Министерство образования и науки РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
(ФГБОУ ВО «АмГУ»)

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

сборник учебно-методических материалов

для специальности 09.02.07 Информационные системы и программирование

Благовещенск

2020

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

Составитель: Галаган Т.А.

Системное программирование: сборник учебно-методических материалов для специальности 09.02.07 – Благовещенск: Амурский гос. ун-т, 2020

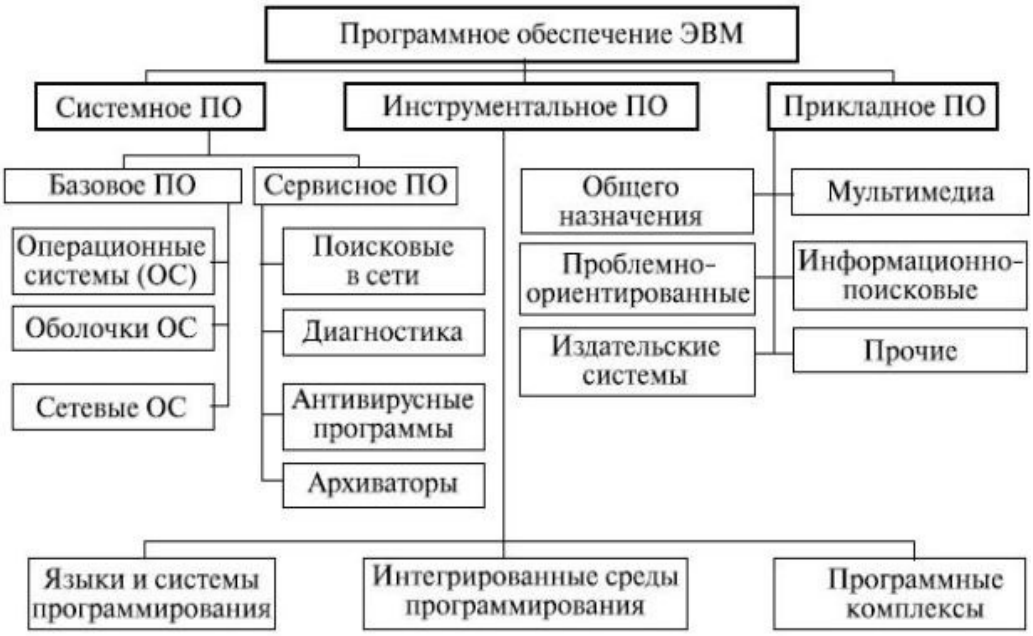
КРАТКОЕ ИЗЛОЖЕНИЕ ЛЕКЦИОННОГО МАТЕРИАЛА

Определение и виды программного обеспечения

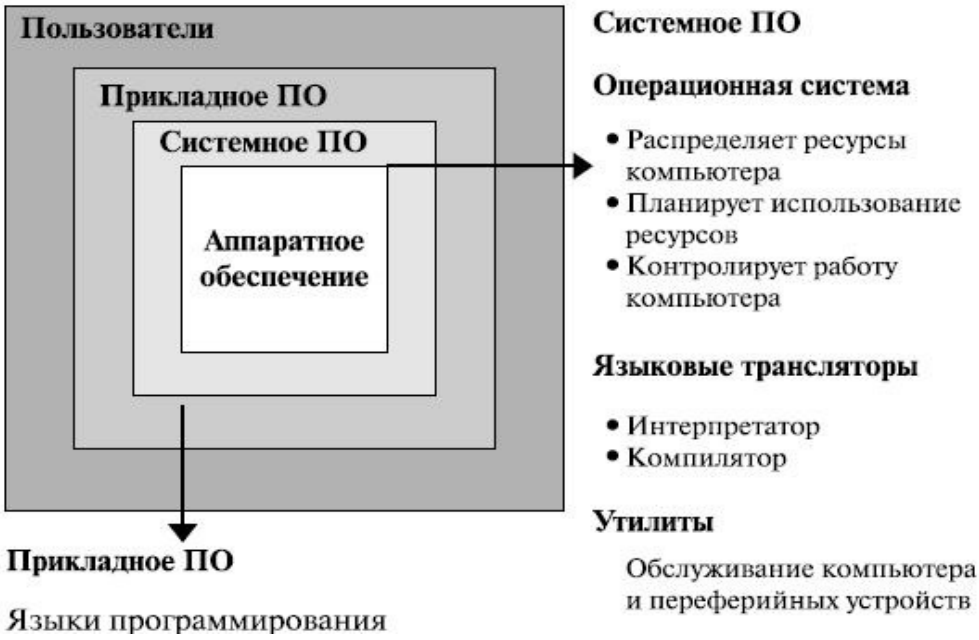
Под **программным обеспечением** (ПО) понимается совокупность программ, выполняемых вычислительной системой.

Системное программное обеспечение – это набор программ, которые управляют компонентами компьютера, такими как процессор, коммуникационные и периферийные устройства. Программистов, которые создают системное *программное обеспечение*, называют системными программистами.

К ***прикладному программному обеспечению*** относятся программы, написанные для пользователей или самими пользователями, для задания компьютеру конкретной работы.



Взаимосвязь системного и прикладного программного обеспечения



ПРИНЦИПЫ РАБОТЫ ТРАНСЛЯТОРА, КОМПИЛЯТОРА, ИНТЕРПРЕТАТОРА

Входными данными для работы транслятора служит программа на исходном языке программирования, которая называется *исходной программой.* Обычно это символьный файл, содержащий текст, удовлетворяющий синтаксическим и семантическим требованиям входного языка. Кроме того, этот файл несет в себе некоторый смысл, определяемый семантикой входного языка.

Транслятор является программой, переводящей исходную программу в эквивалентную ей программу на *результирующем (выходном) языке*. Р*езультирующая программа* строится по синтаксическим правилам выходного языка транслятора, а ее смысл определяется семантикой выходного языка. (Теоретически возможна реализация транслятора с помощью аппаратных средств, однако широкое практическое применение их не известно.) 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.

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. Часто они совпадают, но бывает так, что компилятор работает под управлением вычислительной системы одного типа, а строит объектные программы, предназначенные для выполнения на вычислительных системах совсем другого типа.

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 Требование об эквивалентности исходной программы и порожденных машинных кодов и в этом случае должно обязательно выполняться.

ИНТЕРФЕЙСЫ И ОСНОВНЫЕ СТАНДАРТЫ ОПЕРАЦИОННЫХ СИСТЕМ

Под интерфейсами ОС понимают специальные интерфейсы системного и прикладного программирования (API).

Термин API делят на следующие направления:

- API как интерфейс высокого уровня, принадлежащий к библиотекам RTL;
- API прикладных и системных программ, входящий в состав в ОС;
- прочие интерфейсы API.

Программный интерфейс API включает не только сами функции, но и соглашения об их использовании, которые регламентируются самой ОС, архитектурой целевой вычислительной системы и системой программирования.

Существует несколько вариантов реализации функций API:

- на уровне модулей ОС;
- на уровне систем программирования;
- на уровне внешней библиотеки процедур и функций.

Возможности API оценивают с позиций эффективности выполнения функций API, широты предоставляемых возможностей, зависимости прикладной программы от архитектуры целевой вычислительной системы.

Как правило, функции API не стандартизированы. В каждом конкретном случае набор вызовов API определяется архитектурой ОС. Принимаются попытки стандартизировать некоторый набор функций.

В последние годы большую популярность получили графические интерфейсы GUI (Graphical User Interface), которые являются частным случаем задачи управления вводом-выводом и не относятся к функциям ядра ОС, хотя в ряде случаев разработчики ОС относят GUI к основному системному интерфейсу API.

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

Примером стандартизации API служит один из самых распространенных стандартов – POSIX.

POSIX – независимый от платформы системный интерфейс для компьютерного окружения – является стандартом IEEE. Он описывает системные интерфейсы для открытых ОС, в том числе оболочки, утилиты, инструментарии, задачи обеспечения безопасности, сетевые функции и обработку транзакций. Стандарт базируется на UNIX-системах, но допускает реализацию и в других ОС.

Этот стандарт подробно описывает систему виртуальной памяти, многозадачность и технологию переноса ОС. Он представляет собой множество стандартов POSIX.1 – POSIX.12.

Процессы и потоки

Важнейшей функцией ОС является организация рационального использования всех аппаратных и информационных ресурсов. Задачи управления ресурсами более сложны в мультипрограммных ОС, в которых за ресурсы конкурируют сразу несколько приложений. При этом программы попеременно выполняются на одном процессоре и совместно

используют такие ресурсы компьютера, как оперативная и внешняя память, устройства ввода-вывода, данные.

Одной из основных подсистем мультипрограммной ОС, непосредственно влияющей на функционирование вычислительной машины, является подсистема управления процессами и потоками, которая занимается их созданием и уничтожением, поддерживает взаимодействие между ними, а также распределяет процессорное время между несколько одновременно существующими в системе потоками и процессами.

Чтобы поддерживать мультипрограммирование, ОС должна определить и оформить для себя те внутренние единицы работы, между которыми будет разделяться процессор и другие ресурсы компьютера. В настоящее время в большинстве ОС определены два типа единиц работы. Более крупная единица работы, носящая название *процесса* (или *задачи*) требует для своего выполнения нескольких более мелких работ, для обозначения которых используют термины *«поток»* (*«нить»*).

Потоки возникли в ОС как средство распараллеливания вычислений, облегчающее работу программиста. В ОС, не поддерживающей потоков, процесс всегда состоит из одного потока, а программисту приходится решать задачу синхронизации нескольких параллельных ветвей программы.

Процесс рассматривается ОС как заявка на потребление всех видов ресурсов, кроме одного – процессорного времени. Процессорное время распределяется ОС между – *потоками,* представляющими собой последовательности команд. Создать процесс – это, прежде всего, 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 и контекст. *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 или выгружен на диск.

Дескрипторы отдельных процессов объединены в список, образующий таблицу процессов. Память для таблицы процессов отводится динамически в области ядра. На основании хранящейся в ней информации ОС осуществляет планирование и синхронизацию процессов. В дескрипторе прямо или косвенно (через указатели на структуры процесса) содержится информация о состоянии процесса, о расположении образа процесса в оперативной памяти и на диске, о значении отдельных составляющих приоритета, а также о его итоговом значении – глобальном приоритете, об идентификаторах пользователя, создавшего процесс, о родственных процессах и другая информация.

Контекст процесса содержит менее оперативную, но более объемную часть информации, необходимую для возобновления выполнения процесса с прерванного места. Он хранит содержимое регистров процессора, коды ошибок выполняемых процессором системных вызовов, информация обо всех файлах, открытых процессом, информация о незавершенных операциях ввода-вывода и др.

Контекст процесса также доступен только программам ядра, однако он хранится не в области ядра, а непосредственно примыкает к образу процесса и перемещается вместе с ним при необходимости из оперативной памяти на диск.

Для реализации мультипрограммирования на протяжении существования процесса выполнение его потоков может быть многократно прервано и продолжено. Переход от выполнения одного потока к другому осуществляется в результате планирования и диспетчеризации.

Планирование включает определение момента времени для смены текущего потока, а также выбор нового потока для выполнения. *Диспетчеризация* заключается в реализации найденного в результате планирования решения, т.е. в переключении процесса с од-

ного потока на другой.

Диспетчеризация включает в себя: сохранение контекста текущего потока; загрузку контекстов нового потока, выбранного в результате планирования; запуск нового потока на выполнение.

В системе, не содержащей потоки, все сказанное в дальнейшем о планировании и диспетчеризации потоков может быть отнесено к процессам в целом.

Особенности реализации планирования потоков в наибольшей степени определяют специфику операционной системы, в частности, является ли она системой пакетной обработки, системой разделения времени или системой реального времени.

В большинстве ОС планирование осуществляется динамически (on-line), то есть решения принимаются во время работы системы на основе анализа текущей ситуации. ОС работает в условиях неопределенности – потоки и процессы появляются в случайные моменты времени и также непредсказуемо завершаются. Динамические планировщики могут гибко приспосабливаться к изменяющейся ситуации и не используют никаких предположений относительно мультипрограммной смеси. ОС при этом затрачивает значительные усилия.

Статическое планирование может использоваться в специализированных системах, в которых весь набор одновременно выполняемых задач определен заранее, например, в системах реального времени. Результатом работы статического планировщика является таблица, называемая расписанием, в которой указывается, какому потоку/процессу, когда и на какое время должен быть предоставлен процессор. Для построения расписания планировщику нужны как можно более полные предварительные сведения о характере набора задач.

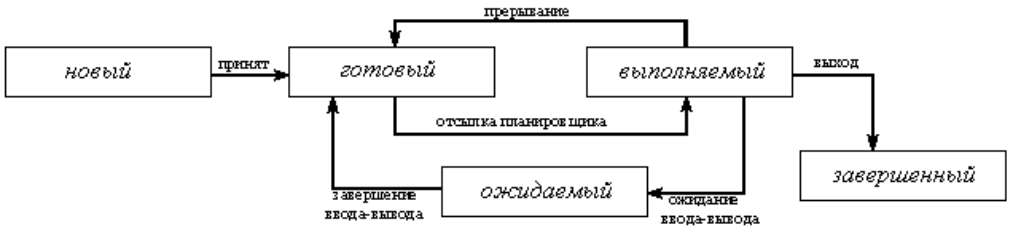
ОС выполняет планирование потоков, принимая во внимание их состояние. В мультипрограммной системе различают три основных состояния потока:

- выполнение – активное состояние потока, во время которого поток обладает всеми необходимыми ресурсами и непосредственно выполняется процессом;
- ожидание – пассивное состояние потока, находясь в котором, поток заблокирован по своим внутренним причинам (ждет осуществления некоторого события, чаще всего завершения ввода-вывода, или освобождения некоторого ресурса);
- готовность – пассивное состояние процесса, но при этом поток заблокирован в связи с внешними обстоятельствами.

Иногда еще добавляют состояние новый, т.е. только созданный поток, и завершенный.

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

На рисунке приведена типовая диаграмма переходов для состояний процессора.



Под мультипрограммирование понимается способ организации вычислений, когда на одном процессоре поочередно выполняется несколько задач, создавая видимость их одновременного выполнения.

Однопрограммные ОС в основном выполняют функцию предоставления пользователю виртуальной машины и включают средства управления периферийными устройствами, средства управления файлами, средства общения с пользователем.

Мультипрограммные ОС, кроме выполнения вышеперечисленных функций, управляют разделением совместно используемых ресурсов, таких как процессор, оперативная память, файлы и внешние устройства.

Близким понятием является мультизадачность. Мультизадачный режим предполагает, что забота о параллельном выполнении приложений ложится на программистов.

Современные ОС реализуют и мультипрограммный и мультизадачный режимы.

Выделяют также системы многопроцессорной обработки. Мультипроцессирование приводит к усложнению всех алгоритмов управления ресурсами.

По способу распределения процессорного времени между несколькими одновременно существующими в системе процессами выделяют две группы алгоритмов:

невытесняющая многозадачность – процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление операционной системе для того, чтобы та выбрала из очереди другой готовый к выполнению процесс;

вытесняющая многозадачность – решение о переключении одного процесса на другой принимается операционной системой, а не самим активным процессом.

По способу взаимодействия с компьютером ОС делят на диалоговые системы и системы пакетной обработки.

Диалоговые системы делят по числу одновременно работающих пользователей на однопользовательские (MS DOS, Windows 3.x) и многопользовательские (UNIX, Windows NT).

Отдельным классом являются ОС реального времени, в которых обработка поступающих заданий обеспечивается в заданном интервале времени, который нельзя превышать.

Операционная система компьютерной сети аналогична ОС автономного компьютера. При организации сетевой работы она играет роль интерфейса экранирующего от пользователя все детали низкоуровневых программно-аппаратных средств сети.

В зависимости от того, какой виртуальный образ создает СОС, различают сетевые ОС и распределенные ОС.

СИНХРОНИЗАЦИЯ ПРОЦЕССОВ И ПОТОКОВ

В мультипрограммных ОС потребность в синхронизации процессов и потоков связана с совместным использованием аппаратных и информационных ресурсов. Средства ОС, обеспечивающие эту синхронизацию, часто называются средствами межпроцессного взаимодействия. – IPC (Inter Process Communication).

Выполнение потоков в мультипрограммной среде всегда происходят асинхронно, независимо друг от друга. Моменты прерывания потоков, время нахождения их в очередях, порядок выбора потоков для выполнения – случайные события, зависящие от стечения многих обстоятельств. В лучшем случае можно лишь оценить их вероятностные характеристики, например вероятность их завершения за данный период времени.

Любое взаимодействие потоков, связанное с разделением ресурсов, обменом данными, совместным использованием аппаратных ресурсов, обеспечивается их синхронизацией. Она заключается в согласование скоростей путем приостановки потока до наступления некоторого события и последующей его активизации.

Для синхронизации потоков прикладных программист может использовать либо собственные средства и приемы синхронизации, либо средства операционной системы.

Важным понятием синхронизации потоков является понятие «*критической секции*» программы. Это часть программы, результат выполнения которой может непредсказуемо меняться, если переменные, относящиеся к этой части программы, изменяются другими потоками в то время, когда выполнение этой части еще не завершено. Критическая секция

всегда определяется по отношению к определенным *критическим данным*, при несогласованном изменении которых могут возникнуть нежелательные эффекты.

Ситуации, в которых несколько потоков обрабатывают разделяемые данные и конечный результат зависит от соотношения скоростей потоков, называются *гонками*.

Чтобы исключить эффект гонок по отношению к критическим данным, необходимо обеспечить, чтобы в каждый момент времени в критической секции, связанной с этими данными, находился только один поток. При этом неважно, в активном состоянии он находится или в пассивном. Этот прием называют *взаимным исключением*. Операционная система использует разные способы реализации взаимного исключения. Некоторые пригодны только для потоков одного процесса, другие – для потоков разных процессов.

Самый простой и неэффективный способ состоит в запрещении любых прерываний на время нахождения потока в критической секции. Этот способ практически не применяется, так как опасно доверять управление системой пользовательскому потоку.

Для синхронизации потоков одного процесса прикладной программист может использовать глобальные *блокирующие переменные*. Каждому набору критических данных ставится в соответствие двоичная переменная, которой поток присваивает значение равное нулю, когда он входит в критическую секцию, и значение единица, если он ее покидает. Перед входом в критическую секцию поток проверяет, не работает ли другой поток с этими же данными. Если значение переменной равно нулю, то данные заняты, и проверка циклически повторяется.

Блокирующие переменные могут использоваться не только при доступе к разделяемым данным, но и при доступе к разделяемым ресурсам любого вида.

При использовании блокирующих переменных потоки могут прерываться в любой момент и в любом месте, в том числе и в критической секции. Единственное ограничение – нельзя прервать поток между операциями проверки и установки значения блокирующей переменной. Для этого в системе команд многих компьютеров предусмотрена единая неделимая команда анализа и присвоения значения логической переменной (например, BTC, BTR и BTC процессора Pentium). При отсутствии такой команды аналогичные действия должны реализовываться специальными системными примитивами, запрещающими прерывания на протяжении операций проверки и установки.

Недостатком изложенного способа взаимного исключения является то, что поток, требующий ресурс, который уже занят другим ресурсом, будет непрерывно опрашивать блокирующую переменную, бесполезно тратя выделяемое ему процессорное время.

Для устранения этого недостатка во многих ОС предусматриваются специальные системные вызовы для работы с критическими секциями. В Windows NT перед тем как начать изменения в критических данных, поток выполняет системный вызов EnterCriticalSection( ). В рамках этого вызова также выполняется проверка блокирующей переменной, но в отличие от предыдущего способа, не выполняется циклический опрос, а поток переводится в состояние ожидания и делается пометка о том, что он должен быть активизирован, когда требуемый ресурс освободится. Поток, используемый в это время ресурс, после выхода из критической области выполняет системную функцию LeaveCriticalSection( ), в результате которой блокирующая переменная принимает значение единица, а операционная система просматривает очередь ожидающих этот ресурс потоков и переводит первый из них в состояние готовности. Однако в случаях, когда объем работы в критической секции небольшой более предпочтительным является использование блокирующих переменных.

Обобщением блокирующих переменных являются так называемые *семафоры Дийкстры*. Дийкстра предложил вместо двоичных переменных использовать переменные, хранящие целые неотрицательные значения. Эти переменные и называются семафорами.

Для работы с семафорами вводятся два примитива (базовые функции ОС), традиционно обозначаемые V и P. Семафор обозначают S. Тогда определяются следующие действия:

V(S) – переменная S увеличивается на единицу единым действием. К переменной S нет доступа другим потокам во время этой операции. Выборка, наращивание и запоминание этой переменной не могут быть прерваны.

P(S) – уменьшение S на единицу, если это возможно. Если S=0, то поток, вызывающий операцию P, ждет, пока уменьшение P не станет возможным. Успешная проверка и уменьшение также являются неделимым операциями.

Никакие прерывания во время примитивов V и P невозможны.

Если семафор принимает всего два значения 0 и 1, он превращается в блокирующую переменную. Такие семафоры называются *мьютексами*. Для работы с ними вводятся специальные системные вызовы CreateMutex( ), OpenMutex( ), ReleaseMutex( ) и др. Конкретные обращения к этим функциям и перечни передаваемых и получаемых параметров содержатся в документации на соответствующую операционную систему.

При параллельном выполнении двух процессов могут возникать также тупиковые ситуации, когда два или более процессов блокируют друг друга, вынуждая ожидать события, связанного с освобождением ресурса. В мультипрограммной системе процесс находится в состоянии тупика (дедлока – dead lock (смертельное объятие), клинча), если он ждет события, которое никогда не произойдет. Тупики чаще возникают из-за конкуренции несвязанных параллельных процессов за ресурсы, реже из-за ошибок программирования.

Пример тупика. Два потока, принадлежащие разным процессам, требуют для выполнения два ресурса – принтер и последовательный порт. В зависимости от соотношения скоростей их выполнения потоки могут либо взаимно блокировать друг друга, либо организовывать очереди к разделяемым ресурсам, либо независимо их использовать. Если поток А запрашиает сначала принтер, а затем порт, а поток В делает это в обратном порядке. После того как ОС назначила принтер потоку А и установила блокирующую переменную, связанную с ним, поток А был прерван. Поток В уже получивший COM-порт, оказался заблокирован по причине занятости принтера. Тогда в этом положении потоки могут находиться сколь угодно долго.

В случае возникновения тупика ОС должна предоставлять администратору средства распознавания тупика, и после его диагностики средства снятия взаимных блокировок и восстановления нормального функционирования системы. Существуют формальные, программно-реализованные методы распознавания тупиков, основанные на ведении таблиц распределения ресурсов и таблиц запросов к занятым ресурсам.

УПРАВЛЕНИЕ ПАМЯТЬЮ

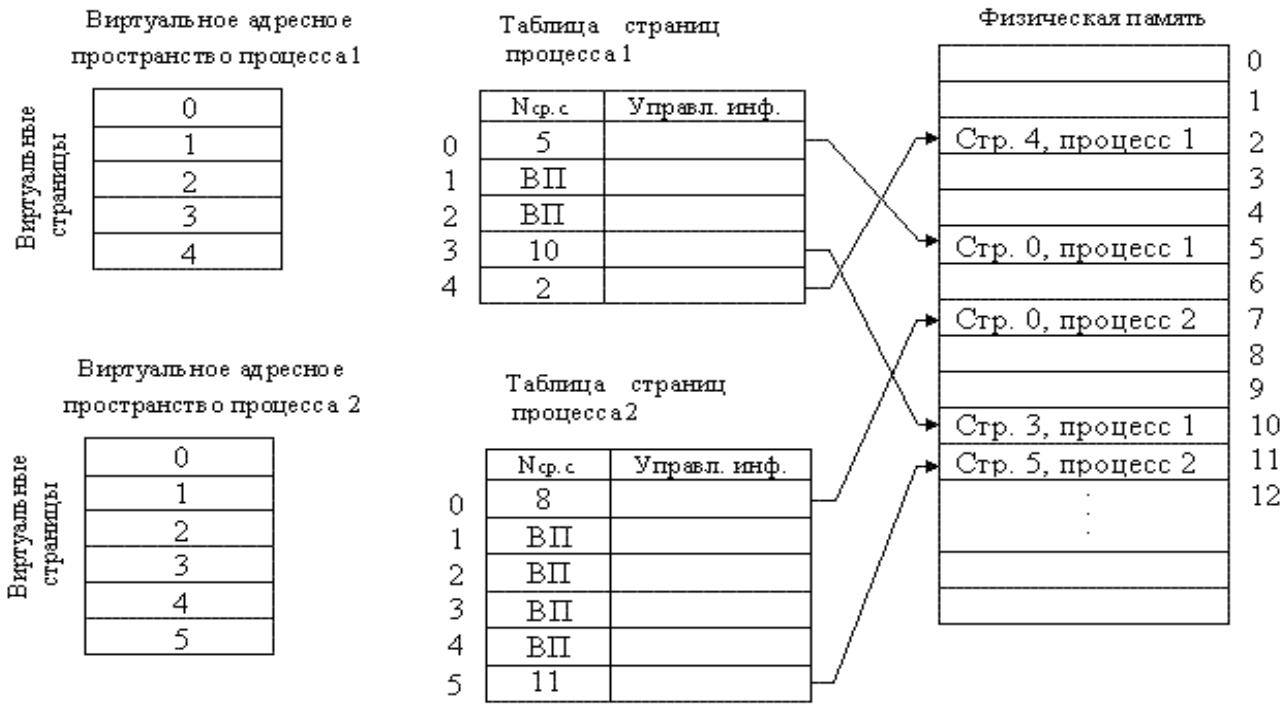
Страничное распределение памяти

Виртуальное адресное пространство каждого процесса делится на части одинакового фиксированного для данной системы размера, называемые *виртуальными страницами*. В общем случае размер виртуального адресного пространства не кратен размеру страницы, поэтому последняя страница каждого процесса дополняется фиктивной областью.

Вся оперативная память машины также делится на части того же размера, называемые *физическими страницами.* Размер страницы выбирается равным степени двойки: 512, 1024, 4096 байт и т. д. Это позволяет упростить механизм преобразования адресов.

При создании процесса ОС загружает в оперативную память несколько его виртуальных страниц. Копия всего виртуального адресного пространства находится на диске. Смежные виртуальные страницы необязательно находятся на смежных физиче-

ских сегментах. Для каждого процесса ОС создает *таблицу страниц* – информационную структуру, содержащую записи обо всех виртуальных страницах процесса.



Дескриптор страницы и содержит следующую информацию:

- номер физической страницы, в которую загружена данная виртуальная страница;
- признак присутствия, устанавливаемый в единицу, если виртуальная страница находится в оперативной памяти;
- признак модификации страницы, который устанавливается в единицу всякий раз, когда производится запись по адресу, относящемуся к данной странице;
- признак обращения к странице, называемой также битом доступа, который также устанавливается в единицу при каждом обращении по адресу, относящемуся к данной странице.

Признаки присутствия, модификации и обращения в большинстве моделей современных процессоров устанавливаются аппаратно, схемами процессора при выполнении операции с памятью.

Сами таблицы страниц, также как и описываемые ими страницы хранятся в оперативной памяти. Адрес таблицы включается в контекст соответствующего процесса.

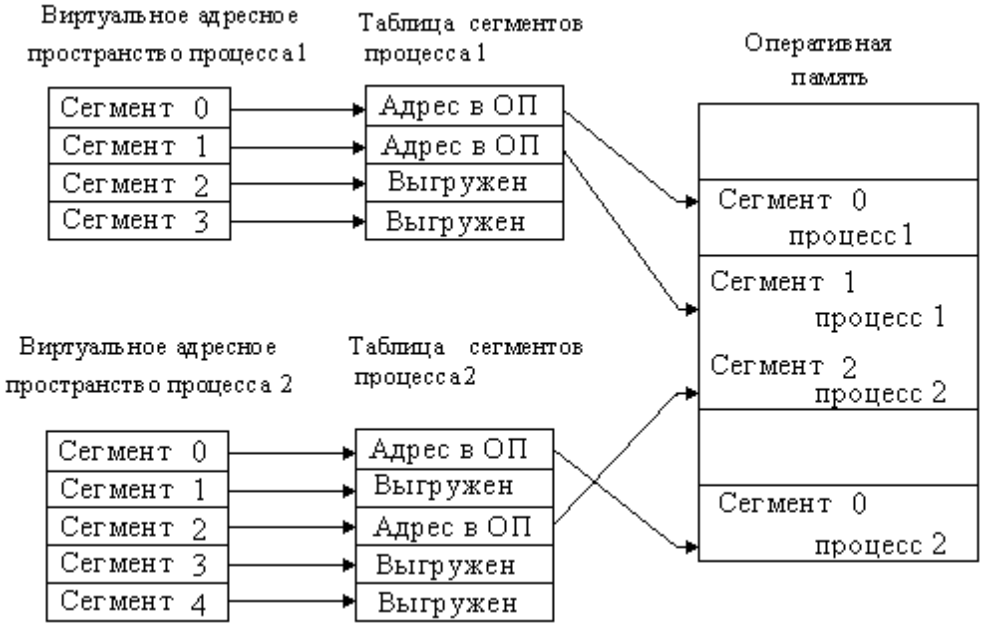
При каждом обращении к памяти выполняется поиск номера виртуальной страницы, содержащей требуемый адрес, затем по этому номеру определяется нужный элемент таблицы страниц, и из него извлекается описывающая страницу информация. Далее анализируется признак присутствия, и, если данная виртуальная страница находится в ОП, выполняется преобразование виртуального адреса - в физический. Если же нужная виртуальная страница в данный момент выгружена на диск, то происходит так называемое *страничное прерывание*. Выполняющийся процесс переводится в состояние ожидание, и активизируется другой процесс из очереди процессов, находящихся в состоянии готовности. Параллельно программа обработки страничного прерывания находит на диске требуемую виртуальную страницу и пытается загрузить ее в ОП. Если в памяти имеется свободная физическая страница, то загрузка выполняется немедленно, если же свободных страниц нет, то на основании принятой в данной системе стратегии замещения страниц решается вопрос о том, какую страницу следует выгрузить из оперативной памяти.

После выбора выгружаемой страницы обнуляется ее бит присутствия и анализируется ее бит модификации. Если выталкиваемая страница была модифицирована, то ее новая версия переписывается на диск. Если нет, запись на диск не производится, т.к. предполагается, что на диске уже есть данная копия. Физическая станица объявляется свободной. Из соображений безопасности в некоторых системах освобождаемая страница обнуляется, с тем, чтобы невозможно было использовать содержимое выгруженной страницы.

Сегментное распределение памяти

При страничной организации виртуальное адресное пространство процесса делится на равные части механически, без учета смыслового значения данных, что не позволяет обеспечивать дифференцированный доступ к разным частям программы. Деление на «осмысленные» части делает возможным совместное использование фрагментов программ разными процессами. При отображении в физическую память сегменты, содержащие коды подпрограмм из разных виртуальных пространств, могут проецироваться на одну область памяти.

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



При загрузке процесса в ОП помещается только часть его сегментов, полная копия виртуального адресного пространства находится в дисковой памяти. Для каждого загружаемого сегмента ОС подыскивает непрерывный участок свободной памяти подходящего размера. Смежные в виртуальной памяти сегменты одного процесса могут занимать в ОП несмежные участки. Если во время выполнения процесса происходит обращение по виртуальному адресу, содержащемуся в сегменте, отсутствующему в памяти, происходит прерывание. ОС приостанавливает процесс, запускает на выполнение следующий процесс из очереди и параллельно организует загрузку нужного сегмента с диска. При отсутствии места ОС выбирает сегмент на выгрузку, используя критерии аналогичные критериям выбора страниц при страничном распределении.

Во время загрузки образа процесса в ОП система создает *таблицу сегментов* процесса, в которой указывается:

- базовый физический адрес сегмента в ОП,
- размер сегмента,
- правила доступа к сегменту,
- признаки модификации, присутствия и обращения к данному сегменту,
- другая информация.

Если виртуальные адресные пространства нескольких процессов включают один и тот же сегмент, то в таблицах сегментов этих процессов делаются ссылки на один и тот же участок ОП.

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

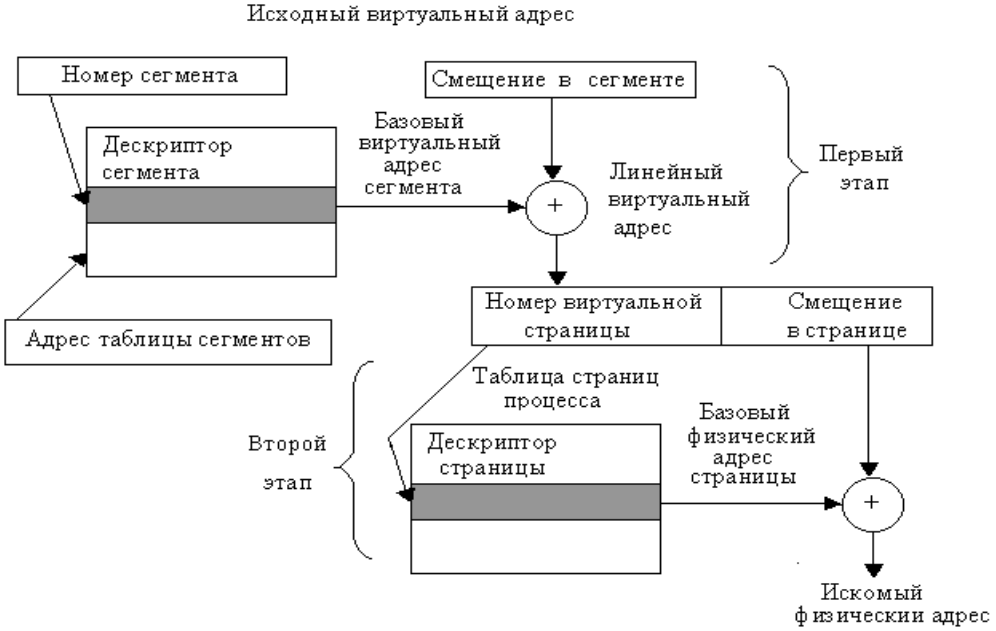
Сегментно-страничное распределение

Данный метод представляет собой комбинацию страничного и сегментного механизмов управления памятью и направлен на реализацию достоинств обоих подходов.

Виртуальное адресное пространство процесса разделено на сегменты так же как при сегментной организации памяти, что позволяет определять разные права доступа к разным частям кодов и данных программы.

Перемещение же данных между памятью и диском осуществляется не сегментами, а страницами. Для этого каждый виртуальный сегмент и физическая память делятся на страницы равного размера, что позволяет более эффективно использовать память, сократив до минимума фрагментацию

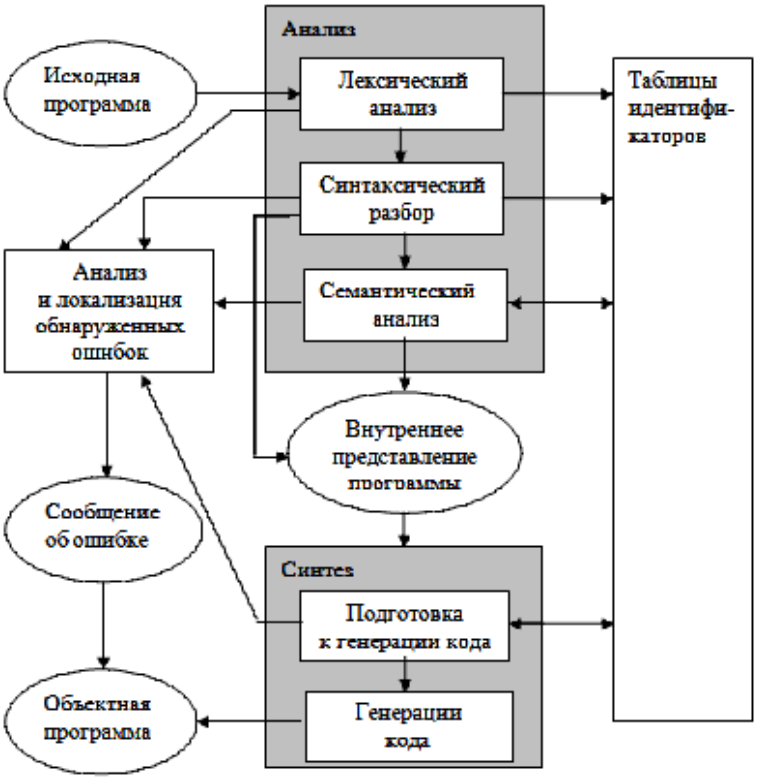
Преобразование виртуального адреса в физический при сегментно-страничной организации памяти



ОБЩАЯ СХЕМА РАБОТЫ КОМПИЛЯТОРА

На рисунке 2 представлена общая схема работы компилятора. Из нее видно, что в

целом процесс компиляции состоит из двух основных этапов – анализа и синтеза.



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.

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

циях 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.

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.

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.

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.

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. *Проходом* называют 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 ком-

пиляторы стремятся максимально использовать для хранения данных оперативную память компьютера, и только при недостатке объема доступной памяти используются накопители на жестких дисках. Другие носители информации в современных компиляторах используются из-за невысокой скорости обмена данными.

Разработчики стремятся максимально сократить количество проходов, выполняемых компиляторами. 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.

ЯЗЫКИ И ГРАММАТИКИ

Языки программирования занимают некоторое промежуточное положение между формальными и естественными языками. С формальными языками их объединяют строгие синтаксические правила, на основе которых строятся предложения языка.

Грамматика – описание способа построения предложений некоторого языка. Грамматика – математическая система, определяющая язык, т.е. – генератор цепочек языка.

Грамматику языка можно описать различными способами. Для синтаксических конструкций языков программирования можно использовать формальное описание грамматики, построенное на основе системы правил (или продукций).

Цепочкой символов называют произвольную упорядоченную конечную последовательность символов, записанных друг за другом. Количество символов в ней называют длиной цепочки.

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

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 грамматики, вида $\alpha \to \beta$, где $\alpha \in (\mathbf{VN} \cup \mathbf{VT})^+$, $\beta \in (\mathbf{VN} \cup \mathbf{VT})^*$;

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

Обозначение вида $\mathbf{V}^+$ означает множество без пустой цепочки, а обозначение $\mathbf{V}^*$ – множество, включающее пустую цепочку.

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

Целевой символ грамматики – это всегда нетерминальный символ.

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: $\alpha \rightarrow \beta_1$, $\alpha \rightarrow \beta_2$, ... $\alpha \rightarrow \beta_n$. Тогда эти правила объединяют вместе и записывают в виде: $\alpha \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n$. 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: < >.

Пример грамматики, которая определяет язык целых десятичных чисел со знаком:
G ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, –, +}, {<число>, <чсл>, <цифра>}, P, <число>),
где P:

<число> → <чсл> | +<чсл> | –<чсл>
<чсл> → <цифра> | <чсл><цифра>
<цифра> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Составляющие элементы данной 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>;

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: $L(G) \cup \{\lambda\} = L(G') \cup \{\lambda\}$, где λ – пустая цепочка.

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.

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.

Для компиляторов языки делятся на простые и сложные и существуют жесткие критерии для такого деления. Сложность построения компилятора зависит от сложности языка программирования, для которого он создается.

Согласно классификации, предложенной Н.Хомским, формальные грамматики классифицируются по структуре их правил. Если все без исключения правила грамматики удовлетворяют некоторой заданной структуре, то такую грамматику относят к определенному типу. Выделяют четыре типа грамматик.

Тип 0: грамматики с фразовой структурой

На структуру их правил не накладывается никаких ограничений. Это самый общий тип грамматик. В него подпадают все без исключения формальные грамматики, но часть из них может быть также отнесена и к другим классификационным типам. 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** $\cup$ **VT** имеют правила вида: $\alpha_1 A \alpha_2 \to \alpha 1 \beta \alpha_2$, где $\alpha_1, \alpha_2 \in V^*$, $A \in$ **VN**, $\beta \in$ **V**$^+$.

Неукорачивающие грамматики G**(VT,VN, P,** S**),** **V** = **VN** $\cup$ **VT** имеют правила вида: $\alpha \to \beta$, 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 той или иной цепочкой символов в зависимости от контекста, в котором он встречается. Отсюда пошло и название «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, т.е. значение одного и того же символа может быть различным в зависимости от контекста, в котором он встречается.

Неукорачивающие грамматики имеют такую структуру правил, что при построении предложений языка, заданного грамматикой, любая цепочка символов может быть заменена цепочкой символов не меньшей длины.

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.

Тип 2: контекстно-свободные (КС) грамматики

КС-грамматики G (**VT, VN, P**, S), **V = VN** $\cup$ **VT** имеют правила вида: $A \to \beta$, где $A \in$ **VN**, $\beta \in$ **V**$^+$. Такие грамматики также иногда называют неукорачивающими контекстно-свободными (НКС) грамматиками (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** $\cup$ **VT**, правила которых могут иметь вид: $A \to \beta$, где $A \in$ **VN**, $\beta \in$ **V***.

Разница между этими двумя классами грамматик заключается в том, что в УКС-грамматиках в правой части правил может присутствовать пустая цепочка (λ), а в НКС-грамматиках нет.

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.

Тип 3: регулярные грамматики

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

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

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

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

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. 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.

Рассмотренная в примере грамматика, определяющая язык целых десятичных чисел со знаком относится к контекстно-свободным грамматикам (тип 2). Следовательно, ее можно отнести и к типу 0 и к типу 1. Данная грамматика не может быть отнесена к типу 3, поскольку правило <чсл> → <чсл><цифра> недопустимо для него.

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

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

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 – их количество зависит от реализации компилятора. Например, можно организовывать различные таблицы идентификаторов для различных модулей исходной программы или для различных типов элементов входного языка.

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 функций – имя функции; количество и типы формальных аргументов функции; тип возвращаемого результата; адрес кода функции.

Приведенный состав хранимой информации является только примерным. Конкретное наполнение таблиц идентификаторов зависит от реализации компилятора. Кроме того, не вся информация, хранимая в таблице идентификаторов, заполняется компилятором сразу – он может несколько раз выполнять обращение к данным в таблице идентификаторов на различных фазах компиляции. Например, имена переменных могут быть выделены на фазе лексического анализа, типы данных для переменных – на фазе синтаксического разбора, а область памяти связывается с переменной только на фазе подготовки к генерации кода. 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.

Следовательно, таблицы идентификаторов должны быть организованы таким образом, чтобы компилятор имел возможность максимально быстрого поиска нужного ему элемента.

Логарифмический поиск

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.

Поскольку поиск в таблице идентификаторов является наиболее часто выполняемой компилятором операцией, а количество различных идентификаторов в реальной исходной программе от нескольких сотен до нескольких тысяч элементов, то 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).

Построение таблиц идентификаторов по методу бинарного дерева

Можно сократить время поиска искомого элемента в таблице идентификаторов, не увеличивая значительно время, необходимое на ее заполнение. Для этого, надо отказаться от организации таблицы в виде непрерывного массива данных.

Существует метод построения таблиц, при котором 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».

Как уже было сказано, 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.

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

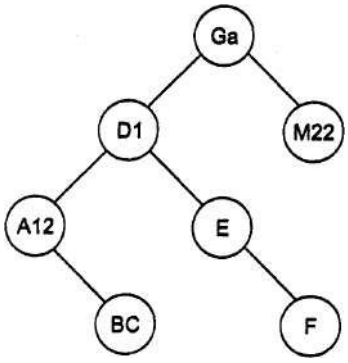


Рисунок 2. Структура бинарного дерева для последовательности идентификаторов GA, D1, M22, E, A12, BC, F

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

Шаг 1. Сделать текущим узлом дерева корневую вершину.

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

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

Шаг 4. Если очередной идентификатор меньше, то перейти к шагу 5, иначе – пе-

рейти 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.

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

Метод рехэширования

Хэш-функцией F называется некоторое отображение множества входных элементов R на множество целых неотрицательных чисел Z: $F(r) = n$, $r \in R$, $n \in Z$. Множество допустимых 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 выполняет отображение имен идентификаторов на множество целых неотрицательных чисел. Областью определения такой хэш-функции будет множество всех возможных имен идентификаторов.

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

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

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

Тогда таблицу идентификаторов можно организовать по следующему алгоритму размещения элемента:

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

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

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

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

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

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

Метод цепочек

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

Шаг 1. Во все ячейки хэш-таблицы поместить пустое значение, таблица 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, поместить в нее значение переменной FreePtr и перейти к шагу 5; иначе перейти к шагу 3.

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

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

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

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

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

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

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

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

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

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

ЛЕКСИЧЕСКИЕ АНАЛИЗ

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) – 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.

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. 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.

Распознаватели. Общая схема работы

Распознаватель – это специальный автомат, который позволяет определить принадлежность цепочки символов некоторому языку. Задача распознавателя заключается в том,

чтобы на основании исходной цепочки дать ответ на вопрос, принадлежит ли она заданному языку или нет.

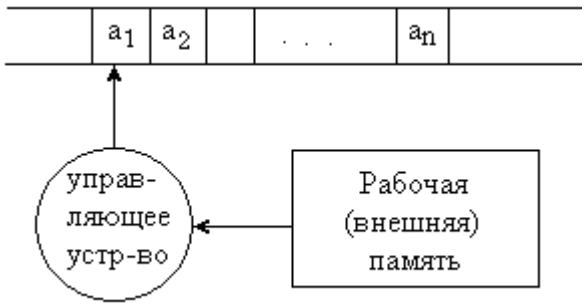
Распознаватель состоит из следующих компонентов:

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

- устройства управления (УУ), которое координирует работу распознавателя, имеет некоторый набор состояний и конечную намять (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.

В общем виде распознаватель можно отобразить в виде условной схемы, представленной на рисунке 3.



Условная схема 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 а*, если, находясь в начальной конфигурации и получив на вход эту цепочку, он может проделать последовательность шагов, заканчивающуюся одной из его конечных конфигураций.

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

Классификация распознавателей

Для каждого из типов языков существует свой тип распознавателя.

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 принадлежности входной цепочки заданному языку. Практического применения такие распознаватели 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, экспоненциально зависит от длины цепочки.

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 намятые – МП-автоматы. При простейшей реализации алгоритма работы такого автомата он имеет экспоненциальную сложность, однако путем некоторых у совершенствований 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 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.

Этого можно было бы избежать, если построить компилятор на основе контекстно-зависимого распознавателя, но скорость работы такою компилятора была бы недопустимо низка, поскольку время разбора в таком варианте будет экспоненциально зависеть от длины исходной программы. Комбинация из распознавателя КС-языка и дополнительного

семантического анализатора является более эффективной с точки зрения скорости разбора 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.

В компиляторах распознаватели на основе регулярных языков используются для лексического анализа текста исходной программы выделении в нем простейших конструкций языка (лексем), таких как идентификаторы, строки, константы т. и. 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.

Конечные автоматы

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

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.

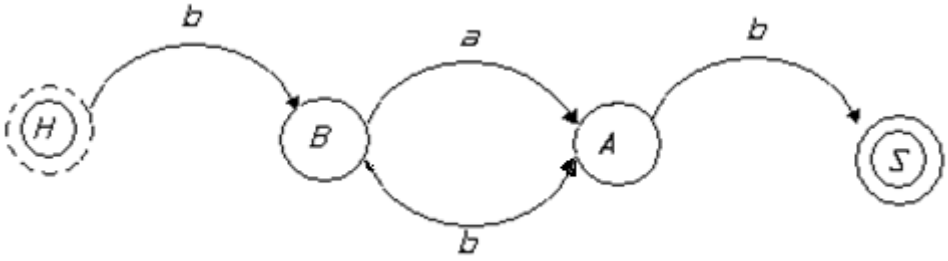
Конечный автомат называют полностью определенным, если в каждом его состоянии существует функция перехода для всех возможных входных символов.

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.



Конечный автомат *M({H, A, B, S}, {a, b}, δ, H, {s}); δ: δ(H, b)=B, δ(B, a)=A, δ(B, b)=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. Таблица задает КА с рисунка 5, приведенный к детерминированному полностью определенному виду.

	a	***b***	
H	E	B	0
B	A	A	0
A	E	S	0
S	E	E	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.

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 гене-

рации текста результирующей программы.

Без выполнения синтаксического разбора работа компилятора бессмысленна.

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.

Функционирование синтаксического анализатора и особенности алгоритма, лежащего в его основе, определяются принципами построения распознавателей для КС-языков.

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

Автоматы с магазинной памятью

В общем виде МП-автомат определяют как R (Q, V, Z, δ, qo, Zo, F),

где Q – множество состояний автомата;

V – алфавит входных символов автомата;

Z – специальный конечный алфавит магазинных символов автомата,

δ – функция переходов автомата,

$q_o \in Q$ – начальное состояние автомата;

$z_o \in Z$ – 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 автомата 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. 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, в отличие от обычного МП-автомата. В отличие от обычного МП-автомата, который на каждом такте работы может изымать из стека только один символ, расширенный МП-автомат может изымать за один такт цепочку символов, находящуюся на вершине стека.

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

ГЕНЕРАЦИЯ И ОПТИМИЗАЦИЯ КОДА

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

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

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

Выдачу пользователю сообщений об обнаруженных ошибках и предупреждениях;

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

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

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

Принципиально различаются два основных вида оптимизирующих преобразований:

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

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

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

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).

Далеко не все лексические единицы языка требуют для себя выделения памяти.

Распределение памяти на локальные и глобальные области целиком определяется семантикой языка исходной программы. Статическая и динамическая память так же может быть глобальной и локальной.

Динамическая память может распределяться либо разработчиком программы, либо автоматически компилятором. Многие компиляторы объектно-ориентированных языков программирования для работы с ней используют специальный менеджер памяти, который следит за рациональным ее использованием. Как правило, роль менеджера заключается в том, что при первом запросе на требование ОП он запрашивает у ОС область памяти значительно большего размера, чем необходимо результирующей программе. Но затем он работает, не обращаясь к функциям ОС, пока она вся не будет использована. Это сокращает количество обращений программы к системным функциям ОС, что увеличивает ее быстродействие. Менеджер памяти сокращает фрагментацию ОП, за счет распределения большими участками и возможности перераспределения уже используемой памяти.

Код менеджера памяти включается в текст результирующей программы или представляется в виде отдельной динамически загружаемой библиотеки. В Паскале и С++ менеджер памяти не обязателен, нов Java, где динамическая память не может выделяться пользователем, его функции достаточно сложны.

Еще одним понятием в механизме распределения памяти является дисплей памяти функции – это область данных, доступных для обработки в этой функции. Как правило, он включает:

глобальные данные всей программы,

формальные аргументы функции,

локальные данные,

адрес возврата – адрес того фрагмента кода результирующей программы, куда передается управление, после завершения функции.

Адрес возврата должен быть сохранен на все время выполнения функции.

Современные вычислительные системы ориентированы главным образом на стековую организацию дисплея памяти. В ней при вызове функции все ее параметры и адрес возврата помещаются в специальный стек параметров. Верхушка стека адресуется одним из регистров процессора. Для доступа к данным используется еще один регистр процессора – базовый регистр.

В начале своего выполнения функция запоминает в стеке значение базового регистра, затем запоминает состояние стека в базовом регистре и увеличивает его значение на размер памяти, необходимой для хранения локальных переменных.

При выполнении функции доступ к локальным переменным осуществляется через базовый регистр. Параметры лежат в стеке ниже места, указанного базовым регистром, а локальные переменные и константы – выше, указанного базовым, но ниже места, указанного регистром стека.

При завершении функции регистру стека присваивается значение базового регистра, извлекается значение базового регистра и адреса возврата. Затем выталкиваются все параметры функции, а управление передается по адресу возврата.

Если происходит несколько вложенных вызовов функций, их параметры и переменные помещаются в стек последовательно.

Для повышения эффективности программ и увеличения скорости вызова функций компиляторы могут предусматривать передачу части параметров через свободные регистры процессора.

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ К ПРАКТИЧЕСКИМ ЗАНЯТИЯМ

Лабораторные работы проводятся по подгруппам в компьютерном классе, оборудование которого подключено к ЛВС университета и имеет возможность выхода в Интернет. На первом занятие обязательно проводится инструктаж по выполнению техники безопасности.

Задания к лабораторным работам выполняются студентами парами на одном компьютере (работа в команде). Задания к лабораторным работам формируются на основе материала, изложенного на лекциях и практических занятиях.

Каждый студент (рабочая группа) получает индивидуальный вариант для выполнения задания лабораторной работы. Задания к лабораторным работам выдаются заранее, как правило, в начале семестра, и для их успешного их выполнения необходимо предварительное освоение теоретического материала.

Для подготовки к выполнению лабораторных работ и повторения, усвоения (изучения пропущенного) теоретического материала студентам рекомендуется самостоятельно организовать по месту проживания дополнительное рабочее место, оборудованное персональным компьютером, подключённым к сети интернет, и установленным программным обеспечением, необходимым для разработки программ и указанным в рабочей программе.

Желательно готовиться к лабораторным работам заранее. Задание выдается преподавателем заранее.

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

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

Тема Алгоритмы построения таблиц идентификаторов

Изучив теоретический материал, рассмотренный на практических занятиях, ответьте на контрольные вопросы:

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

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

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

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

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

6. Использование значения, возвращаемого хэш-функцией, в качестве адреса

ячейки из некоторого массива данных называется

а) хэш-адресацией б) хэш-функцией

в) хэшированием г) рехэшированием

Варианты индивидуального задания

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

б) Написать программу, реализующую создание таблицы идентификаторов на основе метода рехэширования. В качестве исходных данных использовать файл, считая в нем все исходные слова идентификаторами. В качестве хэш-функции использовать коды первых двух букв идентификаторов;

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

г) Сравнить реализованные методы построения таблиц идентификаторов.

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

б) Написать программу, реализующую создание таблицы идентификаторов на основе метода рехэширования. В качестве исходных данных использовать файл, считая в нем все исходные слова идентификаторами. В качестве хэш-функции использовать коды первых двух букв идентификаторов;

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

г) Сравнить реализованные методы построения таблиц идентификаторов.

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

б) Написать программу, реализующую создание таблицы идентификаторов на основе метода рехэширования. В качестве исходных данных использовать файл, считая в нем все исходные слова идентификаторами. В качестве хэш-функции использовать коды последних двух букв идентификаторов;

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

г) Сравнить реализованные методы построения таблиц идентификаторов.

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

б) Написать программу, реализующую создание таблицы идентификаторов на основе метода рехэширования. В качестве исходных данных использовать файл, считая в нем все исходные слова идентификаторами. В качестве хэш-функции использовать коды последних двух букв идентификаторов;

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

г) Сравнить реализованные методы построения таблиц идентификаторов.

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

б) Написать программу, реализующую создание таблицы идентификаторов на основе метода рехэширования. В качестве исходных данных использовать файл, считая в нем все исходные слова идентификаторами. В качестве хэш-функции использовать некоторое случайное число в диапазоне от -10 до 10;

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

г) Сравнить реализованные методы построения таблиц идентификаторов.

Тема Классификация грамматик

Изучив теоретический материал по теме грамматики, ответьте на вопросы

1. Как выглядит описание грамматики в форме Бэкуса-Наура?
2. Каковы составляющие формального описания грамматики?
3. Как классифицируются языки? Как их классификация соотносится с классификацией грамматик?
4. Почему язык программирования нельзя не является чисто формальным языком?
5. Какой тип грамматики самый сложный?

6. Грамматики G(**VT,VN, P**, S), **V = VN** $\cup$ **VT** , которые могут иметь правила двух видов: A $\rightarrow$ Bγ или A $\rightarrow$ γ, где A, B$\in$ **VN**, $\gamma \in$ **VT*** относятся к типу:

а) праволинейных грамматик ;

б) леволинейных грамматик;

в) контексно-свободных грамматик;

г) контексно-зависимых грамматик.

7. Выберите верное утверждение:

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

б) целевой символ грамматики – это всегда терминальный символ;

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

г) языки программирования являются формальными языками.

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

а) контекстно-свободной грамматикой

б) грамматикой с фразовой структурой

в) контексно-зависимой грамматикой

г) регулярной грамматикой

Задание

1. Определить тип указанных грамматик

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

P: A$\rightarrow$ B | +B | –B

B$\rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0B | 1B | 2B | 3B | 4B | 5B | 6B | 7B | 8B | 9B

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

P: B$\rightarrow$ + | – | λ

S$\rightarrow$ B0 | B1 | B2 | B3 | B4 | B5 | B6 | B7 | B8 | B9 | S0 | S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8 | S9

1.3) G3 ({0, 1}, {A, S}, P, S) P: S→ 0A1
0A→00A1
A→ λ

1.4) G4 ({0, 1}, {A, S}, P, S) P: S→ 0A1 | 01
0A→00A1 | 001
A→ λ

1.5) G5 ({0, 1}, { S}, P, S) P: S→ 0S1 | 01

1.6) G5 ({f, g, h}, { G, H, E, S}, P, S) P: S→ GH
G → fGgH | fg
Hg →gH
HE → Hh
gEh → ghh
fgE → fgh

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

3. Поездом называется произвольная последовательность локомотивов и вагонов. Построить грамматику в форме Бэкуса-Наура для понятия «поезд», если
3.1) поезд всегда начинается с локомотива;
3.2) все локомотивы должны быть сосредоточены в начале поезда;
3.3) поезд начинается с локомотива и заканчивается локомотивом;
3.4) в поезде должны чередоваться через два локомотивы и вагоны;
3.5) поезд не должен содержать два локомотива или два вагона подряд;
3.6) поезд не должен содержать подряд два локомотива;
3.7) поезд не должен содержать три локомотива подряд;
3.8) поезд не должен содержать более пяти вагонов подряд;
3.9) поезд не должен заканчиваться локомотивом;
3.10) поезд не должен содержать более двух локомотивов

4. Указать к какому типу относится каждая из грамматик языка десятичных чисел с фиксированной точкой:
G1 ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, – , “.”}, {<число>, <цел>, <дроб>, <цифра>, <осн>, <знак>}, P1, <число>)
P1: <число> → <знак> <осн>
<знак> → λ | + | -
<осн> → <цел>. <дроб> | <цел>.
<цел> → <цифра> | <цифра> <цифра>
<дроб> → λ | <цел>
<цифра> <цифра> → <цифра> <цифра> <цифра>
<цифра> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7| 8 | 9

G2 ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, – , “.”}, {<число>, <часть>, <цифра>, <осн>}, P2, <число>)
P2: <число> → +<осн>| -<осн> | <осн>
<осн> → <часть>.<часть> |<часть>. |<часть>
<часть> → <цифра> | <цифра> <цифра>

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

G3 ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, – , “.”}, {<число>, <часть>, <осн>}, P3, <число>)
P3: <число> → +<осн>| -<осн> | <осн>
<осн> → <часть>. | <часть> | <осн>0 | <осн>1 | <осн>2 | <осн>3 | <осн>4 | <осн>5 | <осн>6 | <осн>7| <осн>8 | <осн>9
<часть> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7| 8 | 9 | <часть>0 | <часть>1 | <часть>2 | <часть>3 | <часть>4 | <часть>5 | <часть>6 | <часть>7| <часть>8 | <часть>9

5. Определить является ли однозначной каждая из грамматик, описанная в задание 2.

Тема Конечные автоматы

Конечный автомат задается пятеркой вида M(Q, V, δ, q_0, F),

где Q – конечное множество состояний автомата,

V – алфавит входных символов,

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

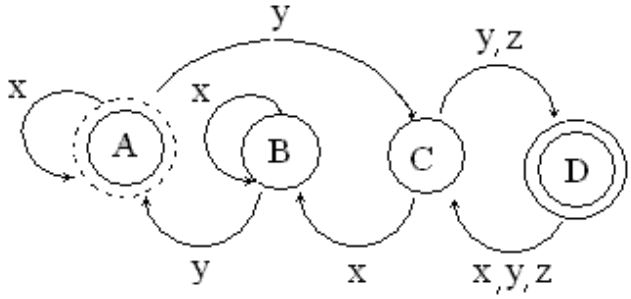
q_0 – начальное состояние автомата ($q_0 \in Q$),

F – непустое множество конечных состояний автомата.

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

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

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



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

столбцы помечены входными символами,

строки помечены символами состояний,

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

первая строка помечена символом начального состояния,

строки, соответствующие допускающим (заключительным) состояниям помечены

справа единицами, а строки, соответствующие отвергающим состояниям, помечены нулями.

Таблица переходов полностью определенного конечного автомата, представленного на рисунке и преобразованного в детерминированный, задается таблицей:

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

Контрольные вопросы

1. Может ли граф переходов конечного автомата использоваться для однозначного определения автомата? Почему?
2. От каких параметров зависит функция переходов конечного автомата?
3. В каком случае конечный автомат называется полностью определенным?
4. Всегда ли недетерминированный конечный автомат может быть приведен к детерминированному?

Задание

1. Построить конечный автомат, распознающий зарезервированные слова языка C++:

   Вариант 1) enum, extern, explicit
   Вариант 2) continue, class, const
   Вариант 3) private, protected, public
   Вариант 4) union, using, unsigned
   Вариант 5) double, delete, default
   Вариант 6) virtual, void, volatile
   Вариант 7) switch, struct, sizeof
   Вариант 8) typedef, true, this,
   Вариант 9) register, return,
   Вариант 10) float, false, for
2. Для реализации конечного состояния построить граф или таблицу переходов.

Реализовать конечный автомат программной на языке программирования высокого уровня.

Тема Преобразования конечного автомата

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

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

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

Шаг 3. Обозначим $q'_0 = [q_0]$.

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

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

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

Шаг 2. $P_{i+1}=\varnothing$.

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

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

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

Контрольные вопросы

1. В чем заключается алгоритм преобразования конечного автомата к детерминированному виду?
2. В каких случаях преобразовывать конечный автомат к детерминированному виду нецелесообразно?
3. Какие два состояния автомата называются эквивалентными?
4. Что собой представляет таблица эквивалентных состояний? Каким образом она заполняется?
5. Как определяется недостижимое состояние?
6. Какое из преобразований приводит к уменьшению количества состояний конечного автомата, а какое к их уменьшению?

Задание

1. Найти различающую цепочку для пары автоматов:

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

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

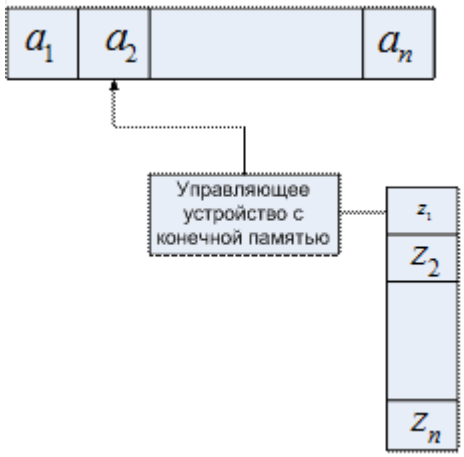
2. Найти минимальную эквивалентную таблицу для каждого из ниже расположенных автоматов и недостижимые состояния.

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

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

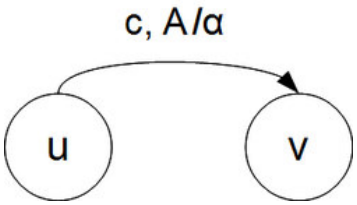
Тема Автоматы с магазинной памятью (МП-автоматы)

Автомат с магазинной памятью – тип распознавателя, представляющий собой естественную модель синтаксических анализаторов контекстно-свободных языков. Автомат с магазинной памятью – это односторонний недетерминированный распознаватель, в потенциально бесконечной памяти которого хранятся элементы информации. Они используются так же, как патроны в магазине автоматического оружия, т.е. в каждый момент доступен только верхний элемент магазина. Распознаватель этого типа изображен на рисунке.

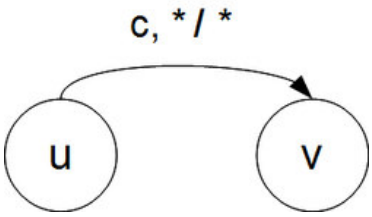


Магазин (или магазинный список, или магазинную ленту) представляют в виде цепочки символов, причем верхним элементом магазина будем считать самый левый или самый правый символ цепочки в зависимости от того, что удобнее в данной ситуации. Пока будем считать верхним самый левый символ цепочки, представляющей магазинный список.

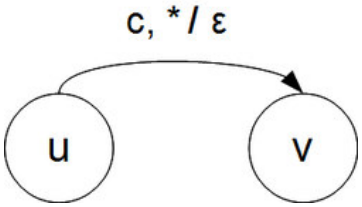
Примеры диаграмм переходов автомата с магазинной памятью:



На рисунке показан переход из состояния U в состояние V, причем
c – символ, прочитанный с ленты;
A – символ, вынутый из стека;
α – строка, помещаемая в стек.

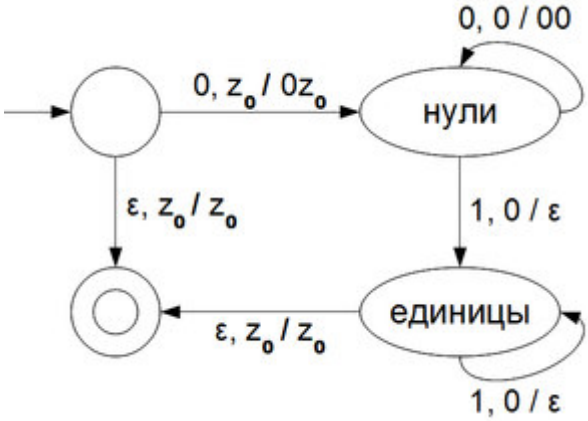


Переход происходит по любому стековому символу, он же возвращается в стек.



Переход происходит по любому стековому символу, в стек кладется пустая строка.

Пример недетерминированного МП-автомата для языка $0^n 1^n$, построенного графом состояний



Данный автомат можно описать и аналитически через функцию переходов.
R=({q0, q1, q2}, {0, 1}, {Z, 0}, δ, q0, Z, {q0}),
где δ(q0, 0, Z) = {(q1, 0Z)},
δ(q1, 0, 0) = {(q1, 00)},

δ(q0, 1, 0) = {(q2, λ)},

δ(q2, 1, 0) = {(q2, λ)},

δ(q2, λ, Z) = {(q0, λ)}.

Задание

1. Построить последовательность тактов работы автомата для цепочек 00110011 и 00110 11
2. Построить МП-автомат, распознающий язык правильных скобочных выражений.
3. Построить МП-автомат, распознающий язык $a^k b^m c^k$, причем k=m или m=k.

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ К ЛАБОРАТОРНЫМ ЗАНЯТИЯМ

ПЕРЕЧЕНЬ ЛАБОРАТОРНЫХ ЗАНЯТИЙ

1. Работа с утилитами в операционной системе Windows .
2. Реализация алгоритма построения таблиц идентификаторов «Логарифмический поиск»
3.Операции с двоичными, восьмеричными и шестнадцатеричными числами
4. Реализация алгоритма построения таблиц идентификаторов «Бинарное дерево»
5. Реализация алгоритма построения таблиц идентификаторов «Метод цепочек»
6. Настройка рабочего окружения для Ассемблер
7. Применение bat-файлов
8. Арифметические инструкции языка ассемблер
9. Логические инструкции языка ассемблер
10. Реализация циклов в ассемблере

Пример лабораторного занятия 10.

Лабораторное занятие
Реализация циклов в ассемблере

Целью работы является приобретение навыков написания программ с циклами.

ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

При организации циклов широко используются команды INC(инкремент) иDEC(декремент), что означает добавление или вычитание единицы из целого числа, помещенного в ячейку памяти, РОН или индексный регистр. Команды имеют формат:

INCоперанд ,

DECоперанд.

Такую программную конструкцию как цикл можно реализовать, используя в программе операции инкремента, декремента, условного и безусловного переходов. Но, учитывая важность такого алгоритмического элемента, как цикл, разработчики ассемблера предусмотрели специальные команды цикла, например:

LOOPметка_перехода.

Команда означает ’повторить цикл’. Выполнение команды заключается в следующем:

- вычитании 1 из регистра CX;
- сравнении регистра CX с нулем;
- если CX=0, то управление передается на следующую после LOOPкоманду, иначе осуществляется передача управления на метку_перехода.

Другими командами цикла являются команды:

LOOPE/LOOPZметка_перехода,

которые означают “повторить цикл, пока CX<>0 или ZF=0”. Обе команды совершенно идентичны, поэтому использовать можно любую из них. Отличаются эти команды от предыдущей команды анализом окончания цикла:

- если CX>0 и ZF=1, управление передается на метку_перехода, иначе если CX=0 илиZF=0, то выполняется следующая после командыLOOPE/LOOPZкоманда.

Еще одной модификацией являются команды цикла

LOOPNE/LOOPNZметка_перехода,

которые означают, “повторить цикл, пока CX<>0 или ZF=1”. Как и в предыдущем случае обе команды совершенно идентичны. В них анализ окончания цикла выполняется по следующему правилу:

- если CX>0 и ZF=0, управление передается на метку перехода, иначе если CX=0 илиZF=1, то выполняется следующая после команды LOOPNE/LOOPNZоперация.

Общая особенность команд цикла в том, что они используют регистр общего назначения CX как счетчик числа повторений цикла, поэтому при их использовании не забудьте до метки перехода послать в этот регистр нужное число – количество повторений цикла!

Недостаток всех команд цикла в том, что они реализуют только короткие переходы. Для работы с длинными циклами используются комбинации команд условного перехода и безусловного перехода.

Приведем пример использования вышеописанных команд в контексте подсчета количества нулевых, положительных и отрицательных элементов вектора (одномерного массива), состоящего из однобайтовых чисел.

Описания переменных в сегменте данных могут быть следующими:

```
Masdb-1, 0, 3,-8,0,9,-6,1,2,-5 ; заданный вектор
Len_mas=$-mas;количество элементов в векторе
Sch_0db0 ;счетчик нулевых элементов вектора
Sch_poldb0 ;счетчик положительных элементов вектора
Sch_otrdb0 ;счетчик отрицательных элементов вектора.
```

Фрагмент сегмента кода для подсчета элементов может быть следующим:

```
Movcx,len_mas;инициализация счетчика цикла
Xorsi,si;инициализация индексного регистра
Cycl:cmpmas[si],0 ;сравниваем элемент вектора с 0
Jzzero;нуль-элементы считаем в блокezero
Jgpol;элементы>0 считаем в блокepol
IncSch_otr;увеличиваем счетчик элементов <0
Jmpkon_cycl
Zero:IncSch_0 ;увеличиваем счетчик нулевых элементов
Jmpkon_cycl
pol:IncSch_pol ;увеличиваем счетчик элементов >0
kon_cycl:incsi;переходим к следующему элементу вектора
loopcycl;завершаем цикл.
```

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Изучить приведенный теоретический материал к лабораторной работе.
2. Написать программы в соответствии с заданным преподавателем вариантом.
3. Оттранслировать программы в объектный код.
4. Провести отладку программ и проверить получаемые результаты.

СОДЕРЖАНИЕ ОТЧЕТА

Отчет должен включать:

- титульный лист;
- описание цели работы;
- описание задания на лабораторную работу;
- словесные пояснения к алгоритму решения задачи и схему программы;
- листинги программ;
- результаты выполнения программ;
- выводы.

ВАРИАНТЫ ЗАДАНИЙ

1. Подсчитать количество положительных и отрицательных элементов в заданном векторе и определить, каких элементов в векторе больше

а) элементы вектора однобайтовые;

б) элементы вектора двухбайтовые.

2. Подсчитать количество нулевых и ненулевых элементов в заданном векторе и определить, каких элементов в векторе больше

а) элементы вектора однобайтовые;
б) элементы вектора двухбайтовые.
3. Подсчитать количество неотрицательных элементов в заданном двумерном массиве
а) элементы массива однобайтовые;
б) элементы массива двухбайтовые.
4. Подсчитать количество неположительных элементов в заданном двумерном массиве
а) элементы массива однобайтовые;
б) элементы массива двухбайтовые.
5. Подсчитать количество положительных и отрицательных элементов в заданном двумерном массиве и определить, каких элементов в нем больше
а) элементы массива однобайтовые;
б) элементы массива двухбайтовые.

Примечание к кодированию пунктов 1-5: в программе необходимо реализовать такую конструкцию, как “вложенные циклы”. Кроме того, понятие массива и индексации массива весьма условны, ибо в памяти ЭВМ элементы массива располагаются последовательно, строка за строкой, в результате чего физическая структура двумерного массива и вектора (одномерного массива) оказываются одинаковыми. Отличие двумерного массива и вектора заключается в интерпретации области памяти, отведенной этим структурам. Наращивание индекса элемента структуры определяется алгоритмом обработки.

Контрольные вопросы

1. Каков синтаксис команд условного перехода?
2. Какие флаги анализируют команды безусловного перехода?
3. Как формируется машинный код команды безусловного перехода ассемблера?
4. Что такое близкий и дальний переходы в ассемблере?
5. Как различить в командах прямой и косвенный переходы?
6. Какие действия выполняют команды цикла в ассемблере?
7. Какую команду необходимо предусмотреть перед меткой перехода для цикла?

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ К САМОСТОЯТЕЛЬНОЙ РАБОТЕ СТУДЕНТОВ

Самостоятельная работа включает два вида – аудиторную и внеаудиторную. В первом случае она выполняется на учебных занятиях под руководством преподавателя и по его заданию. Студенты обеспечиваются необходимым учебным материалом и дидактическими материалами.

Внеаудиторная самостоятельная работа выполняется по заданию преподавателя, но без его непосредственного участия. Видами заданий для внеаудиторной работы являются: изучение текста учебной литературы, конспектирование текста, работа с конспектом лекции, ответы на контрольные вопросы при выполнении индивидуального задания, тестирование, решение задач, продумывание алгоритма будущей программы, работа с компьютером, а именно, кодирование и отладка программы, подготовка отчета по лабораторным заданиям, подготовка к сдаче зачета.

Задания к лабораторным работам выдаются заранее, как правило, на первом занятие текущего семестра, и для их успешного их выполнения необходимо предварительное освоение теоретического материала и разбор, приведенных на лекции примеров программ, проработка алгоритма решения разобранных задач и составление собственных алгоритмов. Для этого наряду с конспектами можно воспользоваться учебно-методическим обеспечением для самостоятельной работы, указанным в рабочей программе, и самопроверкой с помощью тестовых заданий, размещенных там же.

В отчете по выполнению индивидуального варианта заданий к лабораторным занятиям должны содержатся следующие сведения: формулировка задания, входные и выходные данные, текст программы, тестовые (контрольные) значения входных данных и рассчитанные выходные данные.

Для подготовки к выполнению лабораторных работ и повторения, усвоения (изучения пропущенного) теоретического материала студентам рекомендуется самостоятельно организовать по месту проживания дополнительное рабочее место, оборудованное персональным компьютером, подключённым к сети Интернет, и установленным программным обеспечением, необходимым для разработки программ и указанном в рабочей программе.

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

Итоговый контроль – дифференцированный зачет проводится на основании перечней вопросов, представленных в рабочей программе. Подготовка к зачету заключается в изучении и тщательной проработке студентом конспектов по всем видов занятий в соответствии с перечнем вопросов, представленном в рабочей программе дисциплины. Подготовку к зачету требуется начинать с просмотра перечня всех вопросов с целью оценки требуемого объема учебного материала, логики и структуры построения курса. С учетом накопленных за семестр знаний студент должен запланировать распределение времени на подготовку. Желательно зарезервировать время для повторения материала. Работа над каждым из вопросов рекомендуется прочитать конспект лекции, дополнительно прочитать рекомендованный учебник, если материал трудно усваивается. Завершается работа восстановление в памяти прочитанного.

Зачет проводится в виде ответов на вопросы или тестирования. Ответы на поставленные вопросы студент дает после предварительной подготовки. Преподаватель имеет право задать дополнительные вопросы, если ответ дан неполный или затруднительно однозначно оценить ответ. При оценке ответа учитывается: полнота ответа на поставленный вопрос, точность формулировок, логичность ответа, умение делать выводы и использования специализированной терминологии.