedit.exe. Чтобы запустить реактор реестра, достаточно выполнить команду Пуск ► Выполнить, ввести с клавиатуры слово regedit и щелкнуть на кнопке ОК. Данный способ наиболее безосный.

С помощью reg-файлов. Это текстовые файлы с расширением reg, записи в которых имеют довольно простую структуру. Запуск reg-файла, например, двойным щелчком по его имени в Проводнике Windows приводит к открытию диалогового окна с предложением добавить информацию из этого файла в реестр. При вашем согласии данные из reg-файла будут импортированы в файлы реестра.

С помощью программы JavaScript, использующей специальные методы Windows Scripting Host. В этом случае можно организовать не видимую пользователем работу с реестром (если, конечно, он не заблокировал выполнение сценариев).

Рассмотрим сначала структуру записей в reg-файле, который можно создать с помощью обычного текстового редактора, например Блокнота Windows. В первой строке этого файла должно быть написано прописными буквами REGEDIT4. Затем должна быть пустая строка. В следующей, третьей строке в квадратных скобках пишется имя раздела реестра. В четвертой строке располагается запись согласно формату:

"имя_параметра"=значение

Если в данном разделе реестра следует разместить еще один параметр, то запись о нем располагается в следующей строке. Таким образом, сведения о каждом параметре записываются в отдельной строке reg-файла. Аналогичным образом можно создать записи, относящиеся к другому разделу реестра. Однако между такими секциями, каждая из которых соответствует отдельному разделу реестра, обязательно должна быть одна пустая строка. Таким образом, структура reg-файла имеет следующий вид:

```
REGEDIT4
[имя_раздела1]
"имя_параметра 11”=значение11
"имя_параметра12"=значение12
. . .
"имя_параметрэ1N" =значение1N

[имя_раздела2]
"имя_параметра21"=значение21
"имя_параметра22"=значение22
. . .
"имя_параметра2N"=значение2N

[имя_разделаL]
"имя_ларametraL1"=значениеL1
"имя_параметраL2"=значениеL2
. . .
"имя_параметраLN"=значениеLN
```

Значения параметров могут принадлежать одному из трех типов:

строковый — значения этого типа являются просто строкой символов, заключенной в кавычки;

DWORD — для записи значения этого типа используется формат dword: XXXXXXXX. Вместо X записываются шестнадцатеричные цифры. Обычно параметры типа DWORD имеют значение либо 0, либо 1. В этих случаях для задания значения требуется записать либо dword:00000000, либо dword:00000001;

двоичный — для записи значения этого типа используется формат hex: XX,XX,XX, … Вместо XX записываются шестнадцатеричные цифры; пары таких цифр разделяются запятой. Например, для задания значения af 00 01 00 следует записа hex: af,00,01,00.

Кроме того, в реестре могут быть установлены параметры по умолчанию (defai Чтобы присвоить какое-то значение параметру по умолчанию, необходимо г сто записать в reg-файле следующее выражение:

@=значение

Рассмотренные выше записи reg-файла добавляют, а не перезаписывают записи в реестре. Чтобы удалить раздел в реестре, необходимо в reg-файле перед его и нем в квадратных скобках поставить символ «минус», как в следующем примере

```
[-HKEY_LOCAL_MACHINE\Software]
```

Чтобы удалить параметр, следует присвоить ему символ «минус»: "имя_параметра"= -

Ниже в качестве примера приведено содержимое reg-файла, с помощью которс устанавливается начальная веб-страница, загружаемая в браузер Internet Explorer:

```
REGEDIT4
[HKEY_CURENT_USER\SOFTWARE\Microsoft\Internet Explorer\Main]
"Start Page"="http://www.myweb.ru"
```

Читать, создавать и удалять записи в реестре можно с помощью специальных методов объекта Wscript.Shell, входящего в состав WSH:

- RegRead() — возвращает запись реестра или значение параметра;
- RegWrite() — создает новую запись в реестре или изменяет значение параметра уже существующей записи;
- RegDelete() — удаляет запись реестра или параметр.

Применение этих методов имеет следующий синтаксис:

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

Метод RegWriteQ принимает в качестве параметра строку, содержащую имя раздела реестра, за которым указывается имя параметра. Все элементы имени раздела разделяются двойными обратными слэшами. Например, в строке

"HKEY_CURENT_USER\\Myreg\\myparam" последний элемент myparam является именем параметра, а не раздела реестра.

Метод RegWrite() принимает три параметра, из которых последний не является обязательным. Первый параметр — строка, содержащая имя раздела или имя параметра. Если эта строка заканчивается двойным слэшем, то подразумевается, что последний элемент строки — имя раздела, в противном случае — имя параметра в разделе. Второй параметр метода RegWrite() представляет значение параметр раздела. Если имя параметра раздела не указано, то подразумевается параметр по умолчанию. Третий, необязательный параметр метода RegWrite() задает тип параметра в разделе реестра и представляет собой "REG_DWORD" или "REG_BINARY", соответственно для типов DWORD и двоичного. Если тип не указан, то подразумевается строковый тип.

Примеры

```
var Myshell = new ActiveXObject("WScript.Shell")
/* Создание подраздела Myreg в разделе HKEY_CURENT_USER и присвоение параметру по умолчанию значения "значение": */
Myshell.RegWrite("HKEY_CURENT_USER\\Myreg\\", "значение")

/• Создание в разделе HKEY_CURENT_USER\ Myreg строкового параметра myparaml и присвоение ему значения "некоторая строка": */
Myshell.RegWrite("HKEY_CURENT_USER\\Myreg\\myparaml", "некоторая строка")
/* Создание в разделе HKEY_CURENT_USER\ Myreg двоичного параметра myparam2 и присвоение ему значения 5: */
Myshell.RegWrite("HKEY_CURENT_USER\\Myreg\\myparam2",
5,"REG_BINARY")
/* Создание в разделе HKEY_CURENT_USER\ Myreg параметра myparam3 типа DWORD и присвоение ему значения 3: •/
Myshell.RegWrite("HKEY_CURENT_USER\\Myreg\\myparam3", 3,
"REG_DWORD")
```

Метод RegDeleteQ принимает в качестве параметра строку, содержащую имя раздела или параметра. Если эта строка заканчивается двойным слэшем, то подразумевается, что последний элемент строки — имя раздела, в противном случае - имя параметра в разделе.

Примеры

```
var Myshell = new ActiveXObject("WScript.Shell")
/* Удаление в разделе HKEY_CURENT_USER\ Myreg параметра myparaml: •/
Myshell.RegWrite("HKEY_CURENT_USER\\Myreg\\myparaml")
/• Удаление подраздела Myreg в разделе HKEY_CURENT_USER: •/
Myshell.RegDelete("HKEY_CURENT_USERW\\Myreg\\")
```

Вместо полных имен корневых разделов реестра можно использовать их сокращенные обозначения. Например, вместо HKEY_CURENT_USER можно писать HKCU.

Пример программы, делающей запись в реестре, которая устанавливает стартовую веб-страницу.

```
var  Myshell  =  new ActiveXObject("WScript.Shell")
var mystartpage =  http://www.myweb.ru       //  нужная  страница
/* Проверяем,  какая  веб-страница  является  начальной:   */
startpage = Myshell. RegWrite( " HKCU\\ SOFTWARE \\Microsoft\\Internet
Explorer\\Main\\Start  Page")
 if   (startpage   != mystartpage)                    //  если другая
Myshell. RegWrite("HKCU\\ SOFTWARE\\ Microsoft\\ Internet\\ Explorer \\Main
\\StartPage",   mystartpage)
```

Аналогичным образом можно создать записи в реестре, запускающие приложения.

Задание

Выполнить предлагаемые коды программ.

ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ПОСТРОЕНИЯ КОМПИЛЯТОРА

ЛАБОРАТОРНАЯ РАБОТА №7 (2 часа)
Тема **Структура компилятора**

Транслятор – программа, допускающая в качестве входа программу на исходном языке, а в качестве выхода – другую версию, написанную на языке, который называется объектным. Это машинный язык некоторой вычислительной машины, что дает возможность сразу же выполнить программу. Существует довольно условное деление трансляторов на ассемблеры и компиляторы, транслирующие соответственно языки низкого и высокого уровней.

Компиляторы составляют существенную часть программного обеспечения ЭВМ. Построение компилятора как единого целого – нелегкое и трудоемкое занятие. Поэтому лучше рассматривать процесс компиляции как взаимодействие небольших процессов. Выбор таких подпроцессов для каждого конкретного компилятора может зависеть от особенностей обрабатываемого языка, но в любом случае его лучше сделать с учетом соответствующей теории построения компиляторов.

Основные этапы компиляции – лексический, синтаксический и семантический анализ, оптимизация и генератор кода. Эти три блока имеют до-

ступ к общему набору таблиц, куда можно помещать долговременную и глобальную информацию о программе. Например, таблица имен (называемая таблицей идентификаторов, или таблицей символов), в которой накапливается информация о каждой переменной или идентификаторе). Связи между блоками и таблицами показаны на рис.1. Далее работа блоков описывается более детально.

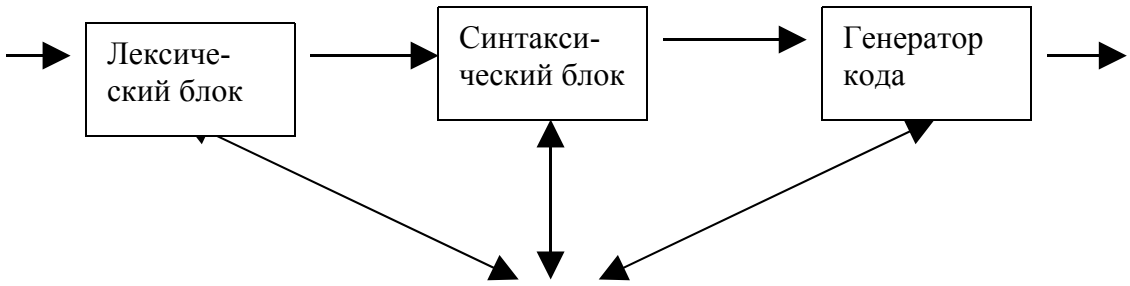


Рис.1.

Основная задача лексического анализа – разбить входной текст, состоящий из последовательности одиночных символов, на последовательность слов или лексем, то есть выделить эти слова из непрерывной последовательности символов. Все символы входной последовательности с этой точки зрения разделяются на принадлежащие каким-либо лексемам и на разделяющие лексемы (разделители).

Обычно все лексемы делятся на классы – числа, идентификаторы, строки. Отдельно выделяют ключевые слова и символы пунктуации (символы-ограничители). Ключевые слова – это некоторое конечное подмножество идентификаторов. В части языков смысл лексемы может зависеть от ее контекста, и провести лексический анализ в отрыве от синтаксического невозможно. С точки зрения дальнейших фаз анализа лексический анализатор выдает информацию двух сортов: для синтаксического анализатора существенна информация о последовательности классов лексем, ограничителей и ключевых слов, а для контекстного – о конкретных значениях отдельных лексем.

Общая схема работы лексического анализатора следующая. Сначала выделяем отдельную лексему, используя символы-разделители. Если выделенная лексема – ограничитель, то он (точнее, его признак) выдается как результат лексического анализа.

Ключевые слова – как правило, выделенные идентификаторы – распознаются следующим образом. Возможны два основных способа их выделения либо очередная лексема сначала диагностируется на совпадение с каким-либо ключевым словом и в случае неуспеха делается попытка выделить лексему из какого-либо класса, либо, наоборот, после выборки лексемы идентификатора требуется заглянуть в таблицу ключевых слов для сравнения. Поиск ключе-

вых слов можно вести в основной таблице имен (в которую до начала лексического анализа загружают ключевые слова), либо в отдельной таблице. В первом случае все ключевые слова непосредственно встраивают в конечный автомат лексического анализатора, во втором – конечный автомат содержит только разбор идентификаторов. Если выделенная лексема принадлежит какому-либо другому классу лексем (число, строка и т.д.), выделяется признак класса лексемы, а значение лексемы сохраняется.

Лексический анализатор может работать как самостоятельная фаза трансляции или как подпрограмма, действующая по принципу "дай лексему". В первом случае выходом является файл лексем, во втором лексема выдается при каждом обращении к анализатору (при этом тип лексемы зачастую возвращается как значение функции "лексический анализатор", а значение передается через глобальную переменную). Лексический анализатор может просто выдавать значение каждой лексемы, и тогда построение таблицы переносится на более поздние фазы; он может также самостоятельно строить таблицы объектов (идентификаторов, строк, чисел и т.д.). В этом случае в качестве значения лексемы выдается указатель на вход в соответствующую таблицу.

Основная задача синтаксического анализа – разбор структуры программы. Этот блок переводит последовательность лексем, построенную лексическим блоком, в другую последовательность, более непосредственно отражающую порядок, в котором по замыслу программиста должны выполняться операции в программе. Например, если имеется запись A+B*C, подразумевается, что числа, представленные идентификатором B и C, будут перемножены и к результату будет прибавлено число, представленное идентификатором A. Указанное выражение можно перевести так: УМНОЖ(B, C, R1) СЛОЖ(A, R1, R2),где УМНОЖ(B, C, R1) интерпретируется как "умножить B на C и заслать результат в R1, а СЛОЖ(A, R1, R2) – как "сложить A и R1 и результат заслать в R2 ".

Таким образом, пять лексем, выданных лексическим блоком, преобразуются в две новые единицы, описывающие то же действие. Эти единицы называются атомами и образуют выход синтаксического блока.

Преимуществом здесь является то, что последовательность атомов отражает порядок, в котором должны выполняться действия. Основной формализм синтаксического анализа – контекстно-свободные грамматики. Результат синтаксического анализа – синтаксическое дерево со ссылками на таблицу имен. На этом этапе обнаруживаются ошибки, связанные со структурой программы.

На этапе контекстного анализа выявляются зависимости между частями программы, которые не могут быть описаны контекстно-свободным синтаксисом. Эту часть работы компилятора называют семантической обработкой. Семантика идентификатора, например, может включать его тип, если это массив- его размерность. Один из видов семантической обработки включает занесение в таблицу имен свойств идентификаторов по мере их выявления. На данном этапе выполняется анализ областей видимости, соответствия параметров, меток и др. Основной формализм – атрибутные грамматики. В процессе контекстного анализа могут быть обнаружены ошибки, связанные с непра-

вильным использованием объектов.

Затем программа с целью оптимизации и удобства генерации может быть переведена во внутреннее представление. Фаз оптимизации может быть несколько.

Генерация кода – последняя фаза трансляции. Результат ее – либо ассемблерный, либо объектный модуль. В процессе генераций кода могут выполняться некоторые локальные оптимизации – такие как распределение регистров, выбор длинных и коротких переходов, учет стоимости команд при выборе их конкретной последовательности. Для генерации кода разработаны различные методы(таблицы решений, сопоставление образцов, включающее динамическое программирование, различные синтаксические методы).

Те или иные фазы транслятора могут либо отсутствовать вовсе, либо объединяться. В простейшем случае однопроходного транслятора нет явной фазы генерации промежуточного представления и оптимизации, остальные фазы объединены в одну, причем нет и явно построенного синтаксического блока.

Задание

1. Представить блок-схему работы лексического блока.

2. Составить алгоритм разбиения цепочки символов на лексемы. В качестве символов разделителей использовать пробел, табуляцию, знаки арифметических операций, запятую, точку, скобку открывающую, скобку закрывающуюся, точку с запятой.

ЛАБОРАТОРНАЯ РАБОТА №8 (4 часа)
Тема **Конечные автоматы**

Теория автоматов лежит в основе теории построения компиляторов. Под автоматом подразумевают не реально существующее устройство, а некоторую математическую модель, свойства и поведение которой можно изучать и которую можно имитировать с помощью программы на реальной, вычислительной машине. Конечные автоматы (КА) обладают рядом свойств, которые позволяют легко использовать их при решении некоторых задач компиляции. Это следующие свойства:

1.Конечный автомат может решать легкие задачи компиляции. В частности, лексический блок зачастую строится на основе конечного автомата.

2.Поскольку при моделировании конечного автомата на ЭВМ обработка одного символа требует небольшого количества операций, программа работает быстро.

3.Моделирование конечного автомата требует фиксированного объема памяти, что упрощает проблемы, связанные с управлением памятью.

4.Существует ряд теорем и алгоритмов, позволяющих конструировать и упрощать конечные автоматы, предназначенные для тех или иных целей.

Имеются различные определения конечного автомата. Общим в них является то, что они моделируют вычислительные устройства с фиксированным и конечным объемом памяти, которые читают последовательности входных символов, принадлежащих некоторому конечному множеству. Различия в определениях связаны с тем, что автоматы делают на выходе. Наиболее простыми являются автоматы, единственный выход которых – является указание на то, допустима или нет данная входная цепочка (последовательность символов). Допустимая – правильно построенная или синтаксически правильная цепочка. Такие автоматы называют конечными распознавателями.

Примером задачи распознавания может служить проверка нечетности числа единиц в произвольной цепочке, состоящей из нулей и единиц. Соответсвующий конечный автомат, называемый контроллером нечетности, будет допускать все цепочки, содержащие нечетное количество единиц, и отвергать цепочки с четным их числом.

На вход конечного автомата подается цепочка символов из конечного множества, называемого входным алфавитом автомата и представляющего собой совокупность символов, для работы с которыми он предназначен. Как допускаемые, так и отвергаемые цепочки состоят только из символов входного адфавита. Символы, не принадлежащие к входному алфавиту, нельзя подавать на вход автомата. Входной автомат контроллера нечетности состоит из двух символов – 0 и 1.

В каждый момент автомат имеет дело с одним входным символом, а информацию о предыдущих символах входной цепочки сохраняет с помощью конечного множества состояний. Согласно этому представлению автомат помнит о прочитанных раннее символах только то, что при обработке он перешел в некоторое состояние, которое является памятью автомата о прошлом.

Контроллер нечетности должен запоминать, четное или нечетное количество единиц ему встретилось. Поэтому множество состояний автомата содержат два состояния, называемые ЧЕТ и НЕЧЕТ.

Одно из состояний должно быть выбрано в качестве начального, в котором автомат начинает работу. Начальным состоянием контроллера нечетности будет ЧЕТ, так как на первом шаге число прочитанных единиц равно нулю, а это число четное.

При чтении очередного входного символа состояние автомата меняется, причем новое зависит от входного символа и текущего состояния. Такое изменения состояния называется переходом. Новое состояние может совпадать со старым.

Работу автомата можно описать с помощью функции δ, называемой функцией перехода. По текущему состоянию $S_{тек}$ и текущему входному символу x она дает новое состояние автомата $S_{нов}$

Определим функцию переходов контроллера нечетности следующим образом:

δ (ЧЕТ, 0)=ЧЕТ

δ (ЧЕТ, 1)=НЕЧЕТ
δ (НЕЧЕТ, 0)=НЕЧЕТ
δ (НЕЧЕТ, 1)=ЧЕТ

Некоторые состояния автомата выбираются в качестве допускающих или заключительных. Если автомат, начав работу в начальном состоянии, при прочтении всей цепочки переходит в одно из допускающих состояний, говорят, что эта входная цепочка допускается автоматом. Если последнее состояние автомата не является допускающим, говорят, что автомат отвергает цепочку. Контроллер нечетности имеет единственное допускающее состояние – НЕЧЕТ.

Конечный автомат задается:

1) конечным множеством входных символов;

2) конечным множеством состояний;

3) функцией перехода F, которая каждой паре, состоящей из входного символа и текущего состояния, приписывает некоторое новое состояние;

4) состоянием, выделенным в качестве начального;

5) подмножеством состояний, выделенных в качестве допускающих или заключительных.

Один из удобных способов представления конечных автоматов – таблица переходов. Информация размещается в таблице переходов в соответствии со следующими соглашениями.

1. Столбцы помечены входными символами.

2. Строки помечены символами состояний.

3. Элементами таблицы являются символы новых состояний, соответствующих входным символам столбцов и состояниям строк.

4. Первая строка помечена символом начального состояния.

5. Строки, соответствующие допускающим состояниям, помечены справа нулями.

Таблица переходов для контроллера нечетности представлена на рис.2.

	0	1	
ЧЕТ	ЧЕТ	НЕЧЕТ	0
НЕЧЕТ	НЕЧЕТ	ЧЕТ	1

Рис.2.

Она задает конечный автомат у которого:

входное множество={0, 1},

множество состояний = {ЧЕТ, НЕЧЕТ},

переходы δ (ЧЕТ, 0)=ЧЕТ, и т.д.,

начальное состояние = ЧЕТ,

допускающие состояния={НЕЧЕТ}.

На рис.3 представлен еще один пример конечного автомата. Переход ав-

томата из состояния $S_{тек}$ в состояние $S_{нов}$ при чтении входного символа a будем обозначать так:

$$S_{тек} \xrightarrow{a} S_{нов}$$

	x	y	z	
1	1	3	4	1
2	2	1	3	0
3	2	4	4	1
4	3	3	3	0

Входное множество –{x, y, z}

Множество состояний ={1, 2, 3, 4}

Функция переходов δ (l, x)=1; δ (1,y)=3 и т.д.

Начальное состояние – 1.

Допускающее состояние {1, 3}.

Рис.3.

Входная цепочка допускается автоматом, если он, начав работу в начальном состоянии, при прочтении всей цепочки переходит в одно из допускающих состояний. Если последнее состояние автомата не является допускающим, говорят, что автомат отвергает цепочку.

Для описанного автомата цепочка xyzz допустима, так как

$$1 \xrightarrow{x} 1 \xrightarrow{y} 3 \xrightarrow{z} 4 \xrightarrow{z} 3$$

и 3 является допускающим состоянием, тогда как цепочка zyx отвергается, потому что

$$1 \xrightarrow{z} 4 \xrightarrow{y} 3 \xrightarrow{x} 2$$

и 2 – отвергающее состояние.

Для построения конечного автомата необходимо знать множество всех цепочек, распознаваемых автоматом, которое обычно называют регулярным.

Регулярное множество в алфавите T (T – конечнь̆й алфавит) определяется рекурсивно следующим образом:

1) {} (пустое множество) – регулярное множество в алфавите T;

2) {*a*} – регулярное множество в алфавите T для каждого *a* из алфавита T;

3) {*e*} –регулярное множество в алфавите T (*e* – пустая цепочка);

4) если P и Q – регулярные множества в алфавите T, то таковы и множества:

а) PUQ (объединение),

б) P*Q (конкатенация) pq p из множества P, q из множества Q,

в) (P)*(итерация) (P)*={e}UPUP*PU...

5) ничто другое не является регулярным множеством в алфавите T. Множество в алфавите T регулярно тогда и только тогда, когда оно либо {},

{e}, {a}, а принадлежит множеству T, либо его можно получить применением конечного числа операций (объединение, конкатенация, итерация).

Примеры регулярных выражений и обозначаемых ими множеств.

Идентификатор=Буква (БукваUцифра)*

Буква={ a,b...z}

Цифра={0,1 ... 9}

Для каждого регулярного множества можно найти по крайней мере одно регулярное выражение, обозначающее это множество. И обратно: для регулярного выражения можно построить регулярное множество, обозначаемое этим выражением. Для каждого регулярного множества существует бесконечное число обозначающих его регулярных выражений.

Задание

1. Построить конечный автомат для одного из представленных ниже множеств цепочек из 0 и 1:

a) между вхождениями единиц четное число нулей,
b) за каждым вхождением пары единиц следует нуль,
c) каждый третий символ – единица,
d) все входные цепочки, начинающиеся 0 и заканчивающиеся 1,
e) все входные цепочки, содержащие в точности три 1,
f) все цепочки, в которых перед и после каждой 1 стоит 0,
g) все цепочки, не содержащие ни одной 1.

2. Построить конечный автомат, который будет распознавать следующие зарезервированные слова Паскаля: STRING, INTEGER, IN, INLINE.

3. Найти самую короткую цепочку, распознаваемую автоматом, изображенном на рис.5. Найти четыре другие цепочки, распознаваемые этим автоматом. Найти четыре цепочки, которые отвергаются этим автоматом.

	0	1	
A	D	A	0
B	A	C	0
C	A	F	0
D	B	C	0
E	B	C	1
F	E	A	1

Рис.5

4. Построить конечный автомат, который распознает химические формулы, составленные из восьми элементов: H, C, N, O, SI, S, CL и SN. Элементы в формулах разделяются запятыми. Они могут появляться в любом порядке и в любых сочетаниях. Формулы не обязательно представляют реально существующие соединения. Вот несколько их образцов: H2,O; O,H7; SN,S,O4; CL; N,H7,C7,H5,O2.

5. Построить таблицу переходов конечного автомата, входной алфавит которого состоит из 31 символа: ОДИН, ДВА, ТРИ, … ДЕВЯТЬ, ДЕСЯТЬ, ОДИННАДЦАТЬ … ДЕВЯТНАДЦАТЬ, ДВАДЦАТЬ, ТРИДЦАТЬ … ДЕВЯНОСТО, СТО, ДВЕСТИ … СТА, СОТ; пусть выходной алфавит состоит из 10 символов 0, 1. 2, 3,…9 и пусть он допускает в качестве входа словесную запись любого числа от 1 до 999 и переводит вход в эквивалентную цифровую запись.

ЛАБОРАТОРНАЯ РАБОТА №9 (4 часа)
Тема **Получение минимального автомата**

Произвольный конечный автомат можно превратить в эквивалентный ему минимальный, выбрасывая недостижимые состояния и объединяя эквивалентные состояния.

Применительно к конечным распознавателям, назначение которых – допускать цепочки, эквивалентность состояний можно определить следующим образом. Состояние s конечного распознавателя M эквивалентно состоянию t конечного распознавателя N тогда и только тогда, когда автомат M, начав работу в состоянии s, будет допускать в точности те же, что и автомат N, начавший работу в состоянии t.

Если два состояния s и t одного автомата эквивалентны, то автомат можно упростить, заменив в таблице переходов все вхождения имен этих состояний каким-нибудь новым именем, а затем удалив одну из двух строк, соответствующих s и t. Например, состояния 4 и 5 КА, изображенные на рис.7, явно имеют одинаковые функции, так как оба являются допускающими, оба переходят в состояние 2 при чтении входного символа a и оба переходят в состояние 3 при чтении b.

	a	b	
1	1	4	0
2	3	5	1
3	5	1	0
4	2	3	1
5	2	3	1

Рис.7.

Поэтому можно объединить состояния 4 и 5 в одно состояние и назвать его X. Получается упрощенная таблица состояний, представленная на рис.8.

	a	b	
1	1	4	0

2	3	5	1
3	5	1	0
X	2	3	1

Рис.8.

Обычно эквивалентность состояний менее очевидна. Существует следующее ее определение: два состояния эквивалентны тогда и только тогда, когда не существует различающей их цепочки. Понятие «эквивалентность состояний» является отношением в математическом смысле; это отношение рефлексивно (каждое состояние эквивалентно себе), симметрично (если s эквивалентно t, то t эквивалентно s) и транзитивно (если s эквивалентно t, а t эквивалентно u, то s эквивалентно u).

Метод проверки эквивалентности состояний основывается на следующем факте. Состояния s и t эквивалентны тогда и только тогда, когда выполняются два условия:

1) условие подобия – состояния s и t должны быть либо оба допускающие, либо оба отвергающие,
2) условие преемственности – для всех входных сисмволов состояния s и t должны переходить в эквивалентные состояния, т.е. их преемники эквивалентны.

Эти условия выполняются тогда и только тогда, когда s и t 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 s и t в неэквивалентные. Поэтому x с приписанной к нему цепочкой, различающей эти новые состояния, образует цепочку, различающую s и t.

Условия 1 и 2 можно использовать в общем методе проверки на эквивалентность произвольной пары состояний. Для этого строятся таблицы эквивалентности состояний. Рассмотрим КА, таблица переходов которого изображена на рис.9, и выявим эквивалентные состояния.

Проверим на эквивалентность состояния 0 и 7. Таблица эквивалентности состояний содержит по одному столбцу для каждого входного символа, – для y и z. Строки будут добавляться в ходе проверки. Первоначально такая таблица имеет вид, представленный на рис.10: имеется одна строка, которая помечена парой состояний, подвергаемых проверке, т.е. парой 0, 7.

	y	z	
0	0	3	0
1	2	5	0
2	2	7	0
3	6	7	0
4	1	6	1

5	6	5	0
6	6	3	1
7	6	3	0

Рис.9.

Сначала необходимо попытаться продемонстрировать неэквивалентность состояний, показав, что нарушается условие подобия. В данном случае это условие выполняется, так как оба состояния являются отвергающие.

	y	z
0, 7		

Рис.10.

Может быть будет нарушено условие преемственности? Чтобы исследовать эту возможность, рассмотрим, как действует на отдельную пару состояний каждый входной символ, и запишем результат в соответствующую ячейку таблицы. Так как состояние 0, 7 под действием входного символа y переходит в состояние 0 и 6 соответственно, то 0, 6 записывается в столбец таблицы, соответствующий символу y. Так как оба состояния 0 и 7 переводятся символом z в состояние 3, то 3 записывается в столбец для z. Таблица эквивалентности примет вид, представленный на рис.11.

Чтобы нарушалось условие преемственности, должны быть неэквивалентны либо состояние 0 и 6, либо состояния 3 и 3. Так как каждое состояние эквивалентно само себе, состояния 3 и 3 эквивалентны автоматически.

	y	z
0, 7	0, 6	3

Рис. 11.

Чтобы исследовать на эквивалентность состояния 0 и 6, к таблице эквивалентности состояний добавляется новая строка и помечается новой парой 0, 6. Для нее повторяется весь процесс, описанный для пары 0,7. Сперва проверяется условие подобия для пары 0, 6. Оказывается, что 0 и 6 неэквивалентны, так как 6 – допускающее, а 0 – отвергающее состояние. Проверка показала, что состояния 0 и 7 неэквивалентны. Таблицу эквивалентности состояний можно использовать для построения различающей цепочки. Строка 0,6 появилась как результат применения входного символа y к паре 0,7, поэтому y является различающей цепочкой.

Проверим на эквивалентность пару состояний 0, 1. Построение таблицы эквивалентности начинается с пары 0, 1. Эти состояния подобны, поэтому вычисляется результат применения к ним каждого входного символа; полученные состояния помещаются в таблицу. Надежды на эквивалентность 0, 1 оправдаются, если будет установлена эквивалентность пар, помещенных в таблицу, 0, 2 и 3, 5. В таблицу добавляется строка для каждой из этих пар. Результат изображен на рис.12.

y z

0, 1	0, 2	3, 5

	y	z
0, 1	0, 2	3, 5
0, 2		
3, 5		

Рис.12.

Обратившись к строке 0, 2, можно определить, что эти состояния подобны, поэтому можно вычислить следующие пары, чтобы проверить условие преемственности. Результат отображен на рис.13. Из двух новых элементов один дает новую строку, а именно пара 3, 7. Другой элемент уже имеется в таблице, и нет надобности его проверять. Далее по списку проверяется пара 3, 5. Состояния 3, 5 подобны. Вычисляем пары 6, 6 и 5, 7. Так как состояние 6 эквивалентно самому себе, единственной новой строкой в таблице будет 5, 7. Таблица эквивалентности примет вид, представленный на рис.14.

	y	z
0, 1	0, 2	3, 5
0, 2	0, 2	3, 7
3, 5		

Рис.13.

	y	z
0, 1	0, 2	3, 5
0, 2	0, 2	3, 7
3, 5	6	5, 7
3, 7		
5, 7		

Рис.14.

Описанные процедуры повторяются для оставшихся пар – 3, 7 и 5, 7. По их окончании не удается получить ни одной новой пары неподобных состояний и ни одной пары, которую надо проверять на подобие. Окончательная таблица представлена на рис.15.

	y	z
0, 1	0, 2	3, 5
0, 2	0, 2	3, 7

3, 5	6	5, 7
3, 7	6	3, 7
5, 7	6	3, 5

Рис.15.

Различающие цепочки для состояний 0, 1 отсутствуют, поэтому эти состояния являются эквивалентными.

Алгоритм проверки эквивалентности двух состояний можно описать следующим образом.

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

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, и процедура оканчивается. Если они подобны, следует вычислить результат применения каждого входного символа к этой паре состояний и записать полученные пары состояний в соответствующие ячейки рассматриваемой строки.

3.Для каждого элемента, полученного на шаге 2, существуют три возможности. Если элементом таблицы является пара одинаковых состояний, для нее не требуется никаких действий. То же самое – если элементом таблицы является пара состояний, которые уже использовались как метки строк. Если же элемент таблицы – пара разных состояний, которая еще не использовалась как метка, для нее нужно добавить новую строку. Порядок состояний в паре не важен, и пары s, t и t, s считаются одинаковыми. После того, как произведены необходимые действия для каждой пары состояний в данной строке, выполняется следующий 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. На рис. 15 видно, что, кроме эквивалентности пары (0,1), которая подвергалась проверке, доказана эквивалентность пар (0,2), (3,5), (3,7), (5,7). По свойству транзитивности из эквивалентности пар состояний 0,2 и 0,1 и следует эквивалентность пары 1, 2. Таким образом, состояния 0, 1, 2 эквивалентны друг другу. Аналогично эквивалентны друг другу состояния 3, 5, 7.

Автомат можно упростить , объединив состояния 0, 1, 2 в состояние A, а 3, 5, 7 – в состояние B. Новые имена подставляются в таблицу переходов, лишние строки удаляются и получается более простой – эквивалентный, изображенному на рис.9 . Новый автомат представлен на рис.16.

	y	z	
A	A	B	0
B	6	B	0
4	A	6	1
6	6	B	1

Рис. 16.

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

Задание

1.Найти различающую цепочку для пары автоматов, изображенных на рис.17.

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

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

Рис.17.

2.Найти минимальную эквивалентную таблицу для каждого из автоматов, представленных на рис.18.

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

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

Рис.18.

3.Для автоматов с рис. 18 найти недостижимые состояния.

4.Найти недостижимые состояния автомата, представленного на рис.19.

Рис.19.

ЛАБОРАТОРНАЯ РАБОТА №10 (2 часа)
Тема **Построение таблиц идентификаторов**

В процессе работы компилятор хранит информацию об объектах программы в таблицах. При организации таблиц возникает проблема идентификации слов – распознавание имен идентификаторов и соотнесение их с соответствующими элементами таблиц. Существуют следующие методы идентификации слов: индексов, линейного списка, упорядоченного списка и расстановки. Рассмотрим сущность, недостатки и преимущества каждого из них.

Метод индексов. По входной строке (имя идентификатора) вычисляется индекс, который обеспечивает прямой доступ к элементу таблицы. Таблицу, организованную по этому методу, называют индексированной. Она аналогична одномерному массиву, причем индекс слова служит индексом компоненты массива.

При использовании этого метода число слов в таблице не должно быть большим; индекс должен легко вычисляться; объем множества слов фиксируется при построении компилятора. Преимущество его в том, что идентификация осуществляется с минимальными затратами времени.

Метод линейного списка – наиболее прямой метод идентификации слов, заключающийся в сравнении входного слова с элементами таблицы до тех пор, пока не произойдет совпадения (если оно возможно).Таблица легко расширяется путем добавления слов на свободные места. Недостаток метода – поиск по длинному списку занимает много времени.

Метод упорядоченного списка. Элементы таблицы упорядочены, – например, лексикографически, т. е. по алфавиту. Предположим, что заданная таблица содержит M элементов. Для предыдущего метода, чтобы найти слово, потребуется (M+1)/2 сравнений, а для слова, которого нет в таблице, – M сравнений. Воспользовавшись упорядоченностью списка, можно уменьшить число

сравнений до logM. Делается это так. Поиск начинают сравнением входного слова со словом, расположенным в середине списка. Если слова совпадают, поиск окончен. В противном случае сравнение показывает, где следует искать входное слово – до или после середины списка.

Дальнейший поиск выполняется в новом списке, который вдвое короче исходного. Поиск продолжается, пока слово не будет найдено или список будет содержать всего один элемент, неравный входному слову, т.е. поиск окончится неудачей.

Недостаток метода – неудобно расширять таблицу, а необходимость вычислить середину списка требует дополнительных затрат времени.

Метод расстановки. По входному слову вычисляется индекс. Он может быть одним и тем же для многих слов. Его можно использовать для нахождения указателей списка. Если указатель равен 0, то входное слово не принадлежит множеству. Иначе элемент таблицы указателей списков указывает на некоторый связанный список слов, индексы которых совпадают с вычисленным. Поиск в этом списке продолжается до тех пор, пока не произойдет совпадения входного слова с элементами списка, или не будет достигнут конец списка. В последнем случае множество не содержит входного слова, и это слово можно добавить, просто связав его с последним элементом списка.

Для реализации этого метода необходимо выбрать функцию расстановки – метод вычисления индекса. Обычно используют следующие способы: двоичный код слова представляют как одно число или как ряд чисел, которые каким-нибудь образом комбинируют, чтобы получилось одно число. Затем это число каким-либо способом уменьшают, чтобы получить индекс желаемого размера. Один из способов состоит в использовании остатка от деления числа на некоторую константу – желательно простое число. Другой метод получения индекса – возведение числа в квадрат и выделение средних двоичных разрядов.

Недостаток метода – затраты времени на вычисление индекса, преимущества – множество легко расширяемо; требует умеренных затрат памяти; сравнительно невелико число сравнений для поиска заданного слова.

Задание

1.Составить алгоритм для формирования и заполнения таблицы идентификаторов, поступающих в следующем порядке: ABCD, BACD, DAB.C, EL, SON, DELO, DUB, KON, OK, NO, LOO, GROM, SOBA. Выбрать и обосновать метод организации таблицы.

2.Составить алгоритм для поиска по таблице идентификаторов, созданной в пункте 1, и дополнить таблицу новым идентификатором в случае неудачного поиска; использовать методы идентификации:

а) метод расстановки со следующими функциями:

Rmod7, где R=код ASCII (второго символа)+код ASCII(третьего символа);

R*Rmod1 7, где R=сумма кодов ASCII(всех символов);

R*R, где R=сумма кодов ASCII(всех символов);

5 первых двоичных разрядов +11 последних двоичных разрядов;

R*R,где R=сумма кодов ASCII(всех символов);

5 первых двоичных разрядов (пропустить 5 двоичных разрядов)+5 последующих двоичных разрядов;

R*Rmod1 1, где R=сумма кодов ASCII(всех символов);

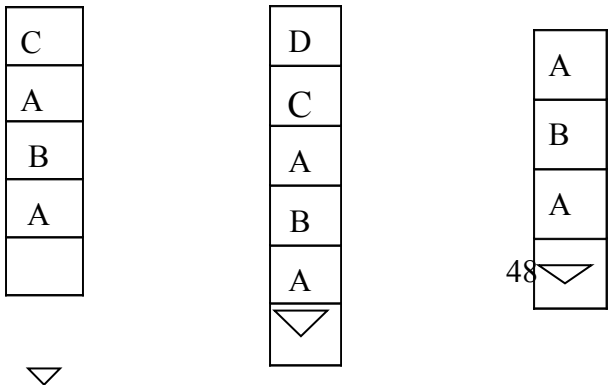
b) метод упорядоченного списка;

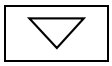
c) метод линейного списка.

ЛАБОРАТОРНАЯ РАБОТА №11 (2 часа)
Тема **Автоматы с магазинной памятью**

Конечный автомат может решать лишь такие вычислительные задачи, которые требуют фиксированного и конечного объема памяти. В компиляторе, однако, возникает множество задач, которые невозможно решить при таком ограничении, поэтому требуется модель более сложного автомата. Рассмотрим, например, задачу обработки скобок в арифметических выражениях. Арифметическое выражение может начинаться с любого количества левых скобок, и компилятор должен проверять, имеется ли в выражении такое же число соответствующих правых скобок. Каждый раз самая левая из скобок в выражении будет играть особую роль, так как каждая из таких ролей требует разного числа соответствующих правых скобок для завершения выражения. Иными словами, компилятор должен эффективно подсчитывать левые скобки, чтобы сбалансировать их. Разумеется, конечное множество состояний не годится для запоминания числа необходимых правых скобок, так как множество этих чисел бесконечно. Для систематического решения проблемы, связанной с выражениями, а также для решения многих других проблем компиляции необходимо использовать в компиляторе модели более мощных автоматов.

Чтобы получить более мощный автомат, память конечного автомата расширяется за счет дополнительного механизма хранения информации. Один из методов хранения информации – использование магазина или стека. Основная особенность магазинной памяти, с точки зрения работы с нею, состоит в том, что символы можно помещать в магазин и удалять из него по одному, причем удаляемый символ – всегда тот, который был помещен в магазин последним. Когда информация помещается в магазин, говорят, что она в него «вталкивается», когда удаляется из магазина, – что она «выталкивается» из него. Говорят, что информация, только что поступившая в магазин, находится в его верхушке , или наверху.



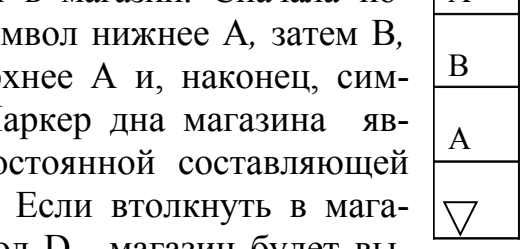


а *б* *в* *г*

Рис. 23.

.

На рис. 23 представлены магазины в различных состояниях. Рис.23,а – на дне магазина находится символ V, маркер дна магазина – специальный символ, который помечает начало, а наверху — символ С. Символы расположены в том порядке, в каком они поступали в магазин. Сначала поступил символ нижнее А, затем В, затем верхнее А и, наконец, символ С. Маркер дна магазина является постоянной составляющей магазина. Если втолкнуть в магазин символ D, магазин будет выглядеть так, как показано на рис. 23, б, где D — верхний символ магазина. Если же, наоборот, вытолкнуть из магазина верхний символ С, верхним символом окажется А, и магазин будет выглядеть, как показано на рис. 23, в. В обоих случаях изменениям подвергается только верх магазина, а остальные символы остаются неизменными. Маркер дна никогда не выталкивается из магазина. Так, если V — верхний символ магазина, как на рис. 23, г, это означает, что других символов в магазине нет. В этом случае говорят, что магазин пуст.



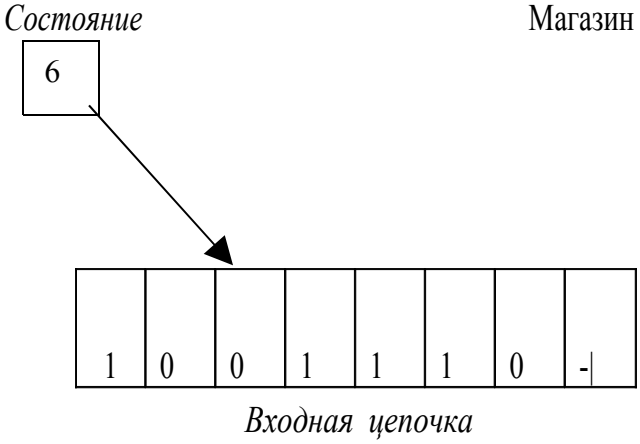


Рис. 24.

Одна из моделей автомата, где используется магазинный принцип организации памяти, – автомат с магазинной памятью (МП-автомат). В нем очень просто комбинируется память конечного автомата и магазинная память. МП-автомат может находиться в одном из конечного числа состояний и имеет магазин, куда он может помещать и откуда извлекать информацию.

Как и в случае конечного автомата, обработка входной цепочки осуществляется рядом мелких шагов. На каждом шаге действия автомата конфигурация его памяти может измениться за счет перехода в новое состояние, а также вталкивания символа в магазин или выталкивания из него. Однако в отличие от конечного автомата МП-автомат может обрабатывать один входной символ в течение нескольких шагов. На каждом шаге управляющее устройство автомата решает, пора ли закончить обработку текущего входного символа и получить, если это так, новый входной символ, либо продолжить обработку текущего символа на следующем шаге. На рис. 24 изображена одна из конфигураций, которая может возникнуть при обработке некоторым гипотетическим МП-автоматом входной цепочки 100110. Для большей наглядности входная цепочка изображена записанной в ячейках файла или ленты с указателем на входной символ, подвергающийся в данный момент обработке.

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. На рис. 24 информация, поступающая в управляющее устройство, такова: состояние 6, верхний символ магазина С, текущий входной символ 0.

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 могут быть следующими.

Операции над магазином:

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

Операция над состоянием:

1) перейти в заданное новое состояние;

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 вход-

ным 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.

На основе всего сказанного МП-автомат определяется следующими пятью объектами:

1) конечным множеством входных символов, включающим входит и концевой маркер;

2) конечным множеством магазинных символов, включающим маркер дна;

3) конечным множеством состояний, включающим начальное состояние;

4) управляющим устройством, которое каждой комбинации входного символа, магазинного символа и состояния ставит в соответствие выход или переход; переход, в отличие от выхода, заключается в выполнении операций над магазином, состоянием и входом, как уже было описано, операции, которые запрашивали бы входной символ после концевого маркера или выталкивали из магазина, а также вталкивали в него маркер дна, исключаются;

5) начальным содержимым магазина, которое представляет собой (при условии, что верхний символ считается расположенным справа) маркер дна, за которым следует (возможно, пустая) цепочка других магазинных символов.

МП-автомат называется МП-распознавателем, если у него два выхода — ДОПУСТИТЬ и ОТВЕРГНУТЬ. Говорят, что цепочка символов входного алфавита (исключая концевой маркер) допускается распознавателем, если под действием этой цепочки с концевым маркером автомат, начавший работу в своем начальном состоянии и с начальным содержимым магазина, делает ряд переходов, приводящих к выходу ДОПУСТИТЬ. В противном случае цепочка отвергается,

При описании переходов МП-автомата будем обозначать действия автомата словами ВЫТОЛКНУТЬ (или, для краткости, ВЫТОЛК), ВТОЛКНУТЬ (или ВТОЛК), СОСТОЯНИЕ, СДВИГ и ДЕРЖАТЬ, причем:

ВЫТОЛКНУТЬ означает вытолкнуть верхний символ магазина;

ВТОЛКНУТЬ (А), где А — магазинный символ, означает втолкнуть символ А в магазин;

СОСТОЯНИЕ(s), где s — состояние, означает, что следующим состоянием становится s;

СДВИГ означает, что текущим входным символом становится следую-

щий входной символ. В некоторых реализациях это может означать сдвиг указателя на входе;

ДЕРЖАТЬ означает, что текущий входной символ надо держать до следующего шага, т. е. оставить его текущим (в некоторых реализациях — оставить указатель на прежнем месте); если автомат содержит в точности одно состояние, слово СОСТОЯНИЕ будем опускать.

Применим МП-распознаватель к проблеме скобок. Каждый раз, когда встречается левая скобка, в магазин будет вталкиваться символ А. Когда будет обнаружена соответствующая правая скобка, символ А будет выталкиваться из магазина. Цепочка отвергается, если на входе остаются правые скобки, а магазин пуст (т. е. во входной цепочке есть лишние правые скобки) или если цепочка прочитана до конца, а в магазине остаются символы А (т. е. входная цепочка содержит лишние левые скобки). Цепочка допускается, если к моменту прочтения входной цепочки до конца магазин опустошается. Полное определение таково.

1. Входное множество {(,), —|}.
2. Множество магазинных символов {А, ∇ }.
3. Множество состояний {s}, где s — начальное состояние.
4. Переходы:

(,A,s = ВТОЛКНУТЬ(А), СОСТОЯНИЕ(s), СДВИГ
(, ∇ , s= ВТОЛКНУТЬ(А), СОСТОЯНИЕ(s), СДВИГ
),A, s= ВЫТОЛКНУТЬ, СОСТОЯНИЕ(s), СДВИГ
), ∇ ,s= ДЕРЖАТЬ
-|, A, s= ДЕРЖАТЬ
-|, ∇ , s=ДОПУСТИТЬ

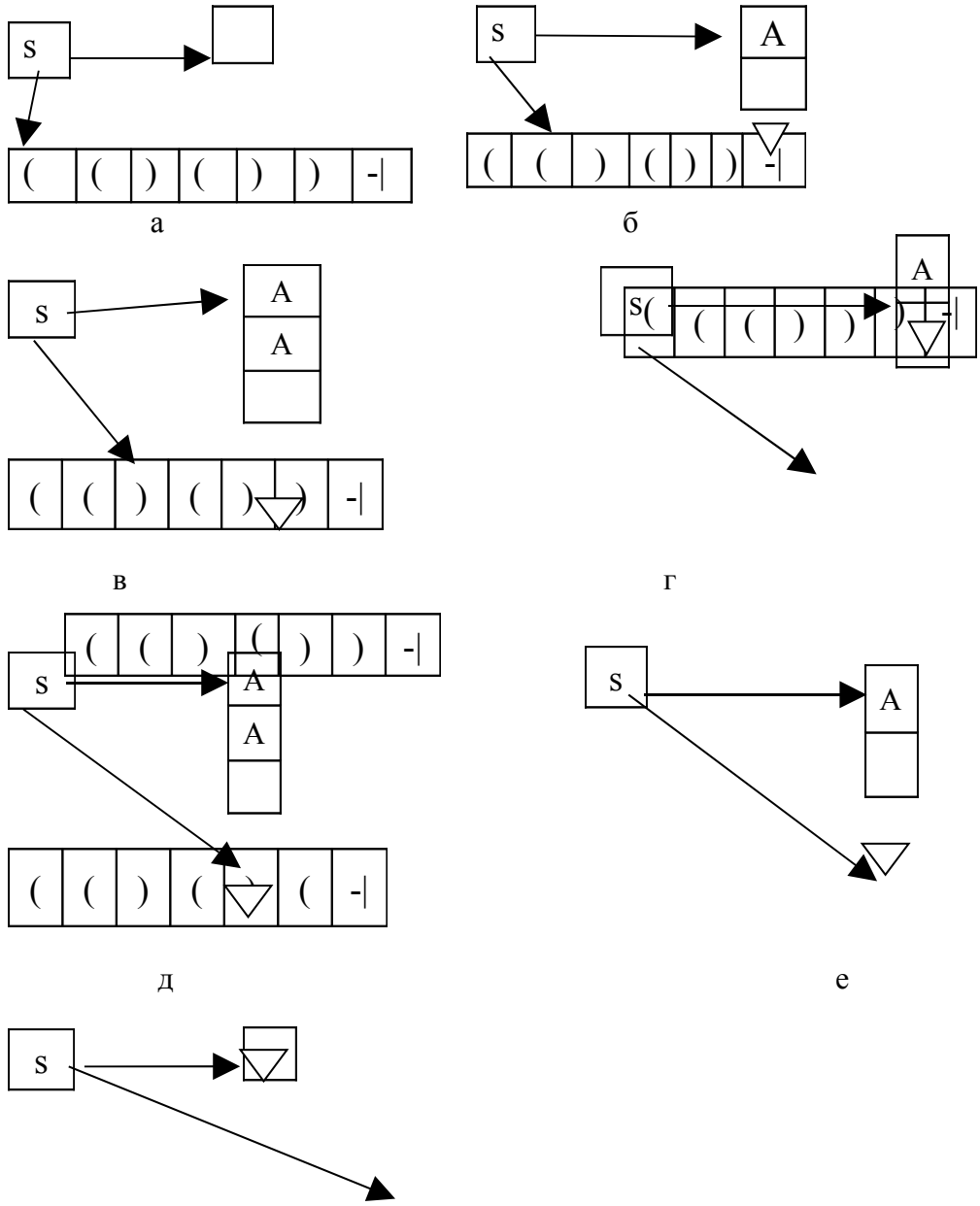
5. Начальное содержимое магазина – ∇

Работа этого МП-автомата изображена на рис. 26, где показан каждый шаг процесса обработки, начиная с начальной конфигурации (а) и кончая допускающей конфигурацией (ж). Такое изображение последовательности конфигураций МП-автомата требует много места, поэтому МП-автомат может быть представлен в более компактном виде, как на рис.25.

а:∇	[s]	(( ) ( )) -|
б:∇А	[s]	( ) ( )) -|
в :∇ АА	[s]	) ( )) -|
г :∇ А	[s]	( )) -|
д:∇ АА	[s]	)) -|
е :∇ А	[s]	) -|
ж:∇	[s]	-|
ДОПУСТИТЬ		

Рис.25.

Обрабатываемая цепочка (( ) ( ))



Допустить

Рис. 26.

Управляющее устройство этого автомата с одним состоянием можно представить в виде управляющей таблицы, как на рис. 27, где показаны действия автомата для каждого сочетания входного символа и верхнего символа магазина. Столбцы таблицы обозначены входными символами, а на пересечении строк и столбцов обозначены соответствующие им действия. Так как этот конкретный автомат имеет лишь одно состояние, информация о состоянии опущена.

Таблица такого вида (т. е. со столбцами для входных символов и

строками для символов магазина) является стандартным представлением МП-автоматов с одним состоянием.

( ) ┤

A	ВТОЛКНУТЬ (A) СДВИГ	ВЫТОЛК-НУТЬ СДВИГ	ОТВЕРГНУТЬ
¯V	ВТОЛКНУТЬ (A) СДВИГ	ОТВЕРГ-НУТЬ	ДОПУСТИТЬ

Рис. 27.

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 первой задачи указать цепочку длины, большей 3. 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, изображенным на рис.28. Для каждой из этих цепочек указать соответствующие последовательности конфигураций, допускающие эти цепочки.

ВТОЛКНУТЬ (A) СДВИГ	ОТВЕРГНУТЬ	ВЫТОЛКНУТЬ СДВИГ	ОТВЕРГНУТЬ	a
ВТОЛКНУТЬ (C) СДВИГ	ВЫТОЛКНУТЬ ДЕРЖАТЬ	ВТОЛКНУТЬ (A) СДВИГ	ОТВЕРГНУТЬ	b c ┤
ВТОЛКНУТЬ (B) ДЕРЖАТЬ	ВТОЛКНУТЬ (C) СДВИГ	ВЫТОЛКНУТЬ СДВИГ	ОТВЕРГНУТЬ	A
ВТОЛКНУТЬ (A) СДВИГ	ОТВЕРГНУТЬ	ОТВЕРГНУТЬ	ДОПУСТИТЬ	

B

C

▽

Начальное содержимое магазина ▽

Рис. 28.

4. Привести пример такого множество цепочек, которое может распознать МП-автомат с магазинным алфавитом, содержащим маркер дна магазина и еще два символа, но не может распознать никакой МП-автомат, у которого магазинный алфавит состоит из маркера дна магазина и еще одного символа.

5. Составить три допускаемые цепочки и соответствующие последовательности конфигураций для МП-автомата с одним состоянием, представленного на рис.29.

0 1 ┤

ЗАМЕНИТЬ (AA) СДВИГ	ВЫТОЛКНУТЬ СДВИГ	ОТВЕРГНУТЬ	A
ОТВЕРГНУТЬ	ОТВЕРГНУТЬ	ДОПУСТИТЬ	∇

Начальное содержимое магазина ▽

Рис. 29.

ЛАБОРАТОРНАЯ РАБОТА №12 (4 часа)

Тема **Нисходящие методы обработки языков с помощью МП-автомата**

Существует нисходящий подход к процессам обработки языков с помощью МП-автоматов, который позволяет распознать не любой КС-язык. Нисходящий подход налагает ограничения на вид входной грамматики.

Рассмотрим такие классы грамматик, для которых можно построить нисходящие МП-автоматов. К ним относятся так называемые s-грамматики, g-грамматики и LL(1)-грамматики.

Контекстно-свободная грамматика называется s-грамматикой (а также разделенной или простой) тогда и только тогда, когда выполняются следующие два условия

1. Правая часть каждого правила начинается терминалом.

2. Если два правила имеют совпадающие левые части, правые части должны начинаться различными терминальными символами.

Например, грамматика

1. <S>→a<T>

2. <S>→<T>b<S>

3. <T>→b<T>

4. <T>→ba

не s-грамматика, так как правая часть правила 2 начинается нетерминальным символом, то есть не удовлетворяет условию 1. Кроме того, правила 3 и 4 начинаются с одинакового терминального символа, что противоречит условию, задаваемому 2. Рассмотрим еще один пример грамматики

1.<S>→ab<R>

2. <S>→b<R>a<S>

3. <R>→a

4. <R>→b<S>.

Это s-грамматика, так как каждое правило имеет вид, удовлетворяющий условию 1, а правые части двух правил, имеющих в левой части нетерминал <S>, начинаются с различных терминальных символов, что и требуется в условии 2. То же самое можно сказать о правилах, в левой части которых стоит нетерминальный символ <R>.

Для s-грамматики МП-распознаватель с одним состоянием задается следующим образом.

1) Множество входных символов – это множество терминальных символов грамматики, расширенных концевым маркером (┤).

2) Множество магазинных символов состоит из маркера дна (▽ нетерминальных символов грамматики и терминалов, которые входят в правые части правил, за исключением тех, что занимают крайнюю левую позицию.

3) В начале магазин состоит из маркера дна и начального нетерминала.

4) Управление работой МП-автомата с одним состоянием описывается управляющей таблицей, строки которой помечены магазинными символами, столбцы – выходными символами, а элементы описываются ниже.

5) С каждым правилом грамматики сопоставляется элемент таблицы. Правило имеет вид <A>→bα, где <A> – нетерминал, b – терминал, а α – цепочка, состоящая из терминальных и нетерминальных символов. Этому пра-

вилу будет соответствовать элемент в строке <A> и столбце b ЗАМЕНИТЬ (α*), СДВИГ, где α*- цепочка α, записанная в обратном порядке. Если правило имеет вид <A>→b, то вместо ЗАМЕНИТЬ используется ВЫТОЛКНУТЬ.

6) Если магазинным символом является терминал b , то элементом таблицы в строке b и столбце b будет ВЫТОЛКНУТЬ, СДВИГ.

7) Элементом таблицы, который находится в строке маркера дна и столбце концевого маркера, является ДОПУСТИТЬ.

8) Элементы таблицы, не описанные ни одним из пунктов 5, 6 и 7, заполняются операцией ОТВЕРГНУТЬ.

Применяя перечисленные правила к s-грамматике, описанной в этом разделе, получим управляющую таблицу (рис.39).

a	b	┤	
ЗАМЕНИТЬ (<R>b) СДВИГ	ЗАМЕНИТЬ (<S>b<R>) СДВИГ	ОТВЕРГНУТЬ	<S>
ВЫТОЛК-НУТЬ сдвиг	ЗАМЕНИТЬ (<R>) СДВИГ	ОТВЕРГНУТЬ	<R>
ОТВЕРГНУТЬ сдвиг	ВЫТОЛКНУТЬ СДВИГ	ОТВЕРГНУТЬ	b
ОТВЕРГНУТЬ	ОТВЕРГНУТЬ	ДОПУСТИТЬ	▽

Рис.39.

Рассмотрим особый класс грамматик – более общий, чем класс s-грамматик. Все грамматики этого класса можно распознать с помощью нисходящих МП-распознавателей. Представители этого класса называются g-грамматики. Пример:

1. <S>→a<A><S>
2. <S>→b
3. <A>→c<A><S>
4. <A>→ε

Для g-грамматики введем понятия множества терминалов, "следующих за" данным нетерминалом, и множества выбора для данного правила.

Для КС-грамматики с начальным символом <S> и нетерминала <X> определим СЛЕД(<X>) как множество терминальных символов, которые могут следовать непосредственно за <X> в какой-либо промежуточной цепочке, выводимой из <S>. Это множество называется множеством "следующих за" <X> терминалов.

Анализируя последний пример, видим, что после нетерминала <A> могут идти только терминальные символы *a* и *b*: СЛЕД(<A>)={a,b}.

Множество выбора для данного правила определим следующим обра-

зом.

Если правило грамматики имеет вид <A>→bα, где b – терминальный символ, а α – цепочка, состоящая из терминальных и нетерминальных символов, то определим ВЫБОР(<A>→bα)={b}.

Если правило имеет вид <A>→ε, ВЫБОР(<A>→ε)=СЛЕД(<A>).

Если p – номер правила, то ВЫБОР(p) – множество выбора правила p.

Множество выбора правила просто содержит те входные символы, для которых соответствующий МП-автомат должен применить это правило.

Для грамматики, представленной выше, множество выбора будет следующим:

ВЫБОР(1)= ВЫБОР(<S>→a<A><S>)= { a };

ВЫБОР(2)=ВЫБОР(<S>→b)={b};

ВЫБОР(3)=ВЫБОР(<A>→c<A><S>)={c};

ВЫБОР(4)=ВЫБОР(<A>→ε)=СЛЕД(<A>)={a,b}.

Контекстно-свободная грамматика называется g-грамматикой тогда и только тогда, когда выполняются следующие два условия: правая часть каждого правила либо представляет собой пустую строку, либо начинается с терминального символа; множества выбора правила с одной и той же левой частью не пересекаются.

Первое условие просто утверждает, что все правила ограничиваются двумя видами, для которых определено понятие множества выбора. Второе условие указывает, что при построении управляющей таблицы соответствующего МП-автомата конфликтные ситуации не возникают. Наш пример удовлетворяет обоим условиям, так как правила имеют надлежащий вид и, кроме того, справедливы равенства

ВЫБОР(1) ∩ВЫБОР(2)={a} ∩{b}={}

ВЫБОР(3) ∩ ВЫБОР(4)={c} ∩ {a,b}={}.

Правила построения МП-распознавателя для g-грамматики такие же, как для s-грамматики, кроме 5-го. Его необходимо заменить на два следующих правила.

5.1. Правило грамматики применяется всякий раз, когда магазинный символ является его левой частью, а входной символ принадлежит его множеству выбора. Чтобы применить правило вида <A>→bα, где b – терминальный символ, а α – цепочка терминальных и нетерминаль-ных символов, используется переход ЗАМЕНИТЬ(α*), СДВИГ.

Если правило имеет вид <A>→b, вместо ЗАМЕНИТЪ(ε) можно воспользоваться операцией ВЫТОЛКНУТЬ.

Чтобы применить правило вида <A>→ε, используется переход

ВЫТОЛКНУТЬ, ДЕРЖАТЬ.

5.2.Если имеется ε-правило с нетерминалом <A> в левой части и элемент, соответствующий магазинному символу <A> и входному символу b, не был создан по правилу 5.1, то таким элементом может быть либо применение этого ε-правила, либо операция ОТВЕРГНУТЬ.

Построим управляющую таблицу для g-грамматики, представленной в

этом разделе (рис. 40).

LL1-грамматика – третий класс грамматик, для которых легко построить МП-распознаватель. '

КС-грамматика называется LL(1)-грамматикой тогда и только тогда, когда множества выбора правил с одинаковой левой частью не пересекаются.

Для определения множества выбора введем понятия ПЕРВ(α) и аннулирующей цепочки. По данной контекстно-свободной грамматике и промежуточной цепочке α, состоящей из символов этой грамматики, определим ПЕРВ(α) как множество терминальных символов, которые стоят в начале промежуточных цепочек, выводимых из α, т.е. являются их первыми символами.

a	b	c	┤	
ЗАМЕНИТЬ(<S><A>) СДВИГ	ВЫТОЛКНУТЬ СДВИГ	ОТВЕРГНУТЬ	ОТВЕРГНУТЬ	<S>
ВЫТОЛКНУТЬ ДЕРЖАТЬ	ВЫТОЛКНУТЬ ДЕРЖАТЬ	ЗАМЕНИТЬ(<S><A>) СДВИГ	ВЫТОЛКНУТЬ ДЕРЖАТЬ	<A>
ОТВЕРГНУТЬ	ОТВЕРГНУТЬ	ОТВЕРГНУТЬ	ДОПУСТИТЬ	▽

Начальное содержимое магазина ▽ <S>

Рис.40.

Цепочка α, состоящая из символов грамматики, называется аннулирующей тогда и только тогда, когда α=>ε.

Правило грамматики называется аннулирующим тогда и только тогда, когда его правая часть является аннулирующей, то есть его можно использовать для порождения пустой цепочки.

Определим множество выбора для правила произвольного вида следующим образом. Для данного правила <A>→α, где α– цепочка, состоящая из терминалов и нетерминалов, определим ВЫБОР(<A>→α=ПЕРВ(α), если α не аннулирующая, и ВЫБОР(<A>→α)=ПЕРВ(α) U СЛЕД(<A>), если α аннулирующая.

Рассмотрим LL(1)-грамматику и определим для нее множество выбора.

1.<S>→< A>b<B>

2.<S>→d

3.<A>→<C><A>b

4.<A>→<B>

5.<B>→c<S>d
6.<B>→e
7.<C>→a
8.<C>→fd
Начальный нетерминал <S>.

Определим для этой КС-грамматики множество ПЕРВ.

ПЕРВ (<A>b<B>)={a,b,c,f}
ПЕРВ (d)={d}
ПЕРВ(<C><A>b)={a,f}
ПЕРВ (<B>)={c}
ПЕРВ (c<S>d)={c}
ПЕРВ(e)={}
ПЕРВ(a)={a}
ПЕРВ (fd)={f}

В рассматриваемом примере правила 4 и 6 аннулирующие. Для каждого правила определим множество выбора.

ВЫБОР(1)=ПЕРВ(<A>b<B>)={a,b,c,f}
ВЫБОР(2)= ПЕРВ (d)={d}
ВЫБОР(3)=ПЕРВ(<C><A>b)={a,f}
ВЫБОР(4)=ПЕРВ(<B>) U СЛЕД(<A>)={c} U {b}={c,b}
ВЫБОР(5)= ПЕРВ(c<S>d)={c}
ВЫБОР(6)=ПЕРВ(ε) U СЛЕД(<B>)={} U {b,d,-+}
ВЫБОР(7)=ПЕРВ(a)={a}
ВЫБОР (8)= ПЕРВ(fd)={f}.

После нахождения множества выбора можно с уверенностью сказать, что это LL(1)-грамматика, так как множества выбора правил с одинаковой левой частью не пересекаются.

ВЫБОР(1) ∩ ВЫБОР(2)={a,b,c,f} ∩{d}={}
ВЫБОР(3) ∩ ВЫБОР(4)={a,f} ∩{c}={}
ВЫБОР(5) ∩ ВЫБОР(6)={c} ∩ {}={}
ВЫБОР(7) ∩ ВЫБОР(8)={a} ∩ {f}={}

После того как для всех правил найдены множества выбора, можно задать управляющую таблицу нисходящего МП-распознавателя. Правила построения МП-распознавателя для s-грамматики применимы для LL(1)-грамматики, если расширить правило 5.1.

5.1. Любое из правил применяется каждый раз, когда магазинный символ является левой частью правила, а входной символ принадлежит его множеству выбора. Чтобы применить правило вида <A>→bα, где b терминальный символ, а α – цепочка, состоящая из терминальных и нетерминальных символов, используется переход ЗАМЕНИТЬ(α*), СДВИГ.

Чтобы применить правило вида <A>→α, где α – цепочка, которая состоит из терминальных и нетерминальных символов и не начинается с терминала, используется переход ЗАМЕНИТЬ(α*), ДЕРЖАТЬ.

Если цепочка пустая, вместо операции ЗАМЕНИТЬ(ε) можно использовать операцию ВЫТОЛКНУТЬ.

5.2.Если для нетерминала <A> имеется аннулирующее правило и по правилу 5.1 не было создано элемента таблицы, соответствующего магазинному символу <A> и входному символу b, можно применить аннулирующее правило или отвергнуть входную цепочку. Построим управляю-

#1	#1	#1	#2	#1	ОТВЕРГ-НУТЬ
#3	#4	#4	ОТВЕРГ-НУТЬ	#3	ОТВЕРГ-НУТЬ
ОТВЕРГ-НУТЬ	#6	#5	#6	ОТ-ВЕРГ-НУТЬ	#6
#7	ОТВЕРГ-НУТЬ	ОТВЕРГ-НУТЬ	ОТВЕРГ-НУТЬ	#8	ОТВЕРГ-НУТЬ
ОТВЕРГ-НУТЬ	#7	ОТВЕРГ-НУТЬ	ОТВЕРГ-НУТЬ	ОТ-ВЕРГ-НУТЬ	ОТВЕРГ-НУТЬ
ОТВЕРГ-НУТЬ	ОТВЕРГ-НУТЬ	ОТВЕРГ-НУТЬ	#7	ОТ-ВЕРГ-НУТЬ	ОТВЕРГ-НУТЬ
ОТВЕРГ-НУТЬ	ОТВЕРГ-НУТЬ	ОТВЕРГ-НУТЬ	ОТВЕРГ-НУТЬ	ОТ-ВЕРГ-НУТЬ	ДОПУ-СТИТЬ

щую таблицу нисходящего МП-распознавателя (рис.41).

Совокупность правил построения распознавателя устанавливает следующий факт: если язык определяется LL(1) грамматикой, его можно распознать с помощью МП-автомата с одним состоянием, использующего расширенную магазинную операцию ЗАМЕНИТЬ. LL(1)-грамматика – универсальная грамматика по сравнению с s- и q-грамматиками.

a b c d f ┤

<S>

<A>

<B>

<C>

b

d

начальное содержимое магазина <S> ▽

#1 ЗАМЕНИТЬ(<B>b<A>), ДЕРЖАТЬ
#2 ВЫТОЛКНУТЬ, СДВИГ
#3 ЗАМЕНИТЬ(b<A><C>)ДЕРЖАТЬ
#4 ЗАМЕНИТЬ(<B>), ДЕРЖАТЬ
#5 ЗАМЕНИТЬ(d<S>), СДВИГ
#6 ВЫТОЛКНУТЬ, ДЕРЖАТЬ
#7 ВЫТОЛКНУТЬ, СДВИГ
#8 ЗАМЕНИТЬ(d), СДВИГ

Рис. 41.

Задание

1 . Определить тип грамматики. Построить МП-распознаватель. Варианты грамматик представлены ниже.

Вариант I.

1 .<S>→a<S><A>
2.<S>→b<A>
3.<S>→ε
4.<A>→c<A>
5.<A>→ε

Вариант 2.

1 .<S>→a<A><B><S>
2.<S>→b
3.<A>→b<B>
4.<B>→c<B>
5.<B>→ε

Вариант 3.

1.<S>→a<A>b<S>
2.<S>→b
3.<A>→c<S>
4.<A>→b<A>
5.<A>→a

Вариант 4.

1 .<S>→a<S><A>
2.<S>→ε
3.<A>→a<A>b
4.<A>→b<A>
5.<A>→ε

Вариант 5.

1 .<S>→a<B><S><A>

2.<B>→b

3.<A>→a<A>

4.<A>→b<B>

5.<S>→b

Вариант 6.

1 .<S>→a<S><S><A>

2.<S>→b

3.<A>→b<B>

4.<A>→a<A><A>

5.<B>→ε

Вариант 7.

1.<S>→a<S><A>b<B>

2.<S>→b

3.<A>→a<A>

4.<A>→e

5.<B>→b<B>

6.<B>→e

Вариант 8.

1 .<S>→a<B><S><A

2. <A>→a<B>

3.<B>→b<B>

4.<B>→c<A><B>

5.<B>→ε

2. LL(1)-грамматика порождает дерево вывода, изображенное на рис.42. Выписать последовательность конфигураций МП-распознавателя для этой грамматики, возникающие при обработке следующих входных цепочек:

a) abcccdaa;

b) ε;

c) accc;

d) adab.

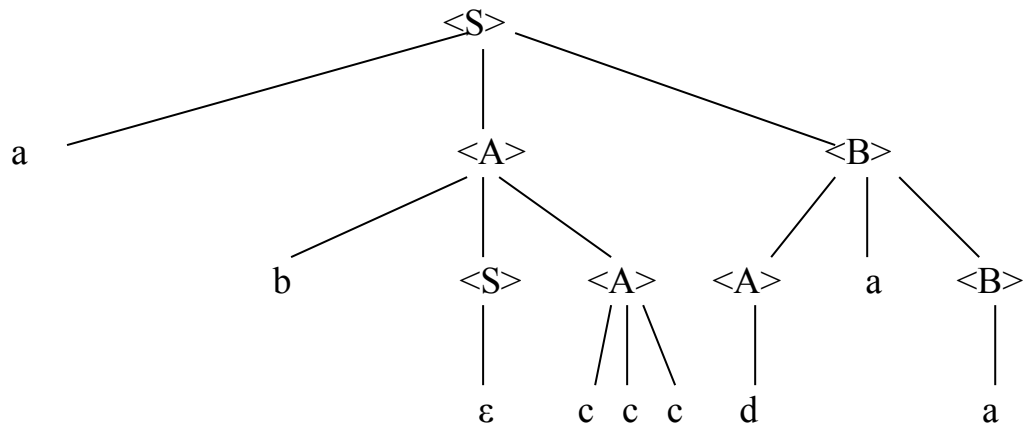


Рис.42.

3. Для следующей грамматики с начальным символом <S>

<S>→ a<A><B>b<C><D> <S>→ε

<A>→ <A><S>d <A>→ε

<B>→ <S><A>c <B>→e<C>

<C>→<C>g <C>→ε

<D>→a<B><D> <D>→ε

a) найти множество СЛЕД для каждого нетерминала;

b) найти ВЫБОР для каждого правила;

c) определить, является ли данная грамматика LL(1)-грамматикой.

4. Построить МП-распознаватель для грамматики арифметических выражений, в которых используются операции + и * .

1. <E>→<T><E->

2. <E->→ +<T><E->

3. <E->→ε

4. <T>→ <P><T->

5. <T->→ *<P><T->

6. <T->→ε

7. <P>→(<E>)

8. <P>→ I

5. Построить g-грамматику для конструкций языка СИ++. Считать терминалами <логическое выражение>, <арифметическое выражение> и <оператор>, т.е. когда доходит до них, вывод прекращается.

a) Описание массивов.

b) Описание условного оператора.

c) Операторы цикла.

V. КОНСПЕКТ ЛЕКЦИЙ

Пользовательский интерфейс операционной системы

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

При работе за алфавитно-цифровым терминалом пользователь имеет систему команд, мощность которой отражает функциональные возможности ОС, позволяет запускать и останавливать приложения, выполнять операции с файлами, получать информацию о состоянии ОС, администрировать систему. Команды могут вводится в интерактивном режиме или считываться из командного файла.

Ввод команды упрощен, если ОС поддерживает графический пользовательский интерфейс. GUI – Graphical User Interface – неотъемлемая часть многих современных ОС.

Ресурсами пользовательского интерфейса являются множества данных, обеспечивающих внешний вид интерфейса пользователя результирующей программы и не связанных напрямую с логикой его выполнения, например, тексты сообщений, выдаваемые программой, цветовая гамма элементов интерфейса, надписи на элементах управления.

Управление задачами, управление памятью, управление вводом-выводом, управление файлами

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

Управление процессами

Для каждого вновь созданного процесса ОС генерирует системные информационные структуры, содержащие данные о потребностях процесса в ресурсах и действительно выделенных ему ресурсах.

Процессы, порождаемые пользователями и их приложениями, называются *пользовательскими*, а инициированные самой ОС для выполнения своих функций – *системными*.

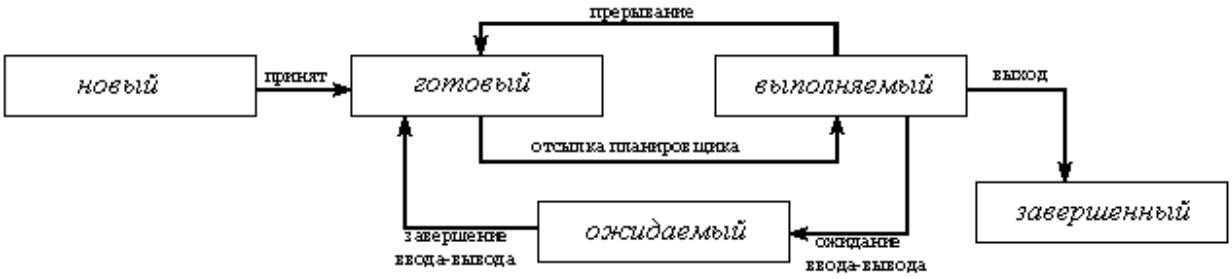
На протяжении своего существования процесс может быть многократно прерван и продолжен. Для его возобновления восстанавливается состояние его операционной среды, определяющееся состоянием регистров, программного счетчика, режимов работы процессора, указателями на открытые файлы, информацией о незавершенных операциях ввода-вывода, кодами ошибок выполняемых данным процессом системных вызовов и др.

Создать процесс означает создать *описатель процесса,* в качестве которого выступает одна или несколько информационных структур, содержащих все сведения о процессе, необходимые операционной системе для управления им.

При управлении процессами ОС использует два основных типа информационных структур дескриптор процесса и контекст процесса. *Дескриптор процесса* содержит информацию, необходимую ядру в течение всего жизненного цикла процесса, независимо от того находится он в пассивном или активном состоянии, находится его образ (совокупность его кодов и данных) в оперативной памяти ли выгружен на диск.

В течение жизни поток переходит из одного состояния в другое в соответствии с алгоритмом планирования.

На рисунке приведена типовая диаграмма переходов для состояний процессора.



В состоянии выполнения в одно процессорном режиме может находиться не более одного потока, а в каждом из состояний ожидания и готовности – несколько. Эти потоки образуют очереди соответственно ожидающих и готовых потоков. Очереди потоков организуются путем объединения в списки описателей отдельных потоков. Каждый описатель потока содержит, по крайней мере, один указатель на другой описатель, что позволяет легко переупорядочивать потоки в очереди, включать и исключать их, переводить в другое состояние.

С самых общих позиций все множество алгоритмов планирования можно разделить на два класса:

- *невытесняющие* алгоритмы – в которых активному потоку позволяется выполняться, пока он по собственной инициативе не отдаст управление операционной системе для того, чтобы она выбрала другой готовый поток;
- *вытесняющие* алгоритмы – в которых решение о переключении процессора с выполнения одного потока на другой принимается ОС.

В основе многих вытесняющих алгоритмов лежит концепция *квантования*, в которой каждому потоку поочередно для выполнения предоставляется ограниченный непрерывный период процессорного времени *квант*.

Другой важной концепцией, лежащей в основе многих вытесняющих алгоритмов, является *приоритетное обслуживание. Приоритет* – это число, характеризующее степень привилегированности потока при использовании ресурсов вычислительной машины, в частности процессорного времени: чем выше приоритет, тем выше привилегии, тем меньше времени будет проводить поток в очередях.

В ОС также широко применяются алгоритмы планирования смешанной структуры. Например, в основе планирования лежит квантование, но величина кванта и/или порядок выбора потока из очереди готовых определяется приоритетами потоков. Примером служит ОС Windows NT, в которой квантование сочетается с динамическими абсолютными приоритетами. На выполнение выбирается готовый поток с наивысшим приоритетом. Ему выделяется квант времени. Если во время выполнения в очереди готовых процессов появляется поток с более высоким приоритетом, то он вытесняет выполняемый поток. Вытесненный поток возвращается в очередь готовых, причем он становится впереди всех потоков, имеющих такой же приоритет.

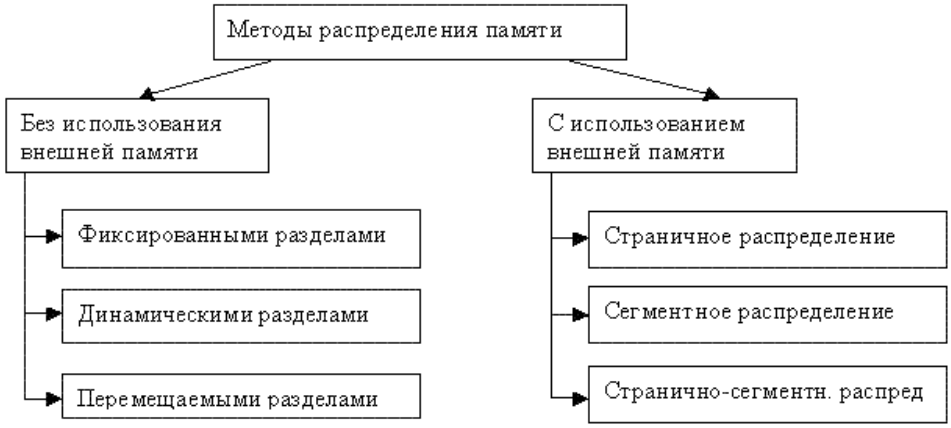
Управление памятью

Эта подсистема включает распределение физической памяти между существующими в данный момент процессами, загрузку кодов и данных процессов в отведенные им области памяти, настройку частей кодов программы на физические адреса памяти, защиту областей памяти каждого процесса.

К базовым вопросам управления памятью относятся:

- решение о назначение каждому процессу одной непрерывной области памяти или «кусками»;
- должны ли сегменты программы, загружаемые в память, находиться на одном месте в течение всего периода выполнения процесса, или ее можно время от времени сдвигать;
- что делать, если программы не помещаются в имеющуюся память.

Разные ОС по-разному решают эти вопросы.



Некоторые из них сохранили актуальность и широко используются в современных ОС, другие представляют собой познавательный интерес, но их можно встретить до сих пор в специализированных системах.

Большое количество задач, необходимое для высокой загрузки процессора, требует большого объема ОП. Для обеспечения приемлемого уровня мультипрограммирования используется метод, в котором образы некоторых процессов целиком или полностью выгружаются на диск. Как правило, это образы процессов, находящиеся в ожидании завершения операций ввода-вывода или освобождения ресурса, а также процессы в состоянии готовности, стоящие в очереди к процессору.

Такая подмена (*виртуализация*) ОП дисковой памятью позволяет повысить уровень мультипрограммирования. Она осуществляется совокупностью программных модулей ОС и аппаратных схем процессора.

Виртуализация памяти может быть осуществлена на основе двух различных подходов:

- *своппиг* - образы виртуальных процессов выгружаются на диск и в ОП целиком;
- *виртуальная память* – между ОП и диском перемещаются части (сегменты, страницы и т.п.) образов процессов.

Управление файлами и внешними устройствами

Способность ОС к экранированию сложностей реальной аппаратуры проявляется в файловой системе. При выполнении своих функций файловая система взаимодействует с подсистемой управления внешними устройствами, которая осуществляет передачу данных между дисками и оперативной памятью.

Подсистема управления внешними устройствами, называемая также подсистемой ввода-вывода исполняет роль интерфейса ко всем устройствам, подключенным к компьютеру.

Программа, управляющая конкретной моделью внешнего устройства, называется драйвером этого устройства. Драйвер может управлять единственной моделью устройства или группой устройств определенного типа.

Созданием драйверов устройств занимаются как разработчики конкретной ОС, так и специалисты компании, выпускающей это устройство. ОС должна поддерживать хорошо определенный интерфейс между драйверами и остальной частью ОС.

Защита данных и администрирование

Безопасность данных обеспечивается средствами отказоустойчивости ОС. Первым этапом защиты от несанкционированного доступа является процедура логического входа. Права пользователей при обращении к ресурсам и выполнение ими тех или иных действий определяет администратор.

Функции аудита заключаются в фиксации всех событий, от которых зависит безопасность системы. Список событий также назначается администратором.

Поддержка отказоустойчивости реализуется резервированием, т.е. хранением нескольких копий данных на разных дисках или накопителях, и использованием нескольких процессоров.

Поддержка отказоустойчивости также входит в обязанности системного администратора.

Программирование в операционной среде, ассемблеры

Язык ассемблера – язык низкого уровня. Структура и взаимосвязь цепочек этого языка близки к машинным командам целевой вычислительной системы, где должна выполняться результирующая программа. Применение языка ассемблера позволяет разработчику управлять ресурсами (процессором, оперативной памятью, внешними устройствами и т. п.). Каждая команда исходной программы на языке ассемблера в результате компиляции преобразуется в одну машинную команду. Компилятор с языка ассемблера зачастую просто называют «ассемблер» или «программа ассемблера».

Язык ассемблера, как правило, содержит мнемонические коды машинных команд. Чаще всего используется англоязычная мнемоника команд, но существуют и другие варианты языков ассемблера (в том числе существуют и русскоязычные варианты).

Все возможные команды в каждом языке ассемблера можно разбить на две группы:

обычные команды языка, которые в процессе трансляции преобразуются в машинные команды;

специальные команды языка, которые в машинные команды не преобразуются, но используются компилятором для выполнения задач компиляции (например, как задача распределения памяти).

Синтаксис языка ассемблера чрезвычайно прост. Команды исходной программы записываются обычно таким образом, чтобы на одной строке программы располагалась одна команда. Каждая команда языка ассемблера, как правило, может быть разделена на три составляющих, расположенных последовательно одна за другой: поле метки, код операции и поле операндов. Компилятор с языка ассемблера обычно предусматривает возможность наличия во входной программе комментариев, которые отделяются от команд заданным разделителем.

Поле метки содержит идентификатор, представляющий собой метку, либо является пустым. Каждый идентификатор метки может встречаться в программе на языке ассемблера только один раз. Метка считается описанной там, где она непосредственно встретилась в программе (предварительное описание меток не требуется). Метка может быть использована для передачи управления на помеченную ею команду. Нередко метка отделяется от осталь-

ной части команды специальным разделителем (чаще всего — двоеточием «:»).

Код операции всегда представляет собой строго определенную мнемонику одной из возможных команд процессора или также строго определенную команду самого компилятора. Код операции записывается алфавитными символами входного языка. Чаще всего его длина составляет 3-4, реже - 5 или 6 символов. Поле операндов либо является пустым, либо представляет собой список из одного, двух, реже – трех операндов. Их количество строго определено и зависит от кода операции — каждая операция языка ассемблера предусматривает строго заданное число своих операндов. Соответственно, каждому из этих вариантов соответствуют безадресные, одноадресные, двухадресные или трехадресные команды (большее число операндов практически не используется, поскольку в современных компьютерах даже трехадресные команды встречаются редко). В качестве операндов могут выступать идентификаторы или константы.

Компилятор с языка ассемблера по своей структуре и назначению ничем не отличается от всех прочих компиляторов. Однако язык ассемблера имеет ряд особенностей, которые упрощают реализацию его компилятора.

Во-первых, особенностью языка ассемблера является то, что ряд идентификаторов в нем выделяется специально для обозначения регистров процессора.

Во-вторых, в компиляторах с языка ассемблера не выполняется дополнительная идентификация переменных — все переменные языка сохраняют имена, присвоенные им пользователем. За уникальность имен в исходной программе отвечает ее разработчик, семантика языка никаких дополнительных требований на этот процесс не накладывает.

В-третьих, в компиляторах с языка ассемблера предельно упрощено распределение памяти. Компилятор с языка ассемблера работает только со статической памятью. Если используется динамическая память, то для работы с нею нужно использовать соответствующую библиотеку или функции ОС, а за распределение памяти отвечает разработчик исходной программы. За передачу параметров и организацию дисплея памяти процедур и функций также отвечает разработчик исходной программы.

В-четвертых, на этапе генерации кода в компиляторе с языка ассемблера не производится оптимизация, поскольку разработчик исходной программы сам отвечает за организацию вычислений, последовательность машинных команд и распределение регистров процессора.

Макроязыки

Разработка программ на языке ассемблера — достаточно трудоемкий процесс, требующий зачастую простого повторения одних и тех же многократно встречающихся операций. Примером может служить последовательность команд, выполняемых каждый раз для организации стекового дисплея

памяти при входе в процедуру или функцию. Для облегчения труда разработчика были созданы так называемые макрокоманды.

Макрокоманда представляет собой текстовую подстановку, в ходе выполнения которой каждый идентификатор определенного вида заменяется на цепочку символов из некоторого хранилища данных. Процесс выполнения макрокоманды называется *макрогенерацией,* а цепочка символов, получаемая в результате выполнения макрокоманды, — *макрорасширением.*

Процесс выполнения макрокоманд заключается в последовательном просмотре текста исходной программы, обнаружении в нем определенных идентификаторов и замене их на соответствующие строки символов. Причем выполняется именно текстовая замена одной цепочки символов (идентификатора) на другую цепочку символов (строку). Такая замена называется *макроподстановкой*.

Для указания какие идентификаторы и на какие строки необходимо заменять, служат *макроопределения.* Макроопределения присутствуют непосредственно в тексте исходной программы. Они выделяются специальными ключевыми словами либо разделителями, которые не могут встречаться нигде больше в тексте программы. В процессе обработки все макроопределения полностью исключаются из текста входной программы, а содержащаяся в них информация запоминается в хранилище данных для обработки в процессе выполнения макрокоманд.

Макроопределение может содержать параметры. Тогда каждая соответствующая ему макрокоманда должна при вызове содержать строку символов вместо каждого параметра. Эта строка подставляется при выполнении макрокоманды везде, где в макроопределении встречается соответствующий параметр. В качестве параметра макрокоманды может оказаться другая макрокоманда, тогда она будет рекурсивно вызвана всякий раз, когда необходимо выполнить подстановку параметра.

Макрокоманды и макроопределения обрабатываются специальным модулем, называемым *макропроцессором,* или *макрогенератором.* Макрогенератор получает на вход текст исходной программы, содержащий макроопределения и макрокоманды, а на выходе его появляется текст макрорасширения исходной программы, не содержащий макроопределений и макрокоманд. Оба текста являются только текстами исходной программы на входном языке, никакая другая обработка не выполняется.

Макроопределения и макрокоманды нашли применение не только в языках ассемблера, но и во многих языках высокого уровня. Там их обрабатывает специальный модуль, называемый препроцессором языка (например, широко известен препроцессор языка С). Принцип обработки остается тем же самым, что и для программ на языке ассемблера — препроцессор выполняет текстовые подстановки непосредственно над строками самой исходной программы.

В языках высокого уровня макроопределения должны быть отделены от текста самой исходной программы, чтобы препроцессор не мог спутать их с синтаксическими конструкциями входного языка. Для этого используются либо специальные символы и команды (команды препроцессора), которые ни-

когда не могут встречаться в тексте исходной программы, либо макроопределения встречаются внутри незначащей части исходной программы - входят в состав комментариев.

Макроязык является чисто формальным языком, поскольку он лишен какого-либо смысла. Семантика макроязыка полностью определяется макрорасширением текста исходной программы, которое получается после обработки всех предложений макроязыка, а сами по себе предложения макроязыка не несут никакой семантики.

Препроцессор макроязыка (макрогенератор) в общем случае представляет собой транслятор, на вход которого поступает исходная программа, содержащая макроопределения и макрокоманды, а результирующей программой является макрорасширение программы на исходном языке.

Формальные языки и грамматики

Цепочки символов. Операции над цепочками символов

Цепочкой символов (или строкой) называют произвольную упорядоченную конечную последовательность символов, записанных один за другим.

Цепочка символов — последовательность, в которую могут входить любые допустимые символы, причем это необязательно некоторая осмысленная последовательность.

Цепочки символов α и β равны (совпадают), $\alpha = \beta$, если они имеют один и тот же состав символов, одно и то же их количество и одинаковый порядок следования символов в цепочке.

Количество символов в цепочке называют *длиной цепочки.* Длина цепочки символа α обозначается $|\alpha|$. Очевидно, что если $\alpha = \beta$, то $|\alpha| = |\beta|$.

Основной операцией над цепочками символов является операция *конкатенации* (объединения или сложения) цепочек.

Конкатенация цепочек символов не обладает свойством коммутативности, то есть в общем случае существуют такие α и β, что $\alpha\beta \neq \beta\alpha$.

Также очевидно, что конкатенация обладает свойством ассоциативности, то есть $(\alpha\beta)\gamma = \alpha(\beta\gamma)$.

Любую цепочку символов языка можно представить как конкатенацию составляющих ее частей. Причем, разбиение на подцепочки можно выполнить несколькими способами.

Если некоторую цепочку символов разбить на составляющие ее подцепочки, а затем заменить одну из подцепочек на любую произвольную цепочку символов, то в результате получится новая цепочка символов. Такое действие называется *заменой,* или *подстановкой,* цепочки.

Можно выделить еще две операции над цепочками:

Обращение цепочки — это запись символов цепочки в обратном порядке. Обращение цепочки α обозначается как α^R. Если α - *«абвг»,* то α^R –

«*гвба*». Для операции обращения справедливо следующее равенство $\forall \alpha, \beta : (\alpha\beta)^R = \alpha^R \beta^R$.

Итерация (повторение) цепочки n раз, где $n \in N$, $n > 0$ — это конкатенация цепочки самой с собой n раз. Итерация цепочки α n раз обозначается как α^n. Для операции повторения справедливы следующие равенства $\forall \alpha : \alpha^1 = \alpha$, $\alpha^2 = \alpha\alpha$, $\alpha^3 = \alpha\alpha\alpha$,... и т. д. Итерация цепочки символов определена и для n = 0 — в этом случае результатом итерации будет пустая цепочка символов.

Пустая цепочка символов — это цепочка, не содержащая ни одного символа. Пустую цепочку принято обозначать греческой буквой λ (в литературе ее иногда обозначают латинской буквой e или греческой ε).

Для пустой цепочки справедливы следующие равенства:

1. $|\lambda| = 0$
2. $\forall \alpha : \lambda\alpha = \alpha\lambda = \alpha$
3. $\lambda^R = \lambda$
4. $\forall n \geq 0 : \lambda^n = \lambda$
5. $\forall \alpha : \alpha^0 = \lambda$

Понятие языка. Формальное определение языка

В общем случае язык — это заданный набор символов и правил, устанавливающих способы комбинации этих символов между собой для записи осмысленных текстов. Основой любого естественного или искусственного языка является алфавит, определяющий набор допустимых символов языка.

Алфавит — счетное множество допустимых символов языка. Алфавит не обязательно должен быть конечным множеством, но реально существующие языки строятся на основе конечных алфавитов.

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

Если V — некоторый алфавит, то:

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

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

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

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

Цепочку символов, принадлежащую заданному языку, часто называют *предложением* языка, а множество цепочек символов некоторого языка L(V) — множеством предложений этого языка.

Для любого языка L(V) справедливо: $L(V) \subseteq V^*$.

Язык L(V) включает в себя язык L'(V): $L'(V) \subseteq L(V)$, если $\forall \alpha \in L(V) : \alpha \in L'(V)$.

Множество цепочек языка L'(V) является подмножеством множества цепочек языка L(V) (или эти множества совпадают). Очевидно, что оба языка должны строиться на основе одного и того же алфавита.

Два языка L(V) и L'(V) совпадают (эквивалентны): L'(V) = L(V), если L'(V) ⊆ L(V) и L(V)⊆ L'(V); или, что то же самое: $\forall\, \alpha \in$ L'(V): $\alpha \in$ L(V) и $\forall\, \beta \in$ L(V): $\beta \in$ L'(V).

Множества допустимых цепочек символов для эквивалентных языков равны.

Два языка L(V) и L'(V) почти эквивалентны: L'(V)≅ L(V), если L'(V) $\cup \{\lambda\} = = L(V) \cup \{\lambda\}$. Множества допустимых цепочек символов почти эквивалентных языков могут различаться только на пустую цепочку символов.

Способы задания языков. Синтаксис и семантика языка

В общем случае язык можно определить тремя способами:

1. перечислением всех допустимых цепочек языка;
2. указанием способа порождения цепочек языка (заданием грамматики языка);
3. определением метода распознавания цепочек языка.

Первый из методов является чисто формальным и на практике не применяется.

Второй способ предусматривает некоторое описание правил, с помощью которых строятся цепочки языка. Тогда любая цепочка, построенная с помощью этих правил из символов алфавита языка, будет принадлежать заданному языку. Например, изучение правил построения цепочек символов в русском языке.

Третий способ предусматривает построение некоторого логического устройства (распознавателя) — автомата, который на входе получает цепочку символов, а на выходе выдает ответ: принадлежит или нет эта цепочка заданному языку.

В любом языке выделяют его синтаксис и семантику. Кроме того, трансляторы имеют дело также с лексическими конструкциями (лексемами), которые задаются лексикой языка.

Синтаксис языка — это набор правил, определяющий допустимые конструкции зыка. Синтаксис определяет «форму языка» — задает набор цепочек символов, которые принадлежат языку. Чаще всего синтаксис языка можно задать в виде строгого набора правил, но полностью это утверждение справедливо только для чисто формальных языков.

Семантика языка — это раздел языка, определяющий значение предложений языка. Семантика определяет «содержание языка» — задает смысл для всех допустимых цепочек языка. Семантика для большинства языков определяется неформальными методами (отношения между знаками и тем, что они

обозначают, ручаются семиотикой). Чисто формальные языки лишены какого-либо смысла.

Лексика — это совокупность слов (словарный запас) языка. Слово или лексическая единица (лексема) языка — это конструкция, которая состоит из элементов алфавита языка и не содержит в себе других конструкций. Иначе говоря, лексическая единица может содержать только элементарные символы и не может содержать других лексических единиц.

Лексическими единицами (лексемами) русского языка являются слова русского языка, а знаки препинания и пробелы представляют собой разделители, не образующие лексем. Лексическими единицами алгебры являются числа, знаки математических операций, обозначения функций и неизвестных величин. В языках программирования лексическими единицами являются ключевые слова, идентификаторы, константы, метки, знаки операций; в них также существуют и разделители (запятые, скобки, точки с запятой и т. д.).

Грамматики и распознаватели
Формальное определение грамматики. Форма Бэкуса—Наура

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

Фактически, определив грамматику языка, указываются правила порождения цепочек символов, принадлежащих этому языку. Таким образом, грамматика — генератор цепочек языка. Она относится ко второму способу определения языков – порождению цепочек символов.

Грамматику языка можно описать различными способами. Например, грамматика русского языка описывается довольно сложным набором правил, которые изучаются в школе. Для некоторых языков (в том числе для синтаксических конструкций языков программирования) можно использовать формальное описание грамматики, построенное на основе системы правил (или продукций).

Правило (или продукция) — упорядоченная пара цепочек символов (α, β). В правилах важен порядок цепочек, поэтому их чаще записывают в виде α -> β (или $\alpha ::= \beta$). Такая запись читается как «α порождает β » или « β по определению есть α ».

Грамматика языка программирования содержит правила двух типов: первые (определяющие синтаксические конструкции языка) довольно легко поддаются формальному описанию; вторые (определяющие семантические ограничения языка) обычно излагаются в неформальной форме. Поэтому любое описание (или стандарт) языка программирования обычно состоит из двух частей: 1) формальное изложение правил построения синтаксических конструкций, 2)описание семантических правил на естественном языке.

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

Две грамматики G и G' называются эквивалентными, если они определяют один и тот же язык: L(G) = L(G'). Две грамматики G и G' называются почти

эквивалентными, если заданные ими языки различаются не более чем на пустую цепочку символов: $L(G) \cup \{\lambda\} = L(G') \cup \{\lambda\}$.

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 $\alpha \to \beta$,
где $\alpha \in (VN \cup VT)^+$, $\beta \in (VN \cup VT)^*$;
S – целевой (начальный) символ грамматики $S \in VN$.

Алфавиты терминальных и нетерминальных символов грамматики не пересекаются: $VN \cap VT$ = пустое множество. Это значит, что каждый символ в грамматике может быть либо терминальным, либо нетерминальным, но не может быть терминальным и нетерминальным одновременно. Целевой символ грамматики — это всегда нетерминальный символ. Множество $V = VN \cup VT$ называют полным алфавитом грамматики G.

Множество терминальных символов VT содержит символы, которые входят в алфавит языка, порождаемого грамматикой. Как правило, символы из множества VT встречаются только в цепочках правых частей правил. Множество нетерминальных символов VN содержит символы, которые определяют слова, понятия, конструкции языка. Каждый символ этого множества может встречаться в цепочках как левой, так и правой частей правил грамматики, но он обязан хотя бы один раз быть в левой части хотя бы одного правила. Правила грамматики обычно строятся так, чтобы в левой части каждого правила был хотя бы один нетерминальный символ.

Во множестве правил грамматики может быть несколько правил, имеющих одинаковые левые части, вида: $\alpha \to \beta_1$, $\alpha \to \beta_2$, ... $\alpha \to \beta_n$. Тогда эти правила объединяют вместе и записывают в виде: $\alpha \to \beta_1 | \beta_2 | \ldots | \beta_n$. Одной строке в такой записи соответствует сразу n правил.

Такую форму записи правил грамматики называют формой Бэкуса-Наура. Форма Бэкуса-Наура предусматривает, как правило, также, что нетерминальные символы берутся в угловые скобки: < >. Иногда знак -> в правилах грамматики заменяют на знак : := (что характерно для старых монографий), но это всего лишь незначительные модификации формы записи, не влияющие на ее суть.

Пример грамматики, которая определяет язык целых десятичных чисел со знаком:
G ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -, +},{<число>, <чс>, <цифра>}, P, <число>)
P:
<<число> -> <чс> | +<чс> | -<чс>
<<чс> -> <цифра> | <чс><цифра>
<<цифра> -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Составляющие элементы грамматики G:
□ множество терминальных символов VT содержит двенадцать элементов: десять десятичных цифр и два знака;

□ множество нетерминальных символов VN содержит три элемента: символы <число>, <чс> и <цифра>;
□ множество правил содержит 15 правил, которые записаны в три строки (то есть имеется только три различных правых части правил);
□ целевым символом грамматики является символ <число>.

Символ <чс> - это бессмысленное сочетание букв русского языка, но это обычный нетерминальный символ грамматики, такой же, как и два других. Названия нетерминальных символов не обязаны быть осмысленными, это сделано просто для удобства понимания правил грамматики человеком.

Для терминальных символов это неверно. Набор терминальных символов всегда строго соответствует алфавиту языка, определяемого грамматикой. Например, грамматика для языка целых десятичных чисел со знаком, в которой нетерминальные символы обозначены большими латинскими буквами, выглядит так:

G' ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -, +}, {S, T, F}, P, S)
P:
S –> T | +T | -T
T –> F | TF
F –> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Здесь изменилось только множество нетерминальных символов. Теперь VN = {S, T, F}. Язык, заданный грамматикой, не изменился — можно сказать, что грамматики G G' эквивалентны.

Принцип рекурсии в правилах грамматики

Рекурсия в правилах грамматики выражается в том, что один из нетерминальных символов определяется сам через себя. Рекурсия может быть непосредственной (явной) — тогда символ 1ределяется сам через себя в одном правиле, либо косвенной (неявной) — тогда) же самое происходит через цепочку правил.

В рассмотренном примере грамматике G непосредственная рекурсия присутствует правиле: <чс> -> <чс><цифра>, а в эквивалентной ей грамматике G' — в правиле: T->TF.

Чтобы рекурсия не была бесконечной, для участвующего в ней нетерминального символа грамматики должны существовать также и другие правила, которые определяют его, минуя его самого, и позволяют избежать бесконечного рекурсивного определения (в противном случае этот символ в грамматике был бы просто не нужен). Такими правилами являются <чс> -> <цифра> - в грамматике G и T -> F — в грамматике G'.

Принцип рекурсии (иногда его называют «принцип итерации», что не меняет сути) — важное понятие в представлении о формальных грамматиках. Так или иначе, явно или неявно рекурсия всегда присутствует в грамматиках любых реальных языков программирования. Именно она позволяет строить бесконечное множество цепочек языка, и говорить об их порождении невоз-

можно без понимания принципа рекурсии. Как правило, в грамматике реального языка программирования содержится не одно, а целое множество правил, построенных с помощью рекурсии.

Другие способы задания грамматик

Форма Бэкуса-Наура — удобный, с формальной точки зрения, но не всегда доступный для понимания способ записи формальных грамматик. Рекурсивные определения хороши для формального анализа цепочек языка, но не удобны с точки зрения человека.

Например, то, что правила <чс> -> <цифра> | <чс><цифра> отражают возможность построения числа дописыванием справа любого числа цифр, начиная от одной, неочевидно и требует дополнительного пояснения. Но при создании языка программирования важно, чтобы его грамматику понимали не только те, кому предстоит создавать компиляторы для этого языка, но и пользователи языка – будущие разработчики программ. Поэтому существуют другие способы описания правил формальных грамматик, которые ориентированы на большую понятность для любого человека.

Запись правил грамматик с использованием метасимволов

Запись правил грамматик с использованием метасимволов предполагает, что в строке правила грамматики могут встречаться специальные символы — метасимволы, — которые имеют особый смысл и трактуются специальным образом. В качестве таких метасимволов чаще всего используются следующие символы: ( ) (круглые скобки), [ ] (квадратные скобки), { } (фигурные скобки), " " (кавычки) и , (запятая).

Они имеют следующий смысл:

- круглые скобки означают, что из всех перечисленных внутри них цепочек символов в данном месте правила грамматики может стоять только одна цепочка;
- квадратные скобки означают, что указанная в них цепочка может встречаться, а может и не встречаться в данном месте правила грамматики (то есть может быть в нем один раз или ни одного раза);
- фигурные скобки означают, что указанная внутри них цепочка может не встречаться в данном месте правила грамматики ни одного раза, встречаться один раз или сколь угодно много раз;
- запятая служит для того, чтобы разделять цепочки символов внутри круглых скобок;
- кавычки используются в тех случаях, когда один из метасимволов нужно включить в цепочку обычным образом — то есть, когда одна из скобок или запятая должны присутствовать в цепочке символов языка (если саму кавычку нужно включить в цепочку символов, то ее надо повторить дважды — этот принцип знаком разработчикам программ).

Перепишем правила рассмотренной выше грамматики G с использованием метасимволов:

<число> -> [(+,-)]<цифра>{<цифра>}
<цифра> -> 0 | 1 | 2 | 3 | 4 | 5 | б | 7 | 8 | 9

Вторая строка правил не нуждается в комментариях, а первое правило читается так: «число есть цепочка символов, которая может начинаться с символов + и -, должна содержать дальше одну цифру, за которой может следовать любое количество цифр». В отличие от формы Бэкуса—Наура, в форме записи с помощью метасимволов, как видно, удален из грамматики малопонятный терминальный символ <чс>, а во-вторых — полностью исключена рекурсия. Грамматика стала более понятной.

Форма записи правил с использованием метасимволов — это удобный и понятный способ представления правил грамматик. Она во многих случаях позволяет полностью избавиться от рекурсии, заменив ее символом итерации { } (фигурные скобки). Эта форма наиболее употребительна для одного из типов грамматик — регулярных грамматик.

Кроме указанных выше метасимволов в целях удобства записи в описаниях грамматик иногда используют и другие метасимволы, при этом предварительно дается разъяснение их смысла. Принцип записи от этого не меняется. Также иногда дополняют смысл уже существующих метасимволов. Например, для метасимвола { } (фигурные скобки) существует удобная форма записи, позволяющая ограничить число повторений цепочки символов, заключенной внутри них: { }n, где n∈ N и n > 0. Такая запись означает, что цепочка символов, стоящая в фигурных скобках, может быть повторена от 0 до n раз (не более n раз). Это очень удобный метод наложения ограничений на длину цепочки.

Для рассмотренной грамматики G таким способом можно, например, записать правила, если предположить, что она должна порождать целые десятичные числа, содержащие не более 15 цифр:

<число> -> [(+, -)]<цифра>{<цифра>}14

<цифра> -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Запись правил грамматик в графическом виде

В такой форме записи каждому нетерминальному символу грамматики соответствует диаграмма, построенная в виде направленного графа. Граф имеет следующие типы вершин:

- точка входа (на диаграмме никак не обозначена, из нее просто начинается входная дуга графа);
- нетерминальный символ (на диаграмме обозначается прямоугольником, в который вписано обозначение символа);
- цепочка терминальных символов (на диаграмме обозначается овалом, кругом или прямоугольником с закругленными краями, внутрь которого вписана цепочка);
- узловая точка (на диаграмме обозначается жирной точкой или закрашенным кружком);
- точка выхода (никак не обозначена, в нее просто входит выходная дуга графа).

Каждая диаграмма имеет только одну точку входа и одну точку выхода, но сколько угодно вершин других трех типов. Вершины соединяются между собой направленными дугами графа (линиями со стрелками). Из входной точки дуги могут только выходить, а во входную точку — только входить. В остальные вершины дуги могут входить и выходить (в правильно построенной грамматике каждая вершина должна иметь как минимум один вход и как минимум один выход).

Описание понятия «число» из грамматики G с помощью диаграмм на рис. 1.

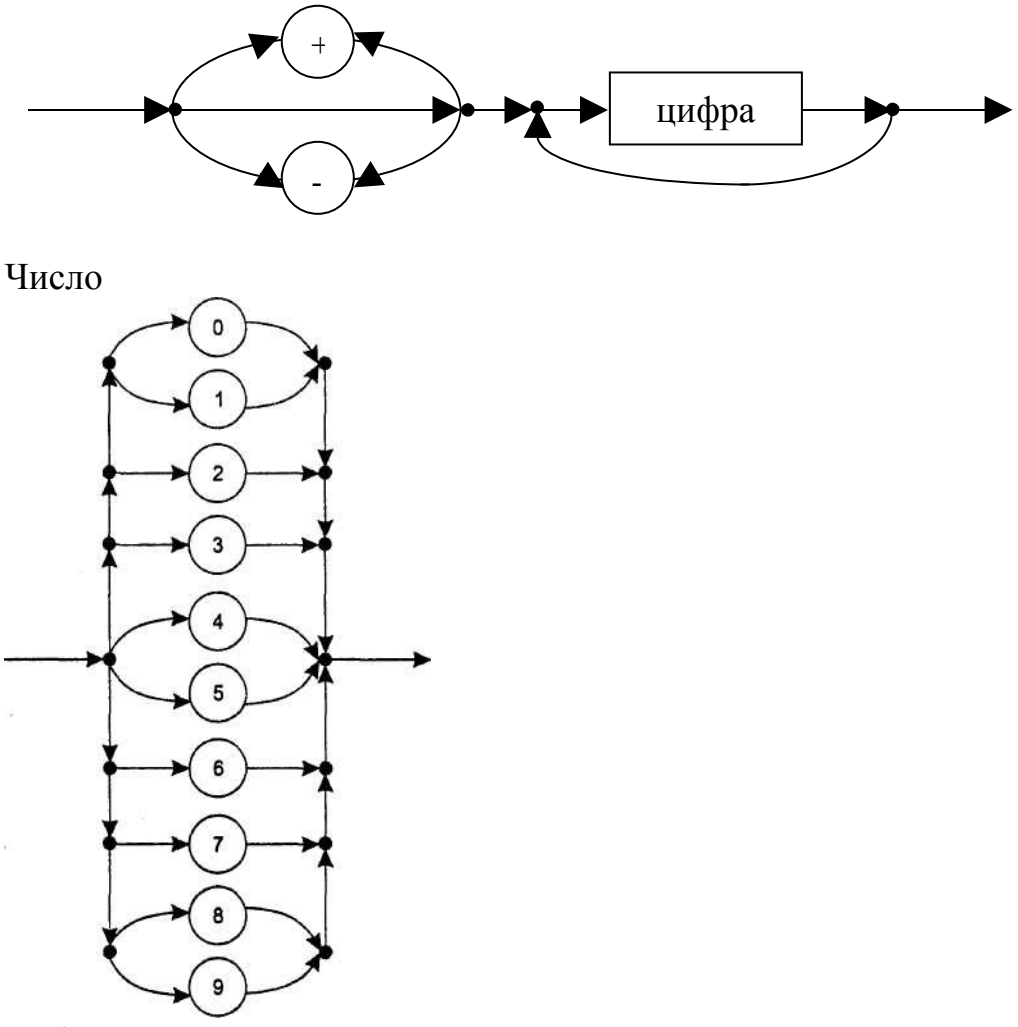


Рис 1. Графическое представление грамматики целых десятичных чисел со знаком

Распознаватели. Общая схема распознавателя

Распознаватель (или разборщик) — 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.

В общем виде распознаватель можно отобразить в виде условной схемы, представленной на рис.2.

Следует подчеркнуть, что представленный рисунок — всего лишь условная схема, отображающая работу алгоритма распознавателя. Ни в коем случае не стоит искать подобное устройство в составе компьютера. Распознаватель, являющийся частью компилятора, представляет собой часть программного обеспечения компьютера.

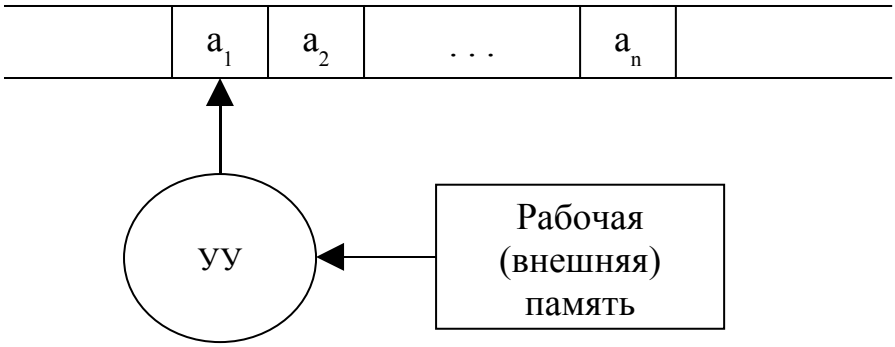


Рис 2. Условная схема распознавателя

Из рисунка видно, что распознаватель состоит из следующих основных компонентов:

- ленты, содержащей входную цепочку символов, и считывающей головки, обозревающей очередной символ в этой цепочке;
- устройства управления (УУ), которое координирует работу распознавателя, имеет некоторый набор состояний и конечную память (для хранения своего состояния и некоторой промежуточной информации);
- внешней (рабочей) памяти, которая может хранить некоторую информацию в процессе работы распознавателя и, в отличие от памяти УУ, имеет неограниченный объем.

Распознаватель работает с символами своего алфавита — алфавита распознавателя. Алфавит распознавателя конечен. Он включает в себя все допустимые символы входных цепочек, а также некоторый дополнительный алфавит символов, которые могут обрабатываться УУ и храниться в рабочей памяти распознавателя.

В процессе своей работы распознаватель может выполнять некоторые элементарные операции:

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

Какие конкретно операции должны выполняться в процессе работы распознавателя, определяется в УУ.

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

Конфигурация распознавателя определяется следующими параметрами:

-содержимым входной цепочки символов и положением считывающей головки в ней;

- состоянием УУ;

- содержимым внешней памяти.

Для распознавателя всегда задается определенная конфигурация, которая считается начальной. В начальной конфигурации считывающая головка обозревает первый символ входной цепочки, УУ находится в заданном начальном состоянии, а внешняя память либо пуста, либо содержит строго определенную информацию.

Кроме начального состояния для распознавателя задается одна или несколько конечных конфигураций. В конечной конфигурации считывающая головка, как правило, находится за концом исходной цепочки (часто для распознавателей вводят специальный символ, обозначающий конец входной цепочки).

Распознаватель *допускает входную цепочку символов* α , если, находясь в начальной конфигурации и получив на вход эту цепочку, он может проделать последовательность шагов, заканчивающуюся одной из его конечных конфигураций.

Формулировка «может проделать последовательность шагов» более точна, чем прямое указание «проделает последовательность шагов», так как для многих распознавателей при одной и той же входной цепочке символов из начальной конфигурации могут быть допустимы различные последовательности шагов, не все из которых ведут к конечной конфигурации.

Язык, определяемый распознавателем, — это множество всех цепочек, которые допускает распознаватель.

Виды распознавателей

Распознаватели можно классифицировать в зависимости от вида составляющих их компонентов: считывающего устройства, устройства управления (УУ) и внешней памяти.

По видам считывающего устройства распознаватели могут быть двусторонние и односторонние.

Односторонние распознаватели допускают перемещение считывающей головки по ленте входных символов только в одном направлении. Это значит, что на каждом шаге работы распознавателя считывающая головка может либо переместиться по ленте символов на некоторое число позиций в заданном направлении, либо остаться на месте. Поскольку все языки программирования подразумевают нотацию чтения исходной программы «слева направо», то так же работают и все распознаватели. Поэтому когда говорят об односторонних распознавателях, то прежде всего имеют в виду левосторонние, которые читают входную цепочку слева направо и не возвращаются назад к уже прочитанной части цепочки.

Двусторонние распознаватели допускают, что считывающая головка может перемещаться относительно ленты входных символов в обоих направлениях: как вперед, от начала ленты к концу, так и назад, возвращаясь к уже прочитанным символам.

По видам устройства управления распознаватели бывают детерминированные и недетерминированные.

Распознаватель называется *детерминированным* в том случае, если для каждой допустимой конфигурации распознавателя, которая возникла на некотором шаге его работы, существует единственно возможная конфигурация, в которую распознаватель перейдет на следующем шаге работы. В противном случае распознаватель называется недетерминированным.

Недетерминированный распознаватель может иметь допустимую конфигурацию, для которой существует некоторое конечное множество конфигураций, возможных на следующем шаге работы. Достаточно иметь хотя бы одну такую «фигурацию, чтобы распознаватель был недетерминированным.

По видам внешней памяти распознаватели бывают следующих типов:

- распознаватели без внешней памяти;
- распознаватели с ограниченной внешней памятью;
- распознаватели с неограниченной внешней памятью.

У распознавателей без внешней памяти внешняя память полностью отсутствует, в процессе их работы используется только конечная память УУ.

У распознавателей с ограниченной внешней памятью размер внешней памяти ограничен в зависимости от длины входной цепочки символов. Эти ограничения могут накладываться некоторой зависимостью объема памяти от длины цепочки — линейной, полиномиальной, экспоненциальной и т. д. Кроме того, для таких распознавателей может быть указан способ организации внешней памяти — стек, очередь, список и т. п.

Распознаватели с неограниченной внешней памятью предполагают, что для их работы может потребоваться внешняя память неограниченного объема (вне зависимости от длины входной цепочки). У таких распознавателей предполагается память с произвольным методом доступа.

Вместе эти три составляющих позволяют организовать общую классификацию распознавателей. Например, в этой классификации возможен такой тип: «двусторонний недетерминированный распознаватель с линейно ограниченной стековой памятью».

Тип распознавателя в классификации определяет сложность создания такого распознавателя, а следовательно, сложность разработки соответствующего программного обеспечения для компилятора. Чем выше в классификации стоит распознаватель, тем сложнее создавать алгоритм, обеспечивающий его работу. Разрабатывать двусторонние распознаватели сложнее, чем односторонние. Можно заметить, что недетерминированные распознаватели по сложности выше детерминированных. Зависимость затрат на создание алгоритма от типа внешней памяти также очевидна.

Классификация языков и грамматик

Тип, к которому относится тот или иной язык программирования, зависит сложности компилятора для этого языка. Чем сложнее язык, тем выше вычислительные затраты компилятора на анализ цепочек исходной программы, написанной на этом языке, а следовательно, сложнее сам компилятор и его структура. Для некоторых типов языков в принципе невозможно построить компилятор, который бы анализировал исходные тексты на этих языках за приемлемое время на основе ограниченных вычислительных ресурсов (именно поэтому до сих пор невозможно создавать программы на естественных языках, например на русском или английском).

Четыре типа грамматик по Хомскому

Согласно классификации, предложенной американским лингвистом Ноамом Хомским, профессором Массачусетского технологического института, формальные грамматики классифицируются по структуре их правил.

Тип 0: грамматики с фразовой структурой

На структуру их правил не накладывается никаких ограничений: для грамматики вида **G(VT,VN,P,S), V = VN∪ VT** правила имеют вид: $\alpha \rightarrow \beta$, где $\alpha \in V^+$, $\beta \in V^*$.

Это самый общий тип грамматик. В него подпадают все без исключения формальные грамматики, но часть из них, к общей радости, может быть также отнесена и к другим классификационным типам. Дело в том, что грамматики, которые относятся только к типу 0 и не могут быть отнесены к другим типам, являются самыми сложными по структуре. Практического применения грамматики, относящиеся только к типу 0, не имеют.

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

В этот тип входят два основных класса грамматик:

Контекстно-зависимые грамматики G**(VT,VN,P,**S**), V = VN∪ VT** имеют правила вида: $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$, где $\alpha_1, \alpha_2 \in V^*$, $A \in$ **VN**, $\beta \in$ **V**$^+$.

Неукорачивающие грамматики G**(VT,VN,P,**S**), V = VN∪ VT** имеют правила вида: $\alpha \rightarrow \beta$, где $\alpha, \beta \in V^+$, $|\beta| \geq |\alpha|$.

Структура правил КЗ-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. Именно поэтому эти грамматики называют «контекстно-зависимыми». Цепочки α_1 и α_2 в правилах грамматики обозначают контекст (α_1 — левый контекст, а α_2 — правый контекст), в общем случае любая из них (или даже обе) может быть пустой. Говоря иными словами, значение одного и того же символа может быть различным в зависимости от того, в каком контексте он встречается. Неукорачивающие грамматики имеют такую структуру правил, что при построении на их основе предложений языка, любая цепочка символов может быть заменена цепочкой символов не меньшей длины. Отсюда и название «неукорачивающие».

Доказано, что эти два класса грамматик эквивалентны. Это значит, что для любого языка, заданного контекстно-зависимой грамматикой, можно построить неукорачивающую грамматику, которая будет задавать эквивалентный язык, и наоборот: для любого языка, заданного неукорачивающей грамматикой, можно построить контекстно-зависимую грамматику, которая будет задавать эквивалентный язык.

При построении компиляторов такие грамматики не применяются, поскольку синтаксические конструкции языков программирования, рассматриваемые компиляторами, имеют более простую структуру и могут быть построены с помощью грамматик других типов.

Тип 2: контекстно-свободные (КС) грамматики

Контекстно-свободные (КС) грамматики G(**VT,VN,P**,S), **V = VN**∪ **VT** имеют правила вида: A -> β , где A∈ **VN**, β ∈ $\mathbf{V}^+$. Такие грамматики также иногда называют неукорачивающими контекстно-свободными (НКС) грамматиками (видно, что в правой части правила у них должен всегда стоять как минимум один символ).

Существует также почти эквивалентный им класс грамматик — укорачивающие контекстно-свободные (УКС) грамматики G(**VT,VN,P**,S), **V = VN**∪ **VT**, правила которых могут иметь вид: A -> β , где A∈ **VN**, β ∈ **V***.

Разница между этими двумя классами грамматик заключается лишь в том, что в УКС-грамматиках в правой части правил может присутствовать пустая цепочка (λ), а в НКС-грамматиках — нет. Отсюда ясно, что язык, заданный НКС-грамматикой, не может содержать пустой цепочки. Доказано, что эти два класса грамматик почти эквивалентны. В дальнейшем, когда речь будет идти о КС-грамматиках, уже не будет уточняться, какой класс грамматики (УКС или НКС) имеется в виду, если возможность наличия в языке пустой цепочки не имеет принципиального значения.

КС-грамматики широко используются при описании синтаксических конструкций языков программирования. Синтаксис большинства известных языков программирования основан именно на КС-грамматиках.

Внутри типа КС-грамматик кроме классов НКС и УКС выделяют еще целое множество различных классов грамматик, и все они относятся к типу 2. Далее, когда КС-грамматики будут рассматриваться более подробно, на некоторые из этих классов грамматик и их характерные особенности будет обращено особое внимание.

Тип 3: регулярные грамматики

К типу регулярных относятся два эквивалентных класса грамматик: леволинейные и праволинейные.

Леволинейные грамматики G(**VT,VN,P**,S), **V = VN**∪ **VT** могут иметь правила ух видов: A -> Bγ или A -> γ , где A, B∈ **VN**, γ ∈ **VT*.**

В свою очередь, праволинейные грамматики G(**VT,VN, P**,S), **V = VN**∪ **VT** могут меть правила тоже двух видов: A -> γ B или A ->γ , где A,B∈ **VN**, β ∈ **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.

Соотношения между типами грамматик

Типы грамматик соотносятся между собой особым образом. Из определения типов 2 и 3 видно, что любая регулярная грамматика является КС-грамматикой, но не наоборот. Также очевидно, что любая грамматика может быть отнесена к типу 0, поскольку он не накладывает никаких ограничений на правила. В то же время существуют укорачивающие КС-грамматики (тип 2), которые не являются ни контекстно-зависимыми, ни неукорачивающими (тип 1), поскольку могут содержать правила вида «A -> λ », недопустимые в типе 1.

Одна и та же грамматика в общем случае может быть отнесена к нескольким классификационным типам (например, как уже было сказано, все без исключения грамматики могут быть отнесены к типу 0). Для классификации грамматики всегда выбирают максимально возможный тип, к которому она может быть отнесена. Сложность грамматики обратно пропорциональна номеру типа, к которому относится грамматика. Грамматики, которые относятся только к типу 0, являются самыми сложными, а грамматики, которые можно отнести к типу 3, — самыми простыми.

Классификация языков

Языки классифицируются в соответствии с типами грамматик, с помощью которых они заданы. Поскольку один и тот же язык в общем случае может быть задан сколь угодно большим количеством грамматик, которые могут относиться к различным классификационным типам, то для классификации самого языка среди всех его грамматик выбирается грамматика с максимально возможным классификационным типом. Например, если язык L может быть задан грамматиками $\mathbf{G_1}$ и $\mathbf{G_2}$, относящимися к типу 1 (контекстно-зависимые), грамматикой $\mathbf{G_3}$., относящейся к типу 2 (контекстно-свободные) и грамматикой $\mathbf{G_4}$, относящейся типу 3 (регулярные), то сам язык должен быть отнесен к типу 3 и является регулярным языком.

От классификационного типа языка зависит не только то, с помощью какой грамматики можно построить предложения этого языка, но и то, насколько сложно распознать эти предложения. Распознать предложения – это значит построить распознаватель для языка (третий способ задания языка).

Сложность языка убывает с возрастанием номера классификационного типа языка. Самыми сложными являются языки типа 0, самыми простыми — языки типа 3. Согласно классификации грамматик, также существует четыре типа языков.

Тип О: языки с фразовой структурой

Это самые сложные языки, которые могут быть заданы только грамматикой, относящейся к типу 0. Для распознавания цепочек таких языков требуются вычислители, равномощные машине Тьюринга. Поэтому, если язык относится к типу 0, то для него практически невозможно построить компилятор, гарантированно выполняющий разбор предложений языка за ограниченное время на основе ограниченных вычислительных ресурсов.

К сожалению, практически все естественные языки общения между людьми, строго говоря, относятся именно к этому типу языков.

Именно поэтому столь велики сложности в автоматизации перевода текстов, написанных на естественных языках, а также отсутствуют компиляторы, которые бы воспринимали программы на основе таких языков.

Тип 1: контекстно-зависимые (КЗ) языки

Второй по сложности тип языков. В общем случае время на распознавание предложений языка, относящегося к типу 1, экспоненциально зависит от длины исходной цепочки символов.

Языки и грамматики, относящиеся к типу 1, применяются в анализе и переводе текстов на естественных языках. Распознаватели, построенные на их основе, позволяют анализировать тексты с учетом контекстной зависимости в предложениях входного языка (но они не учитывают содержание текста, поэтому в общем случае для точного перевода с естественного языка требуется вмешательство человека). На основе таких грамматик может выполняться автоматизированный перевод с одного естественного языка на другой, ими могут пользоваться сервисные функции проверки орфографии и правописания в языковых процессорах. В компиляторах КЗ-языки не используются.

Тип 2: контекстно-свободные (КС) языки

КС-языки лежат в основе синтаксических конструкций большинства современных языков программирования, на их основе функционируют некоторые довольно сложные командные процессоры, допускающие управляющие команды цикла и условия.

В общем случае время на распознавание предложений языка, относящегося к типу 1, полиноминально зависит от длины входной цепочки символов (в зависимости от класса языка это кубическая или квадратичная зависимость). Однако среди КС-языков существует много классов языков, для которых эта зависимость линейна. Практически все языки программирования можно отнести к одному из таких классов.

Тип 3: регулярные языки

Регулярные языки — самый простой тип языков. Они являются самым широко используемым типом языков в области вычислительных систем. Время на распознавание предложений регулярного языка линейно зависит от длины входной цепочки символов.

Регулярные языки лежат в основе простейших конструкций языков программирования (идентификаторов, констант и т. п.), кроме того, на их основе строятся многие мнемокоды машинных команд (языки ассемблеров), а также командные процессоры, символьные управляющие команды.

Для работы с ними можно использовать регулярные множества и выражения, конечные автоматы.

Классификация распознавателей

Для каждого из типов языков существует свой тип распознавателя с определенным составом компонентов и, следовательно, с заданной сложностью алгоритма работы.

Для *языков с фразовой структурой* (тип 0) необходим распознаватель, равномощный машине Тьюринга — недетерминированный двусторонний автомат, имеющий неограниченную внешнюю память. Поэтому для языков данного типа нельзя гарантировать, что за ограниченное время на ограниченных вычислительных ресурсах распознаватель завершит работу и примет решение о том, принадлежит или не принадлежит входная цепочка заданному языку. Следовательно, практического применения языки с фразовой структурой не имеют.

Для *контекстно-зависимых языков* (тип 1) распознавателями являются двусторонние недетерминированные автоматы с линейно ограниченной внешней памятью. Алгоритм работы такого автомата в общем случае имеет экспоненциальную сложность и время, необходимое на разбор входной цепочки по заданному алгоритму, экспоненциально зависит от длины входной цепочки символов.

Такой алгоритм распознавателя уже может быть реализован в программном обеспечении компьютера. Однако экспоненциальная зависимость времени разбора от длины цепочки существенно ограничивает применение распознавателей для контекстно-зависимых языков. Как правило, такие распознаватели применяются для автоматизированного перевода и анализа текстов на естественных языках, когда временные ограничения на разбор текста несущественны (следует также напомнить, что после такой обработки часто требуется вмешательство человека).

Для *контекстно-свободных языков* (тип 2) распознавателями являются односторонние недетерминированные автоматы с магазинной (стековой) внешней памятью — МП-автоматы. При простейшей реализации алгоритма работы такого автомата он имеет экспоненциальную сложность, однако путем некоторых усовершенствований алгоритма можно добиться полиномиальной (кубической) зависимости времени, необходимого на разбор входной цепочки, от длины этой цепочки. Среди всех КС-языков можно выделить класс детерминированных КС-языков, распознавателями для которых являются детерминированные автоматы с магазинной (стековой) внешней памятью — ДМП-автоматы. Для таких языков существует алгоритм работы распознавателя с квадратичной сложностью. Среди всех детерминированных КС-языков существуют такие классы языков, для которых возможно построить линейный распознаватель — распознаватель, у которого время принятия решения о принадлежности цепочки языку имеет линейную зависимость от длины цепочки. Именно эти языки представляют интерес при построении компиляторов. Синтаксические конструкции практически всех существующих языков программирования могут быть отнесены к одному из таких классов языков.

Тем не менее, следует помнить, что только синтаксические конструкции языков программирования допускают разбор с помощью распознавателей КС-языков. Сами языки программирования не могут быть полностью отнесены к типу КС-языков, поскольку предполагают контекстную зависимость в тексте исходной программы (например, такую, как необходимость предварительного описания переменных). Поэтому все компиляторы предполагают дополнительный семантический анализ текста исходной программы.

Для *регулярных языков* (тип 3) распознавателями являются односторонние недетерминированные автоматы без внешней памяти — конечные автоматы (КА). Предполагается линейная зависимость времени разбора входной цепочки от ее длины. Кроме того, конечные автоматы имеют важную особенность: любой недетерминированный КА всегда моет быть преобразован в детерминированный. Это существенно упрощает разработку программного обеспечения для распознавателя.

Простота и высокая скорость работы распознавателей определяют широкую область применения регулярных языков.

В компиляторах распознаватели на основе регулярных языков используются для лексического анализа текста исходной программы — выделения в нем простейших конструкций языка (лексем), таких как идентификаторы, строки, константы т. п. Это позволяет существенно сократить объем исходной информации и упрощает синтаксический разбор программы.

Кроме того, на основе регулярных языков функционируют многие командные процессоры, как в системном, так и в прикладном программном обеспечении. Для регулярных языков существуют развитые математически обоснованные методы, позволяющие облегчить создание распознавателей.

Принципы работы транслятора, компилятора, интерпретатора

Входными данными для работы транслятора служит программа на исходном языке программирования, которая называется *исходной программой.*

Транслятор является программой, переводящей исходную программу в эквивалентную ей программу на *результирующем (выходном) языке*. *Результирующая программа* строится по синтаксическим правилам выходного языка транслятора, а ее смысл определяется семантикой выходного языка.

Все составные части транслятора представляют собой динамически загружаемые библиотеки или модули со своими входными и выходными данными.

Эквивалентность исходной и результирующей программ этих двух программ означает совпадение их смысла с точки зрения семантики входного языка (для исходной программы) и семантики выходного языка (для результирующей программы). Без выполнения этого требования сам транслятор теряет всякий практический смысл.

Результатом работы транслятора будет результирующая программа, но только в том случае, если текст исходной программы является правильным — не содержит ошибок с точки зрения синтаксиса и семантики входного языка. Если исходная программа неправильная (содержит хотя бы одну ошибку), то

результатом работы транслятора будет сообщение об ошибке (как правило, с дополнительными пояснениями и указанием места ошибки в исходной программе).

Компилятор — это транслятор, осуществляющий перевод исходной программы в эквивалентную ей результирующую программу на языке машинных команд или на языке ассемблера. А результирующая программа транслятора, в общем случае может быть написана на любом языке.

Всякий компилятор является транслятором, но не наоборот — не всякий транслятор будет компилятором.

Результирующая программа компилятора называется *объектной программой.*

Вычислительная система, на которой выполняется результирующая (объектная) программа, созданная компилятором, называется *целевой вычислительной системой.* В это понятие входит не только архитектура аппаратных средств компьютера, но и операционная система, а зачастую и набор динамически подключаемых библиотек, необходимый для выполнения объектной программы. При этом следует помнить, что объектная программа ориентирована на целевую вычислительную систему, но не может быть непосредственно выполнена на ней без дополнительной обработки. Целевая вычислительная система не всегда является той же вычислительной системой, на которой работает сам компилятор.

Компиляторы - самый распространенный вид трансляторов. Практически все современные компиляторы создаются группами разработчиков с помощью компиляторов (чаще всего предыдущей версии компилятора этой же фирмы).

Понятия «транслятор» и «компилятор» принципиально отличаются от понятия интерпретатора.

Интерпретатор — это программа, которая воспринимает исходную программу на входном (исходном) языке и выполняет ее.

Интерпретатор, так же как и транслятор, анализирует текст исходной программы. Однако он не порождает результирующую программу, а сразу же выполняет исходную в соответствии с ее смыслом, заданным семантикой входного языка. Результатом работы интерпретатора будет результат, определенный смыслом исходной программы, в том случае, если эта программа синтаксически и семантически правильная с точки зрения входного языка программирования, или сообщение об ошибке в противном случае.

Чтобы исполнить исходную программу, интерпретатор должен преобразовать ее в язык машинных кодов. Однако полученные машинные коды не являются доступными — их не видит пользователь интерпретатора. Эти машинные коды порождаются интерпретатором, исполняются и уничтожаются по мере надобности — так, как того требует конкретная реализация интерпретатора. Пользователь же видит результат выполнения этих кодов, то есть результат выполнения исходной программы. (Требование об эквивалентности исходной программы и порожденных машинных кодов и в этом случае, безусловно, должно выполняться).

Этапы трансляции. Общая схема работы транслятора

На рисунке представлена общая схема работы компилятора. Из нее видно, что в целом процесс компиляции состоит из двух основных этапов — анализа и синтеза.

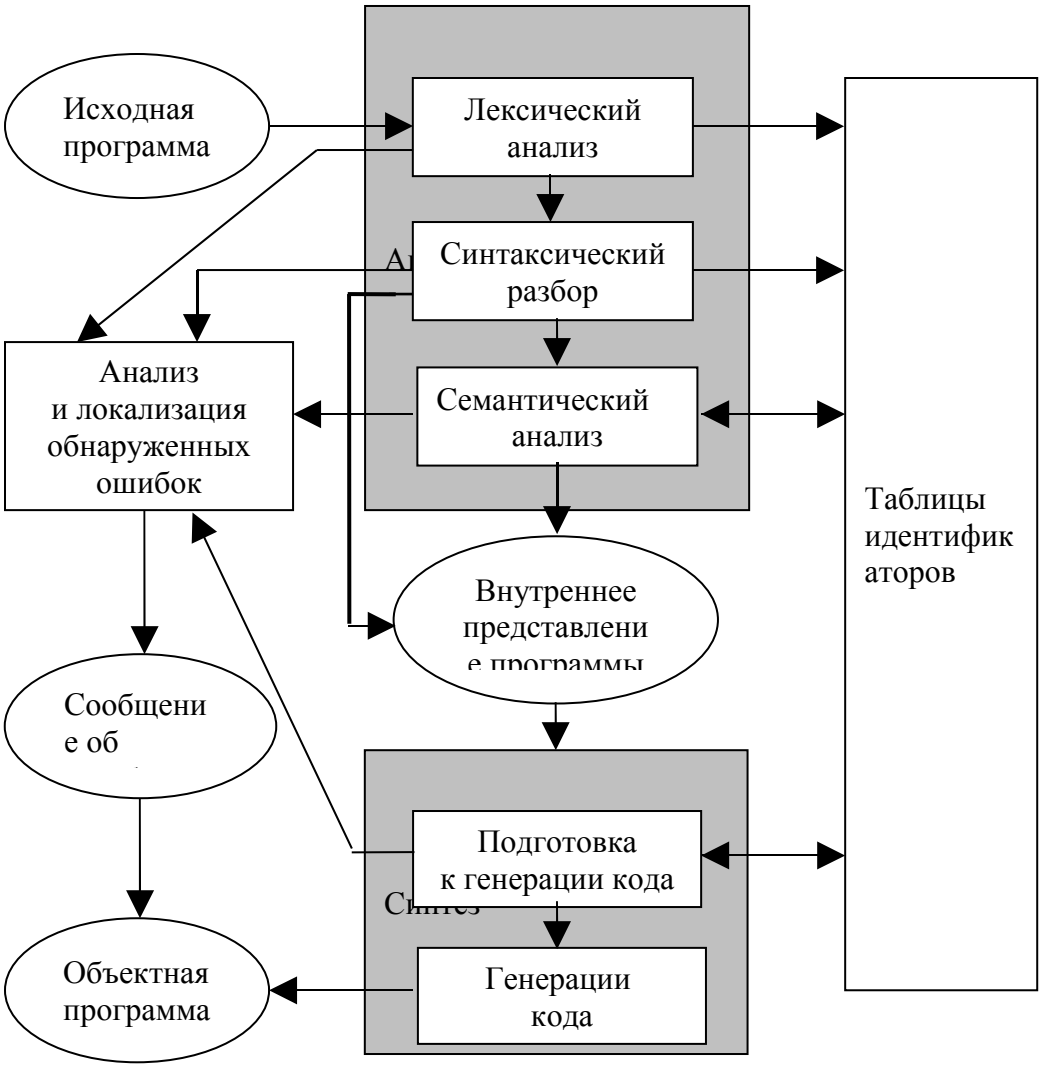


Рис. 3 Общая схема работы компилятора

На этапе анализа выполняется распознавание текста исходной программы, создание и заполнение таблиц идентификаторов. Результатом его работы служит внутреннее представление программы, понятное компилятору. На основании внутреннего представления программы и информации, содержащейся в таблице идентификаторов, порождается текст результирующей программы. Результатом этого этапа является объектный код.

В состав компилятора входит часть, ответственная за анализ и исправление ошибок. При наличии ошибок она должна максимально полно информировать пользователя о типе ошибки и месте ее возникновения, а в лучшем случае предложить пользователю вариант исправления ошибки.

Эти этапы, в свою очередь, состоят из более мелких этапов, называемых фазами компиляции. Состав фаз компиляции на рисунке приведен в самом об-

щем виде, их конкретная реализация и процесс взаимодействия могут различаться в зависимости от версии компилятора. Однако в том или ином виде все представленные фазы практически всегда присутствуют в каждом конкретном компиляторе.

Лексический анализ (сканер) — часть компилятора, которая читает литеры программы на исходном языке и строит из них слова (лексемы) исходного языка. На вход лексического анализатора поступает текст исходной программы, а выходная информация передается для дальнейшей обработки компилятором на этапе синтаксического разбора. С теоретической точки зрения лексический анализатор не является обязательной частью компилятора. Однако существуют причины, определяющие его присутствие практически во всех компиляторах.

Синтаксический разбор — основная часть компилятора на этапе анализа, выполняющая выделение синтаксических конструкций в тексте исходной программы, обработанном лексическим анализатором. На этой же фазе компиляции проверяется синтаксическая правильность программы.

Семантический анализ — это часть компилятора, проверяющая правильность текста исходной программы с точки зрения семантики входного языка. Кроме непосредственной проверки, выполняется преобразования текста, требуемые семантикой входного языка (например, добавление функций неявного преобразования типов). В различных реализациях компиляторов семантический анализ может частично входить в фазу синтаксического разбора, частично — в фазу подготовки к генерации кода.

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. Обычно в эту фазу входят действия, связанные с идентификацией элементов языка, распределением памяти и т. п.

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 (таблицы символов) — это специальным образом организованные наборы данных, служащие для хранения информации об элементах исходной программы, которые затем используются для порождения текста результирующей программы. В конкретной реализации компилятора может быть как одна, так и несколько таблиц идентификаторов. Элементами исходной программы, информацию о которых необходимо хранить в процессе компиляции, являются переменные, константы, функции и т. п. (конкретный состав набора элементов зависит от используемого входного языка программирования). Понятие «таблицы» не предполагает, что это хранилище

должно быть организовано в виде таблиц или других массивов информации. Представленное на рисунке деление процесса компиляции на фазы служит методическим целям и на практике может столь строго не соблюдаться.

На фазе лексического анализа лексемы выделяются из текста входной программы, поскольку они необходимы для следующей фазы синтаксического разбора. Синтаксический разбор и генерация кода могут выполняться одновременно. Таким образом, три фазы компиляции могут работать комбинированно, а вместе с ними может выполняться и подготовка к генерации кода.

Компилятор в целом с точки зрения теории формальных языков выполняет две основные роли:

1. распознаватель языка исходной программы. То есть, получив на вход цепочку символов входного языка, он проверяет ее принадлежность языку и выявляет правила, по которым эта цепочка построена (поскольку на вопрос о принадлежности сам ответ «да» или «нет» представляет мало интереса). Генератором цепочек входного языка выступает пользователь — автор исходной программы.
2. генератор для языка результирующей программы. Он должен построить на выходе цепочку выходного языка по определенным правилам, предполагаемым языком машинных команд или языком ассемблера. В случае машинных команд распознавателем этой цепочки будет выступать целевая вычислительная система, под которую создается результирующая программа.

Понятие прохода. Многопроходные и однопроходные компиляторы

В реальных компиляторах состав этих фаз компиляции может несколько отличаться от выше рассмотренного – некоторые из них могут быть разбиты на составляющие, другие, напротив, объединены в одну фазу. Порядок выполнения фаз компиляции также может меняться в разных вариантах компиляторов. В одном случае компилятор просматривает текст исходной программы, сразу выполняет все фазы компиляции и получает результат — объектный код. В другом варианте он выполняет над исходным текстом только некоторые из фаз компиляции и получает не конечный результат, а набор некоторых промежуточных данных. Эти данные затем снова подвергаются обработке, причем этот процесс может повторяться несколько раз. Реальные компиляторы, как правило, выполняют трансляцию текста исходной программы за несколько проходов.

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

В качестве внешней памяти могут выступать любые носители информации - оперативная память компьютера, накопители на магнитных дисках и т п.

Современные компиляторы стремятся максимально использовать для хранения данных оперативную память компьютера.

При выполнении каждого прохода компилятору доступна информация, полученная в результате всех предыдущих проходов. Информация, получаемая компилятором при выполнении проходов, недоступна пользователю.

Количество выполняемых проходов - важная техническая характеристика компилятора. Фирмы-разработчики компиляторов обычно указывают ее в описании своего продукта.

Разработчики стремятся максимально сократить количество проходов, выполняемых компиляторами. При этом увеличивается скорость работы компилятора, сокращается объем необходимой ему памяти. Однопроходный компилятор - это идеальный вариант. Однако, количество необходимых проходов определяется прежде всего грамматикой и семантическими правилами исходного языка. Чем сложнее грамматика языка и чем больше вариантов предлагают семантические правила — тем больше проходов будет выполнять компилятор. Например, именно поэтому обычно компиляторы с языка Pascal работают быстрее, чем компиляторы с языка C++ - грамматика Pascal более проста, а семантические правила более жесткие.

Однопроходные компиляторы — редкость, они возможны только для очень простых языков. Реальные компиляторы выполняют, как правило, от двух до пяти подходов. Наиболее распространены двух- и трехпроходные компиляторы, например: первый проход — лексический анализ, второй — синтаксический разбор и семантический анализ, третий — генерация и оптимизация кода. В современных системах программирования нередко первый проход компилятора (лексический анализ кода) выполняется параллельно с редактированием кода исходной программы.

Таблицы идентификаторов. Организация таблиц идентификаторов

Проверка правильности семантики и генерация кода требуют знания характеристик переменных, констант, функций и других элементов, встречающихся в программе на исходном языке. Все эти элементы в исходной программе, как правило, обозначаются идентификаторами. Выделение идентификаторов и других элементов исходной программы происходит на фазе лексического анализа. Их характеристики определяются на фазах синтаксического разбора, семантического анализа и подготовки к генерации кода. Состав возможных характеристик и методы их определения зависят от семантики входного языка. В любом случае компилятор должен иметь возможность хранить все найденные идентификаторы и связанные с ними характеристики в течение всего процесса компиляции, чтобы иметь возможность использовать их на различных фазах компиляции. Для этой цели в компиляторах 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 с одной или несколькими таблицам идентификаторов.

Состав информации, хранимой в таблице идентификаторов для каждого элемента исходной программы, зависит от семантики входного языка и типа элемента. Например, в таблицах идентификаторов может храниться следующая информация:

- для переменных:
  - имя переменной;
  - тип данных переменной;
  - область памяти, связанная с переменной;
- для констант:
  - название константы (если оно имеется);
  - значение константы;
  - тип данных константы (если требуется);
- для функций:
  - имя функции;

количество и типы формальных аргументов функции;

тип возвращаемого результата;

адрес кода функции.

Вне зависимости от реализации компилятора принцип его работы с таблицей идентификаторов остается одним и тем же — на различных фазах компиляции компилятор вынужден многократно обращаться к таблице для поиска информации и записи новых данных.

Компилятору приходится выполнять поиск необходимого элемента в таблице идентификаторов по имени чаще, чем помещать новый элемент в таблицу, потому что каждый идентификатор может быть описан только один раз, а использован — несколько раз.

Простейшие методы построения таблиц идентификаторов

Простейший способ организации таблицы состоит в добавлении новых элементов в порядке их поступления. Тогда таблица идентификаторов представляет собой неупорядоченный массив информации.

Поиск нужного элемента в таблице заключается в последовательном сравнении искомого элемента с каждым элементом таблицы. Тогда для поиска в таблице, содержащей N элементов, в среднем будет выполнено N/2 сравнений. Такой способ организации таблиц идентификаторов является неэффективным.

Поиск может быть выполнен более эффективно, если элементы таблицы упорядочены (отсортированы) согласно некоторому естественному порядку. Наиболее естественным является расположение элементов таблицы в прямом или обратном алфавитном порядке.

Эффективным методом поиска в упорядоченном списке является *бинарный* (или *логарифмический*) *поиск.*

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

Так как на каждом шаге число элементов, которые могут содержать искомый элемент, сокращается в 2 раза, максимальное число сравнений равно $1 + \log_2(N)$. Так при для N = 128 бинарный поиск потребует максимум 8 сравнений, а поиск в неупорядоченной таблице — в среднем 64 сравнения.

Метод называют «бинарным поиском», поскольку на каждом шаге объем рассматриваемой информации сокращается в 2 раза, а «логарифмическим» — поскольку время, затрачиваемое на поиск нужного элемента в массиве, имеет логарифмическую зависимость от общего количества элементов в нем.

Недостатком данного метода является требование упорядочивания элементов таблицы идентификаторов. Время упорядочивания напрямую зависит от числа элементов в массиве. Таблица идентификаторов зачастую просматривается компилятором еще до того, как она заполнена полностью, поэтому для построения такой таблицы можно пользоваться только алгоритмом прямого упорядоченного включения элементов.

Таким образом, при организации логарифмического поиска в таблице идентификаторов существенного сокращается время поиска нужного элемента за счет увеличения времени на помещение нового элемента в таблицу. Однако добавление новых элементов в таблицу идентификаторов происходит существенно реже, чем обращение к ним, этот метод следует признать более эффективным, чем метод организации неупорядоченной таблицы.

Построение таблиц идентификаторов по методу бинарного дерева

Таблица имеет форму бинарного дерева. Каждый узел дерева представляет собой элемент таблицы, причем корневой узел является первым элементом, встреченным при заполнении таблицы.

Как минимум, при добавлении нового идентификатора в таблицу компилятор должен проверить, существует ли там такой идентификатор, так как в большинстве языков программирования ни один идентификатор не может быть описан более одного раза. Следовательно, каждая операция добавления нового элемента влечет, как правило, не менее одной операции поиска.

Дерево называется бинарным, так как каждая вершина в нем может иметь не более двух ветвей. Для определенности будем называть ветви «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, то построение дерева закончено.

Шаг 2. Сделать текущим узлом дерева корневую вершину.

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

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

Шаг 5. Если у текущего узла существует левая вершина, то 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, то 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.

В качестве примера последовательность идентификаторов GA, Dl, M22, E, A12, BC, F.

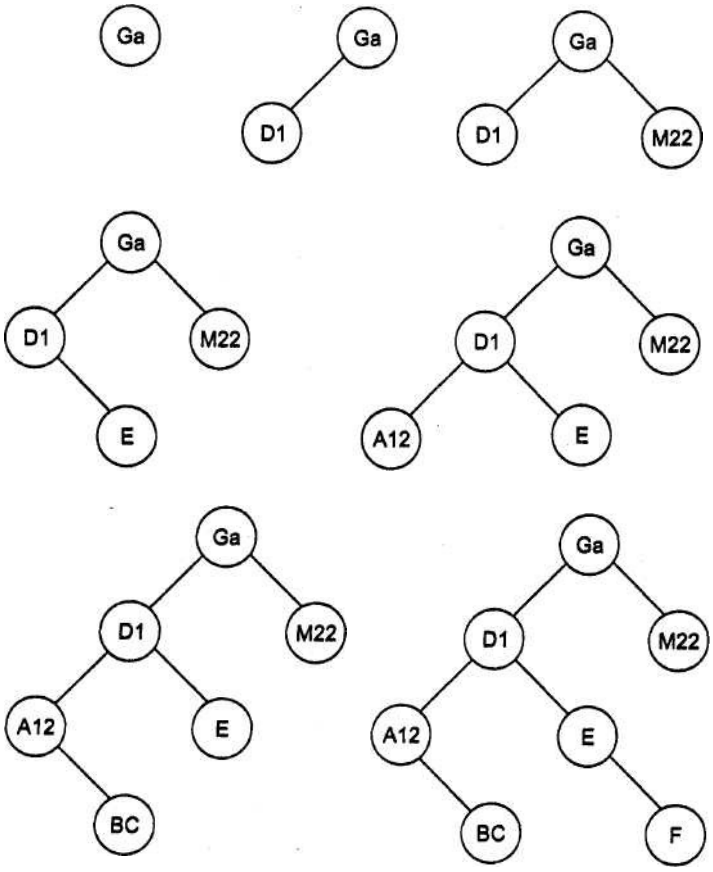


Рис. 4 Заполнение бинарного дерева для последовательности идентификаторов GA, D1, M22, E, A12, BC, F

Поиск нужного элемента в дереве выполняется по алгоритму:
Шаг 1. Сделать текущим узлом дерева корневую вершину.
Шаг 2. Сравнить искомый идентификатор с идентификатором, содержащимся в текущем узле дерева.
Шаг 3. Если идентификаторы совпадают, то искомый идентификатор найден, алгоритм завершается, иначе надо перейти к шагу 4.
Шаг 4. Если очередной идентификатор меньше, то перейти к шагу 5, иначе — перейти к шагу 6.

Шаг 5. Если у текущего узла существует левая вершина, то сделать ее текущим узлом и вернуться к шагу 2, иначе искомый идентификатор не найден, алгоритм завершается.
Шаг 6. Если у текущего узла существует правая вершина, то сделать ее текущим узлом и вернуться к шагу 2, иначе искомый идентификатор не найден, алгоритм завершается.

В данном методе число требуемых сравнений и форма дерева зависят от порядка, в котором поступают идентификаторы.

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

Принципы работы хэш-функций

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: $F(r) = n$, $r \in R$, $n \in Z$. Сам термин «хэш-функция» происходит от английского термина «hash function» (hash — «мешать», «смешивать», «путать»). Вместо термина «хэширование» иногда используются термины «рандомизация», «переупорядочивание».

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

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

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

Метод организации таблиц идентификаторов, основанный на использовании хэш-адресации, заключается в размещении каждого элемента таблицы в ячейке, адрес которой возвращает хэш-функция, вычисленная для этого элемента. Тогда, в идеальном случае, для размещения любого элемента в таблице идентификаторов достаточно вычислить его хэш-функцию и обратиться к

нужной ячейке массива данных. Первоначально таблица идентификаторов должна содержать пустые ячейки.

Для поиска элемента в таблице для него требуется вычислить хэш-функцию. Затем проверить, содержимое ячейки. Если она не пуста - элемент найден, иначе – не найден.

Этот метод весьма эффективен, поскольку как время размещения элемента в таблице, так и время его поиска определяются только временем, затрачиваемым на вычисление хэш-функции, которое в общем случае несопоставимо меньше времени, необходимого для выполнения многократных сравнений элементов таблицы. Метод имеет два недостатка: 1) неэффективное использование объема памяти под таблицу идентификаторов: размер массива для ее хранения должен соответствовать области значений хэш-функции, в то время как реально хранимых в таблице идентификаторов может быть существенно меньше; 2) необходимость соответствующего разумного выбора хэш-функции.

Построение таблиц идентификаторов на основе хэш-функций

Существуют различные варианты хэш-функций. Самой простой хэш-функцией для символа является код внутреннего представления в компьютере литеры символа. Эту хэш-функцию можно использовать и для цепочки символов, выбирая первый символ в цепочке. Так, если двоичное ASCII-представление символа A есть двоичный код 001000012, то результатом хэширования идентификатора ATable будет код 001000012.

Данная хэш-функция не является удовлетворительной, т.к. при ее использовании двум различным идентификаторам, начинающимся с одной и той же буквы, будет соответствовать одно и то же значение хэш-функции. Тогда при хэш-адресации в одну ячейку таблицы идентификаторов по одному и тому же адресу должны быть помещены два различных идентификатора, что явно невозможно. Ситуация, когда двум или более идентификаторам соответствует одно и то же значение функции, называется *коллизией.*

Хэш-функция, допускающая коллизии, не может быть напрямую использована для хэш-адресации в таблице идентификаторов. Причем для этого достаточно получить хотя бы один случай коллизии на всем множестве идентификаторов.

Для полного исключения коллизий хэш-функция должна быть взаимно однозначной: каждому элементу из области определения хэш-функции должно соответствовать одно значение из ее множества значений, и каждому значению из множества значений этой функции должен соответствовать только один элемент из области ее определения. Тогда любым двум произвольным элементам из области определения хэш-функции будут всегда соответствовать два различных ее значения. Теоретически, для идентификаторов такую хэш-функцию построить можно, так как и область определения хэш-функции (все возможные имена идентификаторов), и область ее значений (целые неотрицательные числа) являются бесконечными счетными множествами.

В реальности область значений любой хэш-функции ограничена размером доступного адресного пространства в данной архитектуре компьютера. Множество адресов любого компьютера с традиционной архитектурой может быть велико, но всегда конечно, то есть ограничено. Организовать взаимно однозначное отображение бесконечного множества на конечное, даже теоретически, невозможно. Даже с учетом того, что длина принимаемой во внимание части имени идентификатора в реальных компиляторах также практически ограничена — обычно в пределах от 32 до 128 символов (то и область определения хэш-функции конечна). Но и тогда количество элементов в конечном множестве, составляющем область определения функции, будет превышать их количество в конечном множестве области значений функции. Следовательно, невозможно избежать возникновения коллизий.

Существует несколько способов для решения проблемы. Одним из них является метод *рехэширования* (или расстановка). Согласно этому методу, если для элемента A адрес h(A), вычисленный с помощью хэш-функции h, указывает на уже занятую ячейку, то необходимо вычислить значение функции $n_1 = h_1(A)$ и проверить занятость ячейки по адресу n_1. Если и она занята, то вычисляется значение $h_2(A)$ и так до тех пор, пока либо не будет найдена свободная ячейка, либо очередное значение $h_i(A)$ совпадет с h(A). В последнем случае считается, что таблица идентификаторов заполнена, и места в ней больше нет — выдается информация об ошибке размещения идентификатора в таблице. Такую таблицу идентификаторов можно организовать по следующему алгоритму размещения элемента:

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

Шаг 2. Если ячейка по адресу n пустая, то поместить в нее элемент A и завершить алгоритм, иначе i:= 1 и перейти к шагу 3.

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

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

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

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

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

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

Алгоритмы размещения и поиска элемента схожи по выполняемым операциям. Поэтому они будут иметь одинаковые оценки времени, необходимого для их выполнения.

При такой организации таблиц идентификаторов в случае возникновения коллизии алгоритм размещает элементы в пустых ячейках таблицы, выбирая их определенным образом. При этом элементы могут попадать в ячейки с адресами, которые потом будут совпадать со значениями хэш-функции, что приведет к возникновению новых, дополнительных коллизий. Таким образом, количество операций, необходимых для поиска или размещения в таблице элемента, зависит от степени заполнения таблицы.

Для организации таблицы идентификаторов по методу рехэширования необходимо определить хэш-функцию h_i для каждого i. Чаще всего функции h_i определяют как некоторые модификации хэш-функции h. Например, самым простым методом вычисления функции $h_i(A)$ является ее организация в виде $h_i(A) = (h(A) + p_i) \bmod N_m$, где p_i — некоторое вычисляемое целое число, а N_m — максимальное значение из области значений хэш-функции h. В свою очередь, самым простым подходом здесь будет положить $p_i = i$, т.е. $h_i(A) = (h(A) + i) \bmod N_m$. В этом случае при совпадении значений хэш-функции для каких-либо элементов поиск свободной ячейки в таблице начинается последовательно от текущей позиции, заданной хэш-функцией h(A).

Этот способ нельзя признать особенно удачным — при совпадении хэш-адресов элементы в таблице начинают группироваться вокруг них, что увеличивает число необходимых сравнений при поиске и размещении. Среднее время поиска элемента в такой таблице в зависимости от числа операций сравнения можно оценить следующим образом:

$T_п = O((1 - Lf/2)/(l - Lf))$.

где Lf — (load factor) степень заполненности таблицы идентификаторов — отношение числа занятых ячеек N таблицы к максимально допустимому числу элементов в ней: $Lf = N/N_m$.

Даже такой примитивный метод рехеширования является достаточно эффективным средством организации таблиц идентификаторов при частном заполнении таблицы. Имея, например, заполненную на 90% таблицу для 1024 идентификаторов, в среднем необходимо выполнить 5.5 сравнений для поиска одного идентификатора, в то время как даже логарифмический поиск дает в среднем от 9 до 10 сравнений. Сравнительная эффективность метода будет еще выше при росте числа идентификаторов и снижении заполненности таблицы.

Среднее время на помещение одного элемента в таблицу и на поиск элемента в таблице можно снизить, если применить более совершенный метод рехэшировашия: использование в качестве p_i для функции $h_i (A) = (h(A) + p_i) \bmod N_m$ последовательности псевдослучайных целых чисел p1, p_2, ..., p_k. При хорошем выборе генератора псевдослучайных чисел длина последовательности k будет $k = N_m$. Тогда среднее время поиска одного элемента таблице можно оценивается как:

$En = O((1/Lf)*\log_2(l- Lf))$.

Существуют и другие методы организации функций рехеширования $h_i(A)$, основанные на квадратичных вычислениях или, например, на вычислении по формуле: $h_i(A) = (h(A)*i) \bmod N_m$, если N_m — простое число. В целом,

рехеширование позволяет добиться неплохих результатов, но требование частичного заполнения таблицы ведет к неэффективному использованию объема доступной памяти.

Построение таблиц идентификаторов по методу цепочек

Частичное заполнение таблицы идентификаторов при применении хэш-функций ведет к неэффективному использованию всего объема памяти, доступного компилятору. Причем объем неиспользуемой памяти будет тем выше, чем больше информации хранится для каждого идентификатора. Этого недостатка можно избежать, если дополнить таблицу идентификаторов некоторой промежуточной хэш-таблицей.

В ячейках хэш-таблицы может храниться либо пустое значение, либо значение указателя на некоторую область памяти из основной таблицы идентификаторов. Тогда хэш-функция вычисляет адрес, по которому происходит обращение сначала к хэш-таблице, а потом уже через нее по найденному адресу — к самой таблице идентификаторов. Если соответствующая ячейка таблицы идентификаторов пуста, то ячейка хэш-таблицы будет содержать пустое значение. Тогда вовсе не обязательно иметь в самой таблице идентификаторов ячейку для каждого возможного значения хэш-функции — таблицу можно сделать динамической так, чтобы ее объем рос по мере заполнения (первоначально таблица идентификаторов не содержит ни одной ячейки, а все ячейки хэш-таблицы имеют пустое значение).

Такой подход имеет два преимущества: во-первых, таблицу идентификаторов не заполняется пустыми значениями; во-вторых, количество ячеек в таблице равно числу идентификаторов. Пустые ячейки будут только в хэш-таблице. Объем неиспользуемой памяти не будет зависеть от объема информации, хранимой для каждого идентификатора, — для каждого значения хэш-функции будет расходоваться только память, необходимая для хранения одного указателя на основную таблицу идентификаторов.

На основе этой схемы можно реализовать еще один способ организации таблиц идентификаторов с помощью хэш-функций - «метод цепочек». Для этого в таблицу идентификаторов для каждого элемента добавляется еще одно поле, в котором может содержаться ссылка на любой элемент таблицы. Первоначально это поле всегда пустое (никуда не указывает). Необходимо создание специальной переменной, которая всегда указывает на первую свободную ячейку основной таблицы идентификаторов (первоначально — указывает на начало таблицы).

Метод цепочек работает по следующему алгоритму:

Шаг 1. Во все ячейки хэш-таблицы поместить пустое значение, таблица идентификаторов пуста, переменная FreePtr (указатель первой свободной ячейки) указывает на начало таблицы идентификаторов; i := 1.

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

Шаг3. Положить j:= 1, выбрать из хэш-таблицы адрес ячейки таблицы идентификаторов m_j и перейти к шагу 4.

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

Шаг 5. Добавить в таблицу идентификаторов новую ячейку, записать в нее информацию для элемента A_i (поле ссылки должно быть пустым), в переменную FreePtr поместить адрес за концом добавленной ячейки. Если больше нет идентификаторов, которые надо разместить в таблице, то выполнение алгоритма закончено, иначе i := i + 1 и перейти к шагу 2.

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

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

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

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

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

Комбинированные способы построения таблиц идентификаторов

В реальных компиляторах практически всегда, так или иначе, используется хэш-адресация. Алгоритм применяемой хэш-функции обычно составляет «ноу-хау» разработчиков компилятора. Обычно при разработке хэш-функции создатели компилятора стремятся свести к минимуму количество возникающих коллизий не на всем множестве возможных идентификаторов, а на тех их вариантах, которые наиболее часто встречаются во входных программах. Конечно, принять во внимание все допустимые исходные программы невозможно. Чаще всего выполняется статистическая обработка встречающихся имен

идентификаторов на некотором множестве типичных исходных программ, а также принимаются во внимание соглашения о выборе имен идентификаторов, общепринятые для входного языка. Хорошая хэш-функция — это шаг к значительному ускорению работы компилятора, поскольку обращение к таблицам идентификаторов выполняются многократно на различных фазах компиляции.

То, какой конкретно метод применяется в компиляторе для организации таблиц идентификаторов, зависит от реализации компилятора. Один и тот же компилятор может иметь даже несколько разных таблиц идентификаторов, организованных на основе различных методов.

Как правило, применяются комбинированные методы. В этом случае, как и для метода цепочек, в таблице идентификаторов организуется специальное дополнительное поле ссылки. Но в отличие от метода цепочек оно имеет несколько иное значение. При отсутствии коллизий для выборки информации из таблицы используется хэш-функция, поле ссылки остается пустым. Если же возникает коллизия, го с помощью поля ссылки организуется поиск идентификаторов, для которых значения хэш-функции совпадают, по одному из рассмотренных выше методов: неупорядоченный список, упорядоченный список или же бинарное дерево. При хорошо построенной хэш-функции коллизии будут возникать редко, поэтому количество идентификаторов, для которых значения хэш-функции совпали, будет не столь велико. Тогда и время поиска одного среди них будет незначительным (в принципе, при высоком качестве хэш-функции подойдет даже перебор по неупорядоченному списку).

Такой подход имеет преимущество по сравнению с методом цепочек, поскольку не требует использования промежуточной хэш-таблицы. Недостатком метода является необходимость работы с динамически распределяемыми областями памяти. Эффективность такого метода, очевидно, в первую очередь зависит от качества применяемой хэш-функции, а во вторую — от метода организации дополнительных хранилищ данных.

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

Лексические анализаторы (сканеры).

Лексема (лексическая единица языка) — это структурная единица языка, состоящая из элементарных символов языка и не содержащая в своем составе других структурных единиц языка. Лексемами языков программирования являются идентификаторы, константы, ключевые слова языка, знаки операций и т. п.

Лексический анализатор (или сканер) — это часть компилятора, которая читает исходную программу и выделяет в ее тексте лексемы входного языка. На вход лексического анализатора поступает текст исходной программы, а

выходная информация передается для дальнейшей обработки компилятором на этапе синтаксического анализа и разбора.

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

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

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

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

Функции, выполняемые лексическим анализатором, и состав лексем, которые он выделяет, могут меняться в зависимости от реализации компилятора. То, какие функции должен выполнять лексический анализатор и какие типы лексем он должен выделять во входной программе, а какие оставлять для этапа синтаксического разбора, решают разработчики компилятора. В основном лексические анализаторы выполняют исключение комментарий из текста исходной программы, незначащих пробелов, символов табуляции и перевода строки, а также выделение лексем следующих типов: идентификаторов, строковых, символьных и числовых констант, ключевых (служебных) слов входного языка, знаков операций и разделителей.

Результатом работы лексического анализатора является перечень всех найденных в тексте исходной программы лексем с учетом характеристик каждой лексемы. Этот перечень лексем, представленный в виде таблицы, называется *таблицей лексем.* Каждой лексеме в ней соответствует некий уникальный условный код, зависящий от типа лексемы, и дополнительная служебная информация. Кроме того, информация о некоторых типах найденных лексем должна помещаться в таблицу идентификаторов.

Любая лексема может встречаться в таблице лексем любое количество раз. Таблица идентификаторов содержит только определенные типы лексем — идентификаторы и константы. В нее не попадают такие лексемы, как ключевые (служебные) слова входного языка, знаки операций и разделители. Кроме того, каждая лексема (идентификатор или константа) может встречаться в таблице идентификаторов только один раз. Также можно отметить, что лексемы в таблице лексем обязательно располагаются в том же порядке, как и в исход-

ной программе, а в таблице идентификаторов лексемы располагаются в любом порядке так, чтобы обеспечить удобство поиска.

В таблице лексем помещается сама лексема, ее тип и некое кодовое значение, используемое компилятором.

Принципы построения лексических анализаторов

Лексический анализатор имеет дело с такими объектами, как различного рода константы и идентификаторы, в том числе и ключевые слова. Язык констант и идентификаторов является регулярным — то есть может быть описан с помощью регулярных грамматик.

Вспомним, что *грамматикой* называют описание способа построения предложений некоторого языка. Грамматика – математическая система, определяющая язык. Формально грамматика определяется как четверка:

1. множество терминальных символов VT,
2. множество нетерминальных символов VN,
3. множество правил (продукций) грамматики вида $\alpha \to \beta$, где α, β цепочки языка $\in VN \bigcup VT$, причем α не может быть пустой цепочкой.
4. начальный символ языка грамматики.

Множество терминальных и нетерминальных символов не пересекаются. Начальный символ грамматики всегда нетерминальный символ. Множество терминальных символов содержит символы, входящие в алфавит языка, порождаемого грамматикой. Символы этого множества встречаются только в цепочках правых частей правил. Множество нетерминальных символов содержат символы, определяющие понятия, слова, конструкции языка. Каждый символ этого множества может встречаться как в левой, так и в правой частей правил грамматики. Но он обязан хотя бы один раз быть в левой части хотя бы одного правила.

Регулярные грамматики используются для описания простейших конструкций языков программирования: идентификаторов, констант, строк, комментариев и т.д.

Распознавателями для регулярных языков являются конечные автоматы. Существуют правила, с помощью которых для любой регулярной грамматики можно построен конечный автомат, распознающий цепочки языка, заданного этой грамматикой. Он дает ответ на вопрос о том, принадлежит или нет цепочка языку, заданному автоматом. Однако в общем случае задача лексического анализатора несколько шире. Кроме этого, он должен:

определить границы лексем, которые в тексте исходной программы явно не указаны;

выполнить действия для сохранения информации об обнаруженной лексеме (или выдать сообщение об ошибке, если лексема неверна).

Определение границ лексем

Выделение границ лексем является нетривиальной задачей. Ведь в тексте исходной программы лексемы не ограничены никакими специальными символами. Определение границ лексем — это выделение тех строк в общем потоке входных символов, для которых надо выполнять распознавание.

В большинстве компиляторов лексический и синтаксический анализаторы — это взаимосвязанные части. Возможны два принципиально различных метода организации взаимосвязи лексического и синтаксического анализа: последовательный и параллельный.

При последовательном варианте лексический анализатор просматривает весь текст исходной программы от начала до конца и преобразует его в таблицу лексем. Таблица лексем заполняется сразу полностью, компилятор использует ее для последующих фаз компиляции, но в дальнейшем не изменяет. Дальнейшую обработку таблицы лексем выполняют следующие фазы компиляции. Если в процессе разбора лексический анализатор не смог правильно определить тип лексемы, то считается, что исходная программа содержит ошибку.

При параллельном варианте лексический анализ текста исходной программы выполняется поэтапно, по шагам. Лексический анализатор выделяет очередную лексему в исходном коде и передает ее синтаксическому анализатору. Синтаксический анализатор, выполнив разбор очередной конструкции языка, может подтвердить правильность найденной лексемы и обратиться к лексическому анализатору за следующей лексемой либо же отвергнуть найденную лексему. Во втором случае он может проинформировать лексический анализатор о том, что надо вернуться назад к уже просмотренному ранее фрагменту исходного кода и сообщить ему дополнительную информацию о том, какого типа лексему следует ожидать.

Взаимодействуя между собой, таким образом, лексический и синтаксические анализаторы могут перебрать несколько возможных вариантов лексем, и если ни один из них не подойдет, будет считаться, что исходная программа содержит ошибку. Только после того, как синтаксический анализатор успешно выполнит разбор очередной конструкции исходного языка (обычно такой конструкцией является оператор исходного языка), лексический анализатор помещает найденные лексемы в таблицу лексем и в таблицу идентификаторов и продолжает разбор дальше в том же порядке.

Последовательная работа лексического и синтаксического анализаторов представляет собой самый простой вариант их взаимодействия. Она проще в реализации и обеспечивает более высокую скорость работы компилятора. Поэтому разработчики компиляторов стремятся организовать взаимодействие лексического и синтаксического анализаторов именно таким образом.

Для большинства языков программирования границы лексем распознаются по заданным терминальным символам. Эти символы — пробелы, знаки операций, символы комментариев, а также разделители (запятые, точки с запятой и т. п.) Набор таких терминальных символов зависит от синтаксиса входного языка. Важно отметить, что знаки операций сами также являются лексемами, и необходимо пропустить их при распознавании текста.

Но для многих языков программирования на этапе лексического анализа может быть недостаточно информации для однозначного определения типа и границ очередной лексемы. Однако даже и тогда разработчики компиляторов стремятся избежать параллельной работы лексического и синтаксического анализаторов. В ряде случаев помогает принцип выбора из всех возможных лексем лексемы наибольшей длины: очередной символ из входного потока данных добавляется в лексему всегда, когда он может быть туда добавлен. Как только символ не может быть добавлен в лексему, то считается, что он является границей лексемы и началом следующей лексемы (если символ не является пустым разделителем — пробелом, символом табуляции или перевода строки, знаком комментария).

Разработчики компиляторов сознательно идут на то, что отсекают некоторые правильные, но не вполне читаемые варианты исходных программ. Попытки усложнить лексический распознаватель неизбежно приведут к необходимости его взаимосвязи с синтаксическим разбором, что потребует организации их параллельной работы.

Большинство современных широко распространенных языков программирования, таких как C и Pascal, тем не менее, позволяют построить лексический анализ по более простому, последовательному методу.

Выполнение действий, связанных с лексемами

Выполнение действий в процессе распознавания лексем представляет для лексического анализатора гораздо меньшую проблему, чем определение границ лексем. Фактически конечный автомат, который лежит в основе лексического анализатора, должен иметь не только входной язык, но и выходной. Он должен не только уметь распознать правильную лексему на входе, но и породить связанную с ней последовательность символов на выходе. В такой конфигурации конечный автомат преобразуется в конечный преобразователь.

Для лексического анализатора действия по обнаружению лексемы могут трактоваться несколько шире, чем только порождение цепочки символов выходного языка. Он должен уметь выполнять такие действия, как запись найденной лексемы в таблицу лексем, поиск ее в таблице идентификаторов и запись новой лексемы в таблицу идентификаторов. Набор действий определяется реализацией компилятора. Обычно эти действия выполняются сразу же при обнаружении конца распознаваемой лексемы.

В конечном автомате, лежащем в основе лексического анализатора, эти действия можно отобразить довольно просто — достаточно иметь возможность с каждым переходом на графе автомата (или в функции переходов автомата) связать выполнение некоторой произвольной функции f(q,a), где q — текущее состояние автомата, a — текущий входной символ. Функция f(q,a) может выполнять любые действия, доступные лексическому анализатору:
помещать новую лексему в таблицу лексем;
проверять наличие найденной лексемы в таблице идентификаторов;
добавлять новую лексему в таблицу идентификаторов;

выдавать сообщения пользователю о найденных ошибках и предупреждения об обнаруженных неточностях в программе;
прерывать процесс компиляции.

Регулярные и автоматные грамматики

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 следующем: там, 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. Это значит, что для любого языка, который задан регулярной грамматикой, можно построить автоматную грамматику, определяющую почти эквивалентный язык (обратное утверждение очевидно).

Существует алгоритм, который позволяет преобразовать произвольную регулярную грамматику к автоматному виду — то есть построить эквивалентную ей автоматную грамматику. Этот алгоритм рассмотрен ниже. Он является исключительно полезным, поскольку позволяет существенно облегчить построение распознавателей для регулярных грамматик.

Преобразование регулярной грамматики к автоматному виду

Имеется регулярная грамматика G(VT,VN,P,S), необходимо преобразовать ее в почти эквивалентную автоматную грамматику G'(VT,VN',P',S'). Рассматривать будем случай леволинейной грамматики. Для праволинейной легко построить аналогичный алгоритм. Алгоритм преобразования заключается в следующей последовательности действий:

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

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

Если встречаются правила вида A -> $Ba_1a_2,\ldots, a_n$, $n > 1$, $A,B \in VN$, $\forall\ n > i > 0$: $a_i \in VT$, то во множество нетерминальных символов VN' грамматики G' добавляются символы $A_1, A_2, \ldots, A_{n-1}$, 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 ->A_1a_2
A_1, -> Ba_1
Если встречаются правила вида A -> $a_1,a_2,...$, a_n, n > 1, A,B∈ VN, ∀ n >i> 0: a_i∈ 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 -> A_{n-1} a_n
A_{n-1} -> A_{n-2} a_{n-1}
. . .
A_2 ->A_1a_2
A_1, -> a_1

Если встречаются правила вида A -> B или вида A -> λ , то они переносятся во множество правил P' грамматики G' без изменений.

Шаг3. Просматривается множество правил P' грамматики G'. В нем ищутся правила вида A -> B или вида A -> λ .

Если находится правило вида A -> B, то просматривается множество правил P’ грамматики G'. Если в нем присутствуют правила вида B -> C, B -> Ca, B -> a или B ->λ , то в него добавляются правила вида A -> C, A -> Ca, A -> a и A -> λ соответственно, ∀ A,B,C∈ VN', ∀ a∈ VT' (при этом следует учитывать, что в грамматике не должно быть совпадающих правил, и если какое-то правило уже присутствует в грамматике G', то повторно его туда добавлять не следует). Правило A -> B удаляется из множества правил P'.

Если находится правило вида A -> λ (и символ A не является целевым символом S), то просматривается множество правил P' грамматики G'. Если в нем присутствуют правила вида B -> A или B -> Aa, то в него добавляются правил: вида B -> λ и B -> a соответственно, ∀ A,B,C∈ VN', ∀ a∈ VT' (при этом следует учитывать, что в грамматике не должно быть совпадающих правил, и если какое-то правило уже присутствует в грамматике G', то повторно его туда добавлять не следует). Правило A —> λ удаляется из множества правил P'.

Шаг4. Если на шаге 3 было найдено хотя бы одно правило вида A -> B или A ->λ 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, в принципе, можно не выполнять, если грамматика не содержит правил вида A -> B (такие правила называются цепными) или вида A -> λ (такие правила называются λ -правилами).

Реальные регулярные грамматики обычно не содержат правил такого вида.

Способы задания регулярных языков

Регулярные языки являются удобным типом языков. Для них разрешимы многие проблемы, неразрешимые для других языков:

- проблема эквивалентности двух языков,
- проблема принадлежности языку заданной цепочки символов,
- проблема пустоты языка.

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

1) регулярными грамматиками (праволинейными или леволинейными)
2) конечные автоматы
3) регулярные множества.

Все три способа равноправны. Существуют алгоритмы, которые позволяют для регулярного языка, заданного одним из способов, построить другой способ задания того же самого языка. Для этих способов верны следующие утверждения.

Утверждение1. Язык является регулярным множеством тогда и только тогда, когда он задан леволинейной (праволинейной) грамматикой.

Утверждение2. Язык может быть задан леволинейной (праволинейной) грамматикой тогда и только тогда, когда он является регулярным множеством.

Утверждение3. Язык является регулярным множеством тогда и только тогда, когда он задан с помощью конечного автомата.

Утверждение4. Язык распознается с помощью конечного автомата тогда и только тогда, когда он задан регулярным множеством.

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

Конечные автоматы

Под автоматом понимают не реально существующее устройство, а некоторую математическую модель, свойства и поведение которой можно изучать. Конечным автомат является простейшим из моделей теории автоматов и служит управляющим устройством для всех остальных автоматов.

Конечные автоматы применяются при построении компиляторов, благодаря следующим свойствам:

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

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 отвергнута.

Таким образом, конечный автомат задается пятеркой вида $M(Q, V, \delta, q_0, F)$, 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, $q_0 \in Q$,

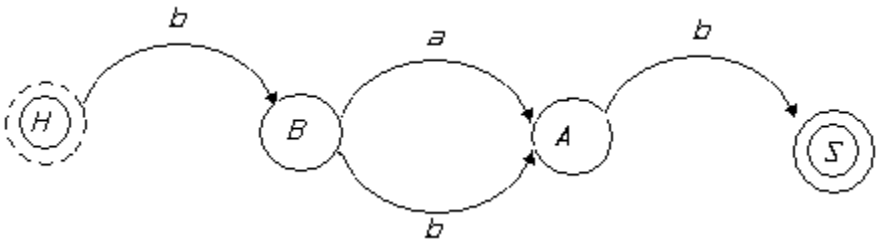
F – непустое множество конечных состояний автомата.

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

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

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

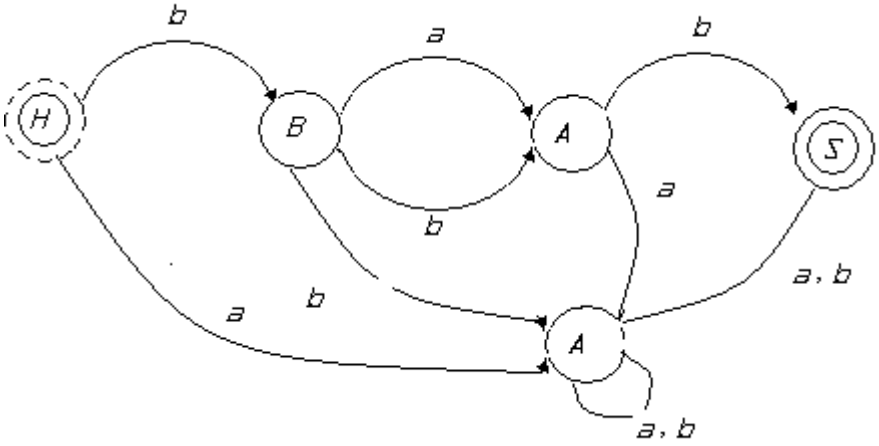
КА часто представляют в виде диаграммы или графа переходов автоматов. Граф переходов КА – направленный граф, вершины которого помечены символами состояний КА, а дуга, помечена символом a, если определена соответствующая функция перехода δ. Начальное и конечное состояние автомата помечаются специальным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



Таким образом, на рисунке задан КА *M({H, A, B, S}, {a, b}, δ, H, {s});*

δ: δ(H, b)=B, δ(B, a)=A, δ(A, b)={B, S}

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



Другой способ представления КА – 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.

	a	***b***	
H	E	B	0
B	A	A	0
A	E	S	1
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. Автоматы, рассмотренные ранее, являются недетерминированными.

Доказано, что для любого КА можно построить эквивалентный ему ДКА. Моделировать работу ДКА существенно проще, чем работу произвольного ДКА. Поэтому всегда стремятся сделать это преобразование. При построении компиляторов чаще всего используют полностью определенный детерминированный конечный автомат.

Многие КА можно минимизировать. Минимизация КА заключается в построении эквивалентного КА с меньшим числом состояний.

Для минимизации автомата используется алгоритм построения эквивалентных состояний КА. Два различных состояния q и q' в конечном автомате $M(Q, V, \delta, q_0, F)$ 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 $M(Q, V, \delta, q_0, F)$ 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) = $\{F , F\text{-}Q\}$.

Алгоритм построения эквивалентных состояний:

1. n=0, строим R(0).
2. n=n+1, строим R(n) на основе R(n - 1). В класс эквивалентности на шаге n входят те состояния, которые по одинаковым символам переходят n-1 эквивалентные состояния.
3. Если R(n)=R(n-1), работа закончена, иначе перейти к шагу 2.

Доказано, что алгоритм построения множества классов эквивалентности завершится максимум за n=m-2, где m – общее количество состояний автомата.

Алгоритм минимизации КА заключается в следующем:

1. Из автомата исключаются все недостижимые состояния.
2. Строятся классы эквивалентности автомата.
3. Классы эквивалентности состояний исходного КА становятся состояниями результирующего минимизированного КА.
4. Функции переходов результирующего КА очевидным образом строится на основе функции переходов исходного КА.

Для этого алгоритма доказано: во-первых, что он строит минимизированный КА, эквивалентный заданному КА; во-вторых, что он строит КА с минимально возможным числом состояний (минимальный КА).

Синтаксические анализаторы

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

поиск и выделение синтаксических конструкции в тексте исходной программы;

установка типа и проверка правильности каждой синтаксической конструкции;

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

Синтаксический анализатор — это основная часть компилятора на этапе анализа. Без выполнения синтаксического разбора работа компилятора бессмысленна, в то время как лексический разбор, в принципе, является необязательной фазой. Синтаксический анализатор воспринимает выход лексического анализатора и разбирает его в соответствии с грамматикой входного языка. Но эти же конструкции могут распознаваться и синтаксическим анализатором. На практике не существует жесткого правила, определяющего, какие конструкции должны распознаваться на лексическом уровне, а какие - на синтаксическом. Обычно это определяет разработчик компилятора исходя из технологических аспектов программирования, а также синтаксиса и семантики входного языка.

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

Чаще всего регулярные грамматики применимы к языкам ассемблера, а синтаксис языков высокого уровня основан на грамматике КС-языков.
Главную роль в том, как функционирует синтаксический анализатор и какой алгоритм лежит в его основе, играют принципы построения распознавателей для КС-языков. Без применения этих принципов невозможно выполнить эффективный синтаксический разбор предложений входного языка.

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

Автоматы с магазинной памятью

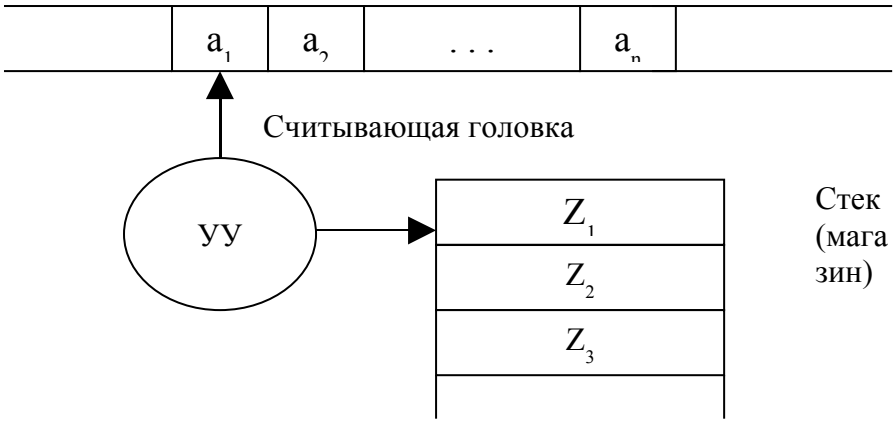
Контекстно-свободными (КС) называются языки, определяемые грамматиками типа G(VT,VN,P,S), в которых правила P имеют вид: A –> β , где A$\in$ VN и $\beta \in V^*$, V=VT$\cup$ VN. Распознавателями КС-языков служат автоматы с магазинной пастью *(МП-автоматы)*.

В общем виде МП-автомат можно определить следующим образом:
R (Q, V, Z, δ , qo, Zo, F), где
Q - множество состояний автомата;
V - алфавит входных символов автомата;
Z — специальный конечный алфавит магазинных символов автомата,
δ — функция переходов автомата,
$q_o \in$ Q — начальное состояние автомата;
$z_o \in$ Z — начальный символ магазина;
F$\in$ Q — множество конечных состояний.

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

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

Конфигурация МП-автомата описывается текущим его состоянием q, цепочкой непрочитанных символов α на входе автомата и содержимого стека ω.



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 также понятие расширенного МП-автомата. *Расширенный МП-автомат* может заменять цепочку символов конечной длины в верхней части стека на другую цепочку символов конечной длины. 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.

Построение синтаксических анализаторов

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

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

Кроме того, процесс построения синтаксического анализатора гораздо сложнее аналогичного процесса для лексического анализатор, поскольку КС-грамматики и МП-автоматы сложнее, чем регулярные грамматики и КА.

Построение синтаксического анализатора — это более творческий процесс, чем построение лексического анализатора, поскольку он не всегда может быть полностью формализован.

Имея грамматику входного языка, разработчик синтаксического анализатора должен в первую очередь выполнить ряд формальных преобразований над этой грамматикой, облегчающих построение распознавателя. После этого он должен проверить, подпадает ли полученная грамматика под один из известных классов КС-языков, для которых существуют линейные распознаватели. Если такой класс найден, можно строить распознаватель (если найдено несколько классов — выбрать тот, для которого построение распознавателя проще, либо построенный распознаватель обладает лучшими характеристиками). Если же такой класс КС-языков найти не удалось, то разработчик должен попытаться выполнить над грамматикой некоторые преобразования, чтобы привести ее к одному из известных классов. Вот эти преобразования не могут быть описаны формально, и в каждом конкретном случае разработчик должен попытаться найти их сам (иногда преобразования имеет смысл искать даже в том случае, когда грамматика подпадает под один из известных классов КС-языков, с целью найти другой класс, для которого можно построить лучший по характеристикам распознаватель). Только в том случае, когда в результате всех этих действий не удалось найти соответствующий класс КС-языков, разработчик вынужден строить универсальный распознаватель. Характеристики такого распознавателя будут существенно хуже, чем у линейного распознавателя, — в лучшем случае удается достичь квадратичной зависимости времени работы распознавателя от длины входной цепочки. Такие случаи бывают редко, поэтому все современные компиляторы построены на основе линейных распознавателей (иначе время их работы было бы недопустимо велико).

Для каждого класса КС-языков существует свой класс распознавателей, но все они функционируют на основе общих принципов, на которых основано моделирование работы МП-автоматов. Все распознаватели для КС-языков можно разделить на две большие группы: нисходящие и восходящие. Нисходящие распознаватели просматривают входную цепочку символов слева направо и порождают левосторонний вывод. При этом получается, что дерево вывода строится таким распознавателем от корня к листьям.

Восходящие распознаватели также просматривают входную цепочку символов слева направо, но порождают при этом правосторонний вывод. Дерево вывода строится от листьев к корню (снизу вверх).

Для моделирования работы этих двух групп распознавателей используются два алгоритма: алгоритм с подбором альтернатив — для нисходящих распознавателей, алгоритм «сдвиг-свертка» — для восходящих распознавателей. В общем случае эти два алгоритма универсальны. Они строятся на основе любой КС-грамматики после некоторых формальных преобразований и поэтому могут быть использованы для разбора цепочки любого КС-языка. В этом случае время разбора входной цепочки имеет экспоненциальную зависимость от длины цепочки. Однако для линейных распознавателей эти алгоритмы могут быть модифицированы так, чтобы время разбора имело линейную зависимость от длины входной цепочки. Указанные модификации не влияют на принципы, на основе которых построена работа алгоритмов, но позволяют оптимизировать их выполнение при условии, что грамматика входного языка принадлежит к определенному классу КС-грамматик. Для каждого такого класса предусматривается своя модификация.

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

На вопрос о том, какой распознаватель — нисходящий или восходящий — выбрать для построения синтаксического анализатора, нет однозначного ответа. Эту проблему необходимо решать, опираясь на некую дополнительную информацию о том, как будут использованы или каким образом будут обработаны результаты работы распознавателя.

Восходящий синтаксический анализ, как правило, привлекательнее нисходящего. Класс языков, заданный восходящими распознавателями, значительно шире. Однако нисходящий синтаксический анализ предпочтительнее с точки зрения процесса трансляции, поскольку на его основе легче организовать процесс порождения результирующего языка.

Преобразование КС-грамматик

Преобразование КС-грамматик преследует две основные цели: упрощение правил грамматики и облегчение создания распознавателя языка. Не всегда эти цели можно совместить. При создании компилятора для языков программирования вторая цель является основной.

Все преобразования делят на две группы:

1) преобразования, связанные с исключением из грамматики избыточных правил и символов, без которых она может существовать;
2) преобразования, в результате которых изменяется вид и состав правил, причем, грамматика может дополняться новыми правилами и новыми нетерминальными символами.

В результате преобразований всегда получается новая КС-грамматика, эквивалентная исходной, т.е. определяющая тот же язык.

Приведенной называют КС-грамматику, не содержащую недостижимые и бесплодные символы, циклы (цепные правила) и правила с пустыми цепочками. Шаги преобразования должны выполняться в указанном порядке.

1. Удаление бесплодных правил

Символ $A \in VN$ называется бесплодным тогда и только тогда, когда из него нельзя вывести ни одной цепочки терминальных символов. В простейшем случае символ бесплодный, если во всех правилах, где он стоит в левой части, он встречается и в правой. В более сложных вариантах предполагаются зависимости между цепочками бесплодных символов, которые в любой последовательности вывода порождают друг друга.

Алгоритм работает со специальным множеством нетерминальных символов. Первоначально в него попадают только символы, из которых можно непосредственно вывести терминальные цепочки, а затем оно пополняется на основе правил исходной грамматики.

Шаг 1. Y_0 = пустое множество, i=0

Шаг 2. $Y_i = \left\{A \mid (A \to \alpha) \in P, \alpha \in (Y_{i-1} \cup VT)^*\right\} \cup Y_{i-1}$

Шаг 3. если $Y_i \neq Y_{i-1}$, то i=i+1 и перейти к шагу 2, иначе перейти к шагу 4

Шаг 4. $VN' = Y_i, VT' = VT$ в P' входят те правила из P, которые содержат только символы из множества $(VT \cup Y_i), S' = S$

2. Удаление недостижимых символов

Символ называется недостижимым, если он не участвует ни в одной цепочке вывода из целевого символа грамматики.

Первоначально во множество достижимых символов входит только целевой символ, а затем оно пополняется на основе правил грамматики.

Шаг 1. $V_0 = \{S\}$, i=0

Шаг 2. $V_i = \left\{x \mid x \in (VT \cup VN) u (A \to \alpha x \beta) \in P, A \in V_{i-1}, \alpha, \beta \in (VT \cup VN)^*\right\} \cup V_{i-1}$

Шаг 3. если $V_i \neq V_{i-1}$, то i=i+1 и перейти к шагу 2, иначе перейти к шагу 4

Шаг 4. $VN' = VN \cap V_i, VT' = VT \cap V_i$ в P' входят те правила из P, которые содержат только символы из множества $V_i, S' = S$

3. Удаление правил с пустыми цепочками (λ-правил)

Правилами с пустыми цепочками называются правила вида $A \to \lambda$, где $A \in VN$.

W_i - множество нетерминальных символов.

Шаг 1. $W_0 = \{A : (A \to \lambda) \in P\}$ $i = 1$

Шаг2. $W_i = W_{i-1} \cup \{A : (A \to \alpha) \in P, \alpha \in W_{i-1}^*\}$

Шаг 3. Если $W_i \neq W_{i-1}$, то $i=i+1$, перейти к шагу 2, иначе к шагу 4.

Шаг 4. $VN' = VN$, $VT' = VT$, в P' входят все правила из *P,* кроме правил вида $A \to \lambda$.

Шаг 5. Если $(A \to \lambda) \in P$ и в цепочку α входят символы из W_i, тогда на основе α строится множество цепочек $\{\alpha'\}$ исключением всех возможных комбинаций символов W_i; все правила вида $A \to \alpha'$ добавляются в P' (при этом надо учитывать дубликаты правил и бессмысленные правила).

Шаг 6. Если $S \in W_i$, тогда в VN' добавляется новый символ S', который становится целевым символом, а в P' добавляются два новых правила $S' \to \lambda | S$; иначе $S' = S$.

4. Удаление цепных правил

Циклы возможны только в том случае, если в КС-грамматике присутствуют правила вида $A \to B$, $A, B \in VN$.

Для устранения цепных правил для каждого нетерминального символа $X \in VN$ строится специальное множество цепных символов N^X, а затем на основании построенных множеств выполняется преобразование правил *P*.

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. $N_1^X = N_{i-1}^X \cup \{B : (A \to B) \in P, B \in N_{i-1}^X\}$.

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 $B \to \alpha$.

Данный алгоритм, также как и алгоритм удаления λ-правил, ведет к увеличению числа правил грамматики, но упрощает построение распознавателей.

Семантический анализ

Для проверки семантической правильности исходной программы необходимо иметь всю информацию о найденных лексических единицах языка. Поэтому семантический анализ входной программы может быть произведен только по завершению ее синтаксического анализа.

Входными данными семантического анализа являются: таблица идентификаторов и результаты разбора синтаксических конструкций входного языка.

Семантический анализ выполняется на двух этапах компиляции: на этапе синтаксического разбора и в начале этапа подготовки к генерации кода. В первом случае всякий раз по завершении анализа определенной синтаксической конструкции входного языка выполняется ее семантическая проверка на

основе имеющихся данных в таблице идентификаторов. Во втором случае, после завершения всей фазы синтаксического анализа выполняется полный семантический анализ.

Семантический анализатор выполняет следующие действия:

- проверку соблюдения во входной программе семантических соглашений входного языка,
- дополнение внутреннего представления программы в компиляторе операторами и действиями, неявно предусмотренными семантикой входного языка,
- проверку элементарных семантических (смысловых) норм языков программирования, напрямую не связанных со входным языком.

Проверка соблюдений семантических соглашений заключается в сопоставлении входных цепочек исходной программы с требованиями семантики входного языка программирования. Примерами таких требований являются:

- каждый идентификатор должен быть описан только один раз и ни больше;
- все операнды в выражениях и операциях должны иметь типы, допустимые для данного выражения или операции;
- типы переменных в выражениях должны быть согласованы между собой;
- при вызове функций число и типы фактических параметров должны соответствовать формальным параметрам и др.

Конкретный состав таких требований жестко связан с семантикой входного языка. Если какое-либо требование не выполняется, компилятор выдает сообщение об ошибке, процесс компиляции прекращается.

Дополнение внутреннего представления программы связано с добавлением операторов и действий, неявно предусмотренных семантикой входного языка. Как правило, это операторы и действия, связанные с преобразованием типов. Другим примером такого рода операций могут служит операции вычисления адреса при обращении к элементам сложных структур данных.

Проверка элементарных смысловых норм языков программирования, напрямую не связанных со входным языком – сервисная функция, входящая в состав большинства современных компиляторов. Она обеспечивает проверку соглашений, выполнение которых связано со смыслом, как всей исходной программы, так и ее отдельных фрагментов. Примеры таких соглашений:

- каждая переменная или константа должна хоть раз использоваться в программе,
- каждая переменная должна быть определена до ее использования,
- результат функции должен быть определен при любом ходе ее выполнения,
- операторы условия и выбора должны предусматривать возможность хода выполнения программы по каждой ветви,
- операторы цикла должны иметь возможность завершения и т.д.

Конкретный перечень таких соглашений зависит от семантики входного языка. В отличии от семантических требований языка выполнение данных соглашений не является обязательным. Поэтому их несоблюдение не может трактоваться как ошибка и не приводит к прекращению процесса компиляции.

На этапе семантического анализа также выполняется идентификация переменных, типов, функций и других лексических единиц языка. Она включает в себя следующие действия:
- имена локальных переменных дополняются именами блоков, в которых они описаны,
- имена функций модулей дополняются именами модулей,
- имена функций, объявленных в классе, дополняются именами классов, которым они принадлежат,
- имена функций модифицируются в зависимости от типов их формальных аргументов.

Распределение памяти

Распределение памяти – процесс, ставящий в соответствии лексическим единицам исходной программы адрес, размер и атрибуты области памяти. Область памяти – блок ячеек памяти, выделяемый для данных, объединенных логически на основе семантики исходного языка.

Каждую область памяти можно классифицировать по двум параметрам:

- ее роли в результирующей программе (локальная или глобальная),
- способа распределения в ходе выполнения результирующей программы (статическая или динамическая). Динамическая память, в свою очередь, может распределяться либо разработчиком программы, либо автоматически компилятором.

Далеко не все лексические единицы языка требуют для себя выделения памяти.

Оптимизация кода. Основные методы оптимизации

Генерация кода в большинстве случаев выполняется компилятором не для всей исходной программы в целом, а последовательно для отдельных конструкций. Полученные для каждой синтаксической конструкции входной программы фрагменты результирующего кода также последовательно объединяются в общий текст результирующей программы, причем связи между ними в полной мере не учитываются. Построенный таким образом код может содержать лишние команды и данные. Это снижает эффективность выполнения программы.

Оптимизация программы – это обработка, связанная с переупорядочиванием операций в компилируемой программе с целью получения эффективной результирующей объектной программы. Она выполняется на этапах подготовки к генерации и непосредственно при генерации объектного кода.

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. Выбор критерия оптимизации обычно непосредственно выполняет пользователь в настройках компилятора. Но выбрав критерий оптимизации, в общем случае практически невозможно построить код результирующей программы, который являлся бы самым коротким и самым быстрым и соответствующим входной программе.

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.

Для операций, составляющих линейный участок программы, могут применяться следующие оптимизирующие преобразования.

Удаление бесполезных присваиваний, т.е. операций задающей значение переменной, которая никогда не используется. Не всегда удается установить использование значения переменной только на основании факта упоминания ее в операциях. Тогда устранение лишних присваиваний становится сложной задачей.

Исключение избыточных вычислений (лишних операций) – нахождение и удаление операций, обрабатывающих одни и те же операнды.

Свертка объектного кода – выполнение во время операций исходной программы, для которых значения операндов известны.

Перестановка операций – изменение порядка следования операций, которое может повысить эффективность программы, но не будет влиять на конечный результат вычислений.

Арифметические преобразования – выполнение изменения характера и порядка следования операций на основании известных алгебраических и логических тождеств. Например, выражение A=B*C+B*D может быть заменено на A+D(C+D), при этом результат не изменится, но объектный код будет содержать на одну операцию меньше.

Особенность оптимизации логических выражений заключается в том, что для того, чтобы узнать результат выражения не всегда требуется вычислять его полностью. Операция называется предопределенной для некоторого значения операнда, если ее результат зависит только от этого операнда и остается неизменным (инвариантным) относительно значений других операндов. Предопределенный результат также могут иметь некоторые математические операции и функции. Например, нет смысла выполнять умножение на нуль.

Для оптимизации циклов используются методы:

- вынесения инвариантных вычислений из циклов – вынесение операций, операнды которых не изменяются в цикле,
- замена операций с индуктивными переменными – изменение сложных операций на более простые. Переменная называется индуктивной в цикле, если ее значения во время выполнения цикла образуют арифметическую прогрессию. Таких переменных в цикле может быть несколько и их можно заменить одной, а реальные значения будут вычисляться с помощью коэффициентов соотношения,
- слияние и развертывание циклов.

VI. ФОНД ТЕСТОВЫХ ЗАДАНИЙ ДЛЯ ОЦЕНКИ КАЧЕСТВА ЗНАНИЙ

1. Какие из следующих утверждений справедливы?

А) если язык задан КС-грамматикой, то он может быть задан с помощью МП-автомата;
Б) если язык задан КС-грамматикой, то он может быть задан с помощью ДМП-автомата;
В) если язык задан ДМП-автоматом, то он может быть задан КС-грамматикой;
Г) если язык задан расширенным МП-автоматом, то он может быть задан КС-грамматикой;

2. Грамматика вида G(VT, VN, P, S), P: $A \to \beta$, $A \in VN, \beta \in V^+$ называется

А) контекстно-зависимой;
Б) регулярной;
В) контекстно-свободной;
Г) с фразовой структурой.

3. Программа, переводящая программу на исходном языке в эквивалентную ей программу на результирующем языке, называется

А) компилятором
Б) транслятором
В) интерпретатором

4. Методы бинарного дерева, логарифмического поиска, хэш-функций относятся к
А) построению таблиц идентификаторов
Б) лексическому анализу конструкций входного языка
В) синтаксическому анализу конструкций входного языка
Г) построению таблиц лексем

5. Для взаимосвязи синтаксического и лексического анализаторов используют
А) только последовательный подход
Б) только параллельный подход
В) последовательный и параллельный подходы

6. Распознаватель на основе автомата с магазинной памятью распознает языки на основе:
А) контекстно-зависимой грамматики;
Б) регулярной грамматики;
В) контекстно-свободной грамматики;
Г) грамматики с фразовой структурой.

7. Выберите верный порядок выполнения преобразований, приводящих КС-грамматики к приведенному виду:
А) удаление недостижимых символов, удаление бесплодных символов, удаление цепных правил, удаление правил с пустыми цепочками,
Б) удаление недостижимых символов, удаление цепных правил, удаление бесплодных символов, удаление правил с пустыми цепочками.
В) удаление правил с пустыми цепочками, удаление недостижимых символов, удаление бесплодных символов, удаление цепных правил.
Г) удаление бесплодных символов, удаление недостижимых символов, удаление правил с пустыми цепочками, удаление цепных правил.

8. Самым примитивным типом распознавателя КС-языков является
А) распознаватель с возвратом
Б) распознаватель на основе алгоритма «сдвиг-свертка»
В) нисходящий распознаватель без возвратов
Г) распознаватель на основе метода рекурсивного спуска

9. По способам распределения памяти она делится на
А) статическую и динамическую
Б) глобальную и локальную
В) распределяемую разработчиком и компилятором
Г) внешнюю и внутреннюю

10. Выберите справедливые утверждения:
А) любой язык программирования является контекстно-свободным

Б) синтаксис любого языка программирования может быть описан контекстно-свободной грамматикой
В) любой язык программирования является контекстно-зависимым языком
Д) семантика любого языка программирования может быть описана контекстно-свободной грамматикой

VII. КОМПЛЕКТ ЭКЗАМЕНАЦИОННЫХ БИЛЕТОВ

Билет №1
1.Реализация алгоритма работы нисходящего распознавателя
2. Виды переменных

Билет№2
1. Алгоритм удаления бесплодных символов КС-грамматики
2. Определение границ лексем

Билет№3
1. Алгоритм удаления недостижимых символов КС-грамматики
2. Автоматы с магазинной памятью

Билет№4
1. Удаление правил с пустыми цепочками
2. Назначение лексического анализатора

Билет№5
1. Алгоритм удаления циклических правил
2. Многопроходные и однопроходные компиляторы

Билет№6
1. Принципы работы транслятора, компилятора, интерпретатора
2. Регулярные и автоматные грамматики Преобразование регулярной грамматики к автоматному виду

Билет №7
1. Организация таблиц идентификаторов
2. Нисходящий распознаватель с подбором альтернатив

Билет №8
1. Метод логарифмического поиска построения таблиц идентификаторов
2. Удаление правил с пустыми цепочками

Билет №9
1. Построение таблиц идентификаторов по методу бинарного дерева
2. Удаление правил с пустыми цепочками

Билет №10

1. Этапы трансляции. Общая схема работы транслятора
2. Алгоритм удаления бесплодных символов КС-грамматики

Билет № 11

1. Построение таблиц идентификаторов с помощью хэш-функции
2. Понятие грамматики. Контекстно-свободные грамматики. Приведенные грамматики КС-грамматики

Билет №12

1. Управление файлами
2. Алгоритм удаления циклических правил

Билет №13

1. Формальное определение грамматики. Форма Бэкуса-Наура
2 Средства трассировки и отладки программ

Билет №14

1. Синтаксический распознаватель с возвратом
2. Управление задачами

Билет №15

1. Построение идентификаторов по методу цепочек
2. Понятие грамматики. Леволинейные и праволинейные грамматики

Билет №16

1. Управление памятью
2. Формальное определение языка

Билет №17

1. Различные виды записи грамматик
2. Операционная система Windows NT

Билет №18

1. Мобильность программного обеспечения
2. Четыре типа грамматик по Хомскому

Билет №19

1. Цепочки символов. Операции над цепочками символов
2. Операционная система Linux

Билет №20

1. Общая схема распознавателя

2. Ассемблеры

Билет №21

1. Современные операционные системы
2. Синхронизация процессов

Билет № 22

1. Классификация операционных систем
2. Принципы функционирования систем программирования

Билет № 23

1. Пользовательский интерфейс операционной системы
2. Удаление правил с пустыми цепочками

Билет №24

1. Управление вводом-выводом
2. Синтаксический распознаватель с возвратом

Билет № 25

1. Макроязыки
2. Интерактивные системы

VIII. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПРОФЕССОРСКО-ПРЕПОДАВАТЕЛЬСКОМУ СОСТАВУ ПО ОРГАНИЗАЦИИ МЕЖСЕССИОННОГО КОНТРОЛЯ ЗНАНИЙ

Межсессионный контроль осуществляется на лабораторных занятий. По итогам сдачи лабораторных работ и тестовых заданий по лекционным материалам и самостоятельно изученным теоретическим вопросам в сроки, установленные деканатом (как правило, на 6-ой и 12-ой неделе семестра) преподавателем выставляется аттестационная оценка.

IX. КАРТА ОБЕСПЕЧЕННОСТИ КАДРАМИ ПРОФЕССОРСКО-ПРЕПОДАВАТЕЛЬСКОГО СОСТАВА

Все виды занятий по данной дисциплине ведет канд. техн. наук, доцент Галаган Т.А.

СОДЕРЖАНИЕ

Татьяна Алексеевна Галаган,
доцент кафедры ИиУС АмГУ

Учебно-методический комплекс по дисциплине «Системное программное обеспечение»

Изд-во АмГУ. Подписано к печати Формат 60x84/16. Усл. печ. л.
Тираж Заказ