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

СИСТЕМНОЕ ПРОГРАМНОЕ ОБЕСПЕЧЕНИЕ
сборник учебно-методических материалов

для направления подготовки 09.04.04 Программная инженерия

Благовещенск

2018

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

Составитель: Галаган Т.А.

Системное программное обеспечение: сборник учебно-методических материалов для направлений подготовки 09.04.04 – Благовещенск: Амурский гос. ун-т, 2018

Рассмотрен на заседании кафедры информационных и управляющих систем 19.02.2018, протокол № 6

КРАТКОЕ ИЗЛОЖЕНИЕ ЛЕКЦИОННОГО МАТЕРИАЛА

КЛАССИФИКАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

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

К ПО относится также вся область деятельности *по* проектированию и разработке *ПО*: технология проектирования программ; методы тестирования программ; методы доказательства правильности программ; *анализ* качества работы программ; *документирование программ*; разработка и использование программных средств, облегчающих процесс проектирования программного обеспечения и другое.

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 представлена на рисунке:

Рисунок 1. Взаимосвязь системного и прикладного ПО

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 програм-*

ма 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.

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.

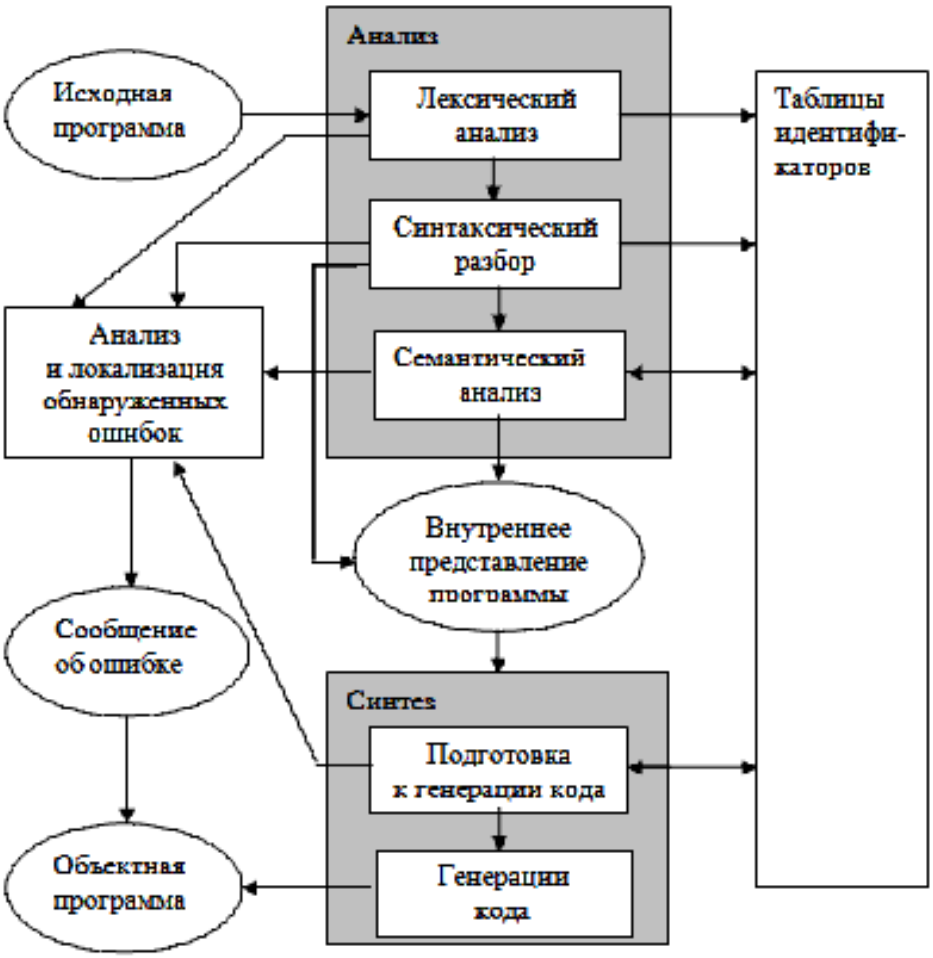


Рисунок 2. Общая схема работы компилятора

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 реализациях компиляторов семантический анализ может частично входить в фазу синтаксического разбора, частично – в фазу подготовки к генерации кода.

Подготовка к генерации кода – фаза, на которой компилятором выполняются предварительные действия, непосредственно связанные с синтезом текста результирующей программы, но еще не ведущие к порождению текста на выходном языке. Обычно в эту фазу входят действия, связанные с идентификацией элементов языка, распределением памяти и т. п.

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

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 послед-

него прохода – объектная программа.

В качестве внешней памяти могут выступать любые носители информации – оперативная память компьютера, накопители на магнитных дисках и т п. Современные компиляторы стремятся максимально использовать для хранения данных оперативную память компьютера, 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.

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.

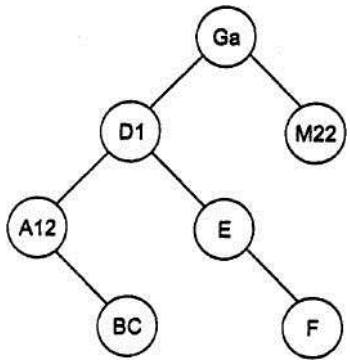


Рисунок 3. Структура бинарного дерева для последовательности идентификаторов GA, D1, M22, E, A12, BC, F

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 – перейти к шагу 6.

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

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

ется.

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_i 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_i 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 воз-

никают 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 АНАЛИЗАТОРЫ (СКАНЕРЫ)

Лексема (лексическая единица языка) – это структурная единица языка, состоящая из элементарных символов языка и не содержащая в своем составе других структурных единиц языка.

Лексемами языков программирования являются идентификаторы, константы, ключевые слова языка, знаки операций и т. п.

Лексический анализатор (или сканер) – 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 4.

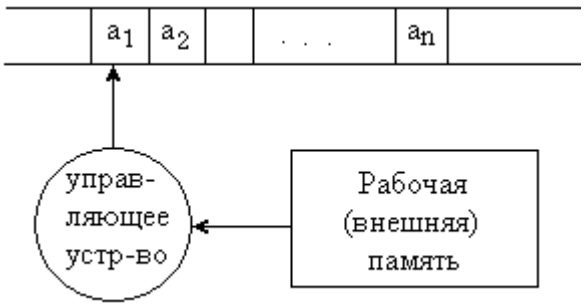


Рисунок 4. Условная схема распознавателя

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.

Простота и высокая скорость работы распознавателей определяют широкую область применения регулярных языков. В компиляторах распознаватели на основе регулярных языков используются для лексического анализа текста исходной программы 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 указа-

теля 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.

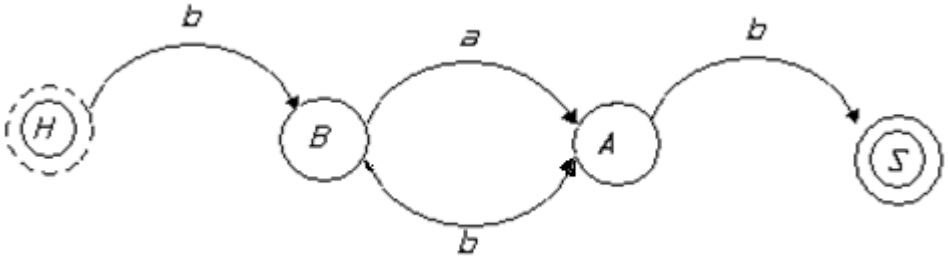


Рисунок 5 Конечный автомат *M({H, A, B, S}, {a, b}, δ, H, {s}); δ: δ(H, b)=B, δ(B, a)=A, δ(B, b)=A, δ(A, b)={B, S}*

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 генерации 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. МП-автомат условно можно представить в виде схемы, показанной на рисунке 6.

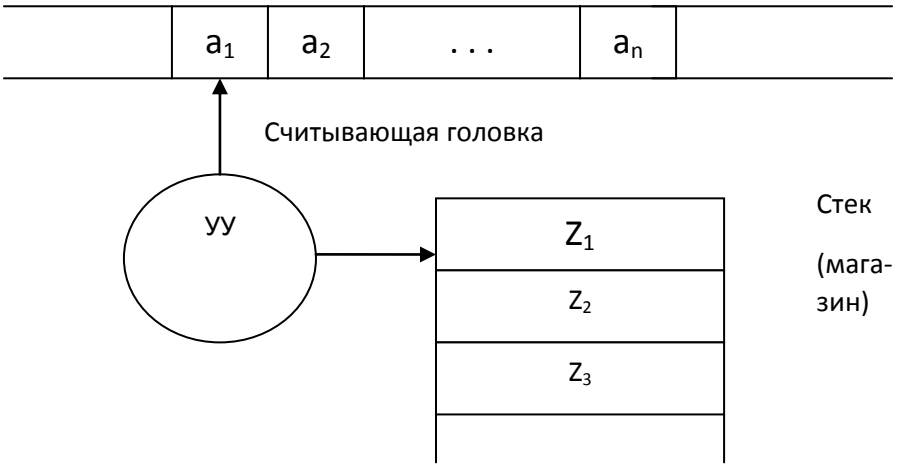


Рисунок 6. Схема МП-автомата

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.

Построение синтаксических анализаторов

Синтаксический анализатор должен распознавать весь текст исходной программы. Поэтому, в отличие от лексического анализатора, ему нет необходимости искать границы распознаваемой строки символов. Он должен воспринимать всю информацию, поступающую ему на вход, и либо подтвердить ее принадлежность входному языку, либо сообщить об ошибке в исходной программе.

Но, как и в случае лексического анализа, задача синтаксического анализа не ограничивается только проверкой принадлежности цепочки заданному языку. Необходимо оформить найденные синтаксические конструкции для дальнейшей генерации текста ре-

зультирующей программы. Синтаксический анализатор должен иметь некий выходной язык, с помощью которого он передает следующим фазам компиляции информацию о найденных и разобранных синтаксических структурах. В таком случае он уже является не разновидностью МП-автомата, а преобразователем с магазинной памятью – МП-преобразователем

Процесс построения синтаксического анализатора гораздо сложнее аналогичного процесса для лексического анализатор, поскольку КС-грамматики и МП-автоматы сложнее, чем регулярные грамматики и КА.

Построение синтаксического анализатора – это более творческий процесс, чем построение лексического анализатора, поскольку он не всегда может быть полностью формализован.

Имея грамматику входного языка, разработчик синтаксического анализатора должен в первую очередь выполнить ряд формальных преобразований над этой грамматикой, что облегчит в дальнейшем построение распознавателя. Затем проверить, подпадает ли полученная грамматика под один из известных классов КС-языков, для которых существуют линейные распознаватели. Если такой класс найден, можно строить распознаватель (если найдено несколько классов – выбрать тот, для которого построение распознавателя проще, либо построенный распознаватель обладает лучшими характеристиками). Если же такой класс КС-языков не удалось найти, разработчик должен попытаться выполнить над грамматикой дополнительные преобразования, чтобы привести ее к одному из известных классов.

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++ ме-

неджер 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.

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Операционная система (ОС) – это комплекс специальных программных средств, предназначенных для управления загрузкой, запуском и выполнением других (пользовательских) программ, а также для планирования и управления вычислительными ресурсами ЭВМ.

В ее состав входят взаимосвязанные программы, неоднородные по характеру и многоплановые по уровню. Этот комплекс программ динамичен по своему составу: из него можно удалять и в него добавлять некоторые составные части.

Операционные системы относятся к системному программному обеспечению.

Рисунок 7. Классификация операционных систем

Архитектура ОС – структурная организация на основе различных программных модулей.

В макроядерной (монолитная) архитектуре все модули поделены на две группы.

1. Ядро – модули, выполняющие основные функции операционной системы, решающие внутрисистемные задачи организации вычислительного процесса, такие как переключение контекстов, обработка прерываний и др.

2. модули ОС выполняют полезные, но менее обязательные функции. Их обычно подразделяют на следующие группы:

- утилиты – программы, решающие отдельные задачи управления и сопровождения компьютерной системы, такие, например, как программы сжатия дисков, архивирования данных на магнитную ленту;

- системные обрабатывающие программы – текстовые или графические редакторы, компиляторы, компоновщики, отладчики;

- программы представления пользователю дополнительных услуг – специальный вариант пользовательского интерфейса, калькулятор и даже игры;

- библиотеки процедур различного назначения, упрощающие разработку приложений, например, библиотека ввода-вывода, специальных символов.

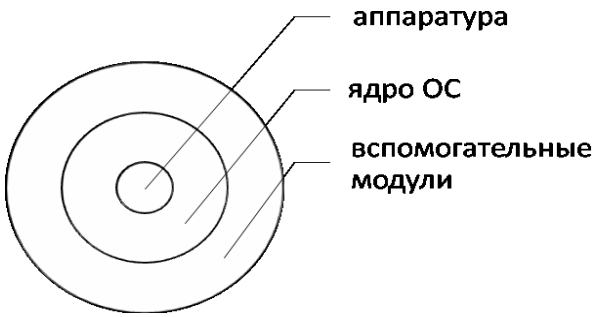


Рисунок 7. Архитектура вычислительной системы, работающей под управлением макроядерной ОС

ПРОЦЕССЫ И ПОТОКИ

Внутренние единицы работы операционной системы:
процесс – задача на стадии выполнения;
поток (нить) – последовательность команд, часть процесса.

Процесс рассматривается ОС как заявка на потребление всех видов ресурсов, кроме одного – процессорного времени. Процессорное время распределяется ОС между – *потоками.*

При управлении процессами операционная система использует два основных типа информационных структур:

дескриптор процесса, содержащий информацию, необходимую ядру в течение всего жизненного цикла процесса (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, содержащий менее оперативную, но более объемную часть информации, необходимую для возобновления выполнения процесса с прерванного места (содержимое регистров процессора, коды ошибок выполняемых процессором системных вызовов, информация обо всех файлах, открытых процессом, о незавершенных операциях ввода-вывода).

Контекст процесса также как дескриптор 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 и обратно.

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 одного потока на другой.

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

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.

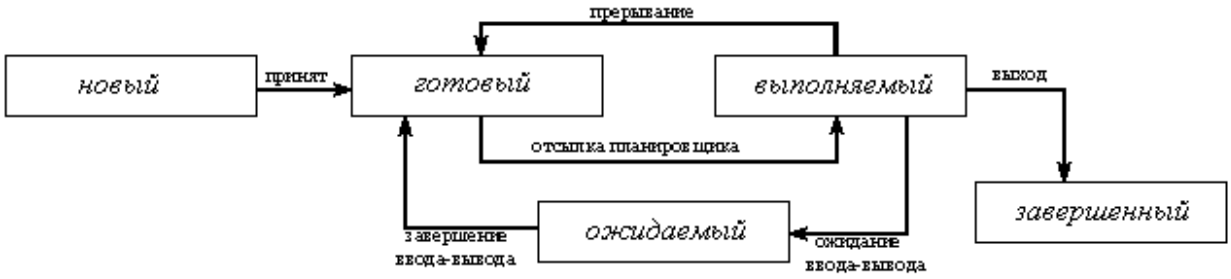


Рисунок 9 – Типовая диаграмма переходов для состояний процесса.

Квантом называют ограниченный непрерывный период процессорного времени, которой предоставляется каждому потоку поочередно для выполнения.

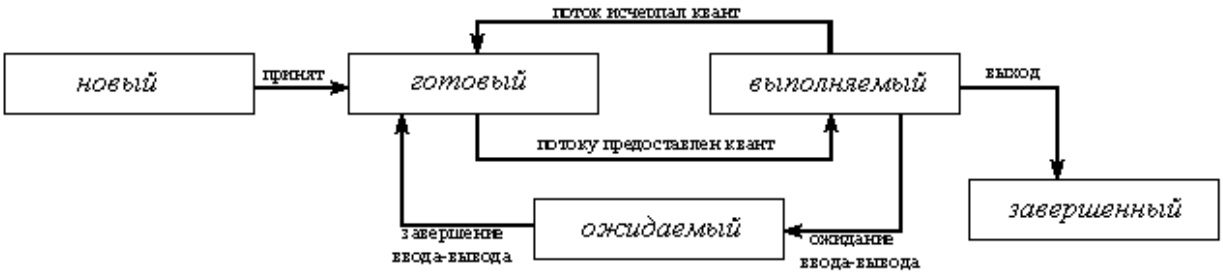


Рисунок 10 – Диаграмма переходов в алгоритме квантования

Рисунок 11 – Диаграмма переходов в алгоритме с абсолютным приоритетом

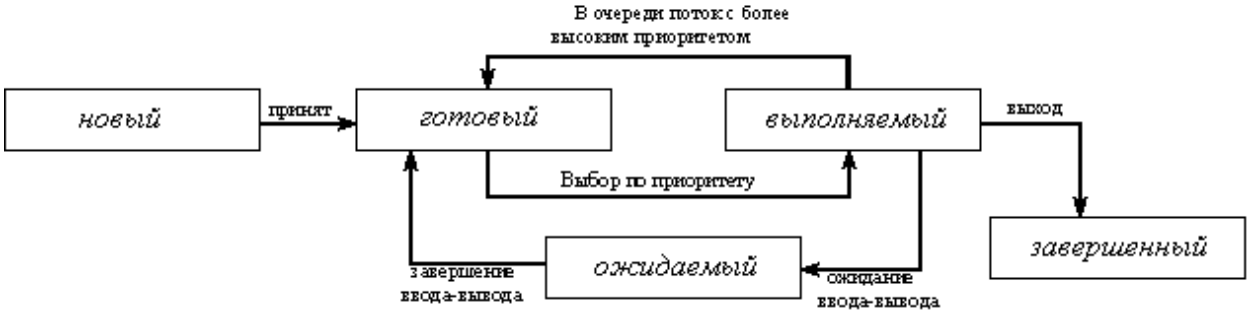


Рисунок 12 – Диаграмма переходов в алгоритме с относительным приоритетом.

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ К ЛАБОРАТОРНЫМ 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 1. 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 2. Конечные автоматы

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

А) Сколькими параметрами определяется конфигурация конечного автомата?

Б) Распознавателями какого типа языков являются конечные автоматы?

В) Как можно описать конечный автомат? Каковы особенности описания КА в каждом случае?

Варианты индивидуального задания

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

1) между вхождениями единиц четное число нулей;
2) за каждым вхождением пары единиц следует нуль;
3) каждый пятый символ единица;
4) все цепочки начинаются на нуль и оканчиваются единицей;
5) в цепочке перед каждой единицей стоит нуль;
6) цепочка должна содержать ровно три единицы;
7) в цепочке перед каждым нулем стоит единица;
8) между вхождениями единиц нечетное число нулей;
9) каждый четвертый символ нуль;
10) цепочка должна содержать не более пяти нулей.

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

Тема 3. Работа с потоками в C#

Среда исполнения *.NET CLR* предоставляет возможность работы с управляемыми потоками через *объект* Thread пространства имен System.Threading. Среда исполнения стремится оптимизировать работу управляемых потоков и использовать для их выполнения потоки процесса, существующие на уровне операционной системы. Поэтому создание потоков типа Thread не всегда сопряжено с созданием потоков процесса.

Основные этапы работы:

```
// Инициализация потока
Thread oneThread = new Thread(Run);
// Запуск потока
oneThread.Start();
// Ожидание завершения потока
oneThread.Join();
```

В качестве рабочего элемента можно использовать *метод класса*, делегат метода или лямбда-выражение. В следующем фрагменте создаются три потока. Первый *поток* в качестве рабочего элемента принимает *статический метод* LocalWorkItem. Второй *поток* инициализируется с помощью лямбда-выражения, третий *поток* связывается с

методом общедоступного класса.

```
class Program
{
 static void LocalWorkItem()
 {
  Console.WriteLine("Hello from static method");
 }
 static void Main()
 {
  Thread thr1 = new Thread(LocalWorkItem);
  thr1.Start();
  Thread thr2 = new Thread(() =>
   {
    Console.WriteLine("Hello from
          lambda-expression");
   });
  thr2.Start();
   ThreadClass thrClass = new  ThreadClass("Hello from thread-class");
   Thread thr3 = new Thread(thrClass.Run);
   thr3.Start();
  }
}
class ThreadClass
{
  private string greeting;
  public ThreadClass(string sGreeting)
  {
   greeting = sGreeting;
  }
  public void Run()
  {
   Console.WriteLine(greeting);
  }
}
```

Вызов метода thr1.Join() блокирует основной *поток* до завершения работы потока thr1.

```
Thread thr1 = new Thread(() =>
  {
   for(int i=0; i<5; i++)
    Console.Write("A");
  });
Thread thr2 = new Thread(() =>
  {
   for(int i=0; i<5; i++)
    Console.Write("B");
  });
Thread thr3 = new Thread(() =>
  {
   for(int i=0; i<5; i++)
    Console.Write("C");
  });
```

```
thr1.Start();
thr2.Start();
thr1.Join();
thr2.Join();
thr3.Start();
```

В общем случае порядок вывода первого и второго потоков не определен. *Вывод* третьего потока осуществляется только после завершения работы потоков thr1 и thr2.
Можно получить результат: AAAAABBBBBCCCCC
или

BBBBBAAAAACCCCC

Маловероятны, но возможны варианты: AABBAAABCCCCC

или AAAABBBBBACCCCC

Передача параметров

Общение с потоком (*передача параметров*, возвращение результатов) можно реализовать с помощью глобальных переменных.

```
class Program
{
  static long Factorial(long n)
  {
    long res = 1;
    do
    {
      res = res * n;
    } while(--n > 0);
    return res;
  }
  static void Main()
  {
    long res1, res2;
    long n1 = 5000, n2 = 10000;
    Thread t1 = new Thread(() =>
      {
        res1 = Factorial(n1))
      });
    Thread t2 = new Thread(() => { res2=Factorial(n2); });
    // Запускаем потоки
    t1.Start();  t2.Start();
    // Ожидаем завершения потоков
    t1.Join();  t2.Join();
    Console.WriteLine("Factorial of {0} equals {1}",
              n1, res1);
    Console.WriteLine("Factorial of {0} equals {1}",
              n2, res2);
  }
}
```

Существует возможность передать параметры в рабочий метод потока с помощью перегрузки метода Start. *Сигнатура* рабочего метода строго фиксирована – либо без аргументов, либо только один *аргумент* типа object. Поэтому при необходимости передачи нескольких параметров в рабочем методе необходимо выполнить правильные преобразования.

```
class Program
{
 static double res;
 static void ThreadWork(object state)
 {
  string sTitle = ((object[])state)[0] as string;
  double d = (double)(((object[])state)[1]);
  Console.WriteLine(sTitle);
  res = SomeMathOperation(d);
 }
 static void Main()
 {
  Thread thr1 = new Thread(ThreadWork);
  thr1.Start(new object[] {"Thread #1", 3.14});
  thr1.Join();
  Console.WriteLine("Result: {0}", res);
    }
}
```

Приостановление потока

Метод Sleep() позволяет приостановить выполнение текущего потока на заданное число миллисекунд:

```
// Приостанавливаем поток на 100 мс
Thread.Sleep(100);
// Приостанавливаем поток на 5 мин
Thread.Sleep(TimeSpan.FromMinute(5));
```

Если в качестве аргумента указывается ноль Thread.Sleep(0), то выполняющийся *поток* отдает выделенный квант времени и без ожидания включается в конкуренцию за *процессорное время*. Такой прием может быть полезен в отладочных целях для обеспечения параллельности выполнения определенных фрагментов кода.

Например, следующий фрагмент

```
static void ThreadFunc(object o)
{
 for(int i=0; i<20; i++)
  Console.Write(o);
}
static void Main()
{
 Thread[] t = new Thread[4];
 for(int i=0; i<4; i++)
  t[i] = new Thread(ThreadFunc);

 t[0].Start("A"); t[1].Start("B");
 t[2].Start("C"); t[3].Start("D");
```

```
 for(int i=0; i<4; i++)
   t[i].Join();
}
```

Вывод на *консоль*:

BBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAACCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDDDDDD

Параллельность не наблюдается, так как каждый *поток* за выделенный *квант процессорного времени* успевает обработать все 20 итераций. Изменим *тело цикла* рабочей функции:

```
static void ThreadFunc(object o)
{
  for(int i=0; i<20; i++)
  {
    Console.Write(o);
    Thread.Sleep(0);
  }
}
```

Вывод стал более разнообразный:

AAAACACACACACACACACACACACACACACACAACCBBDDDBDBDBDBDBDBDBDBDBDBDBDBCCCDBBBDDBBDD

Существует аналог метода Thread.Sleep(0), который позволяет вернуть выделенный квант – Thread.Yield(). При этом возврат осуществляется только в том случае, если для ядра, на котором выполняется данный *поток*, есть другой готовый к выполнению *поток*. Неосторожное применение методов Thread.Sleep(0) и Thread.Yield() может привести к ухудшению быстродействия из-за не оптимального использования кэш-памяти.

Задание

1. Выполненные представленные фрагменты программ в среде Visual Studio.
2. Представьте отчет с результатами тестирования программы

Тема 4 Свойства и приоритеты потока

Каждый *поток* имеет ряд свойств: Name – имя потока, ManagedThreadId – номер потока, IsAlive – признак существования потока, IsBackground – признак фонового потока, ThreadState – состояние потока. Эти свойства доступны и для внешнего вызова.

```
// Объявляем массив потоков
Thread[] arThr = new Thread[N];
for(int i=0; i<arThr.Length; i++)
{
  arThr[i] = new Thread(SomeFunc);
  arThr[i].Start();
}
for(int i=0; i<arThr.Length; i++)
```

```
    {
     // Выводим информацию о потоках
     Console.WriteLine("Thread Id: {0},
        name: {1}, IsAlive: {2}",
          arThr[i].ManagedThreadId,
          arThr[i].Name,
          arThr[i].IsAlive);
    }
```

Свойства текущего потока можно получить с помощью объекта Thread.CurrentThread. Следующий фрагмент с помощью механизма рефлексии выводит все свойства текущего потока.

```
using System;
using System.Reflection;
using System.Threading;
class ThreadInfo
{
 static void Main()
 {
  Thread t = Thread.CurrentThread;
  t.Name = "MAIN THREAD";
  foreach(PropertyInfo p in t.GetType().GetProperties())
  {
   Console.WriteLine("{0}:{1}",
      p.Name,p.GetValue(t, null));
  }
    }
}
```

Вывод всех свойств текущего потока:

```
ManagedThreadId: 10
ExecutionContext: System.Threading.ExecutionContext
Priority: Normal
IsAlive: True
IsThreadPoolThread: False
CurrentThread: System.Threading.Thread
IsBackground: False
ThreadState: Running
ApartmentState: MTA
CurrentUICulture: ru-RU
CurrentCulture: ru-RU
CurrentContext: ContextID: 0
CurrentPrincipal: System.Security.Principal.GenericPrincipal
Name: MAIN THREAD
```

Приоритеты потоков

Приоритеты потоков определяют очередность выделения доступа к ЦП. Высокоприоритетные потоки имеют преимущество и чаще получают *доступ* к ЦП, чем низкоприоритетные. Приоритеты потоков задаются перечислением ThreadPriority, которое имеет пять значений: Highest – наивысший, AboveNormal – выше среднего, Normal - средний, BelowNormal – ниже среднего, Lowest – низший. *По* умолчанию *поток* имеет средний приоритет. Для изменения приоритета потока или чтения текущего используется

свойство Priority.Влияние приоритетов сказывается только в случае конкуренции *множества* потоков за мощности ЦП.

В следующем фрагменте пять потоков с разными приоритетами конкурируют за *доступ* к ЦП с двумя ядрами. Каждый *поток* увеличивает свой *счетчик*.

```
class PriorityTesting
{
 static long[] counts;
 static bool finish;
 static void ThreadFunc(object iThread)
 {
  while(true)
  {
   if(finish)
     break;
   counts[(int)iThread]++;
  }
 }
 static void Main()
 {
  counts = new long[5];
  Thread[] t = new Thread[5];
  for(int i=0; i<t.Length; i++)
  {
   t[i] = new Thread(ThreadFunc);
   t[i].Priority = (ThreadPriority)i;
  }
  // Запускаем потоки
  for(int i=0; i<t.Length; i++)
   t[i].Start(i);

  // Даём потокам возможность поработать 10 с
  Thread.Sleep(10000);

  // Сигнал о завершении
  finish = true;

  // Ожидаем завершения всех потоков
  for(int i=0; i<t.Length; i++)
   t[i].Join();
  // Вывод результатов
  for(int i=0; i<t.Length; i++)
    Console.WriteLine("Thread with priority {0, 15},
    Counts: {0}", (ThreadPriority)i, counts[i]);
 }
}
```

Вывод программы

```
Thread with priority       Lowest, Counts:    7608195
Thread with priority    BelowNormal, Counts:   10457706
Thread with priority       Normal, Counts:   17852629
Thread with priority    AboveNormal, Counts:  297729812
```

Thread with priority Highest, Counts: 302506232

Задание

1. Выполненные представленные фрагменты программ в среде Visual Studio.
2. Представьте отчет с результатами тестирования программы

Варианты индивидуального задания по темам 3, 4.

1. Реализуйте последовательную обработку элементов вектора, например, умножение(деление) элементов вектора на число. Число элементов вектора задается параметром.

2. Реализуйте многопоточную обработку элементов вектора, используя разделение вектора на равное число элементов. Число потоков задается параметром.

3. Выполните анализ эффективности многопоточной обработки при разных параметрах N (10, 100, 1000, 100000) и M (2, 3, 4, 5, 10). Результаты представьте в табличной форме.

4. Выполните анализ эффективности при усложнении обработки каждого элемента вектора.

5. Исследуйте эффективность разделения по диапазону при неравномерной вычислительной сложности обработки элементов вектора.

6. Исследуйте эффективность параллелизма при круговом разделении элементов вектора. Сравните с эффективностью разделения по диапазону.

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ К ПРАКТИЧЕСКИМ ЗАНЯТИЯМ

Тема 1 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: $\alpha \to \beta_1$, $\alpha \to \beta_2$, ... $\alpha \to \beta_n$. Тогда эти правила объединяют вместе и записывают в виде: $\alpha \to \beta_1 | \beta_2 | \ldots | \beta_n$. Одной строке в такой записи соответствует сразу n правил.

Такую форму записи правил грамматики называют формой Бэкуса-Наура. Форма Бэкуса-Наура предусматривает, как правило, также, что нетерминальные символы берутся

в угловые скобки: < >.

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: $\alpha_1 A \alpha_2 \rightarrow \alpha 1 \beta \alpha_2$, где $\alpha_1, \alpha_2 \in V^*$, $A \in$ **VN**, $\beta \in \mathbf{V}^+$.

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

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 лю-

бая 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$\rightarrow\beta$, 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$\rightarrow\beta$, 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$\rightarrow$Bγ или A$\rightarrow\gamma$, 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$\rightarrow\gamma$B или A$\rightarrow\gamma$, где A, B$\in$**VN**, $\beta\in$**VT***.

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

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

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$\rightarrow\lambda$», 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.

Контрольные вопросы

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

B→ 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→ + | – | λ
S→ 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

Тема 2. Однозначность и эквивалентность грамматик

Цепочка $\beta = \delta_1\gamma\delta_2$ называется *непосредственно выводимой* из цепочки $\alpha = \delta_1\omega\delta_2$ в грамматике G**(VT,VN, P,** S**), V** = **VN** $\cup$ **VT,** $\delta_1, \gamma, \delta 2 \in \mathbf{V}^*$, $\omega \in \mathbf{V}^+$, Если в грамматике G существует правило: $\omega \rightarrow \gamma \in$ **P**. Непосредственная выводимость цепочки β из цепочки α обозначается: $\alpha \Rightarrow \beta$. Из определения следует, что если взять несколько символов в цепочке α и заменить их на другие символы согласно некоторому правилу грамматики и получить цепочку β, то β непосредственно выводима из α.

Цепочка β называется *выводимой* из цепочки α (обозначается: $\alpha \Rightarrow^* \beta$), в том случае, если выполнено одно из двух условий:

1. β непосредственно выводима из α ($\alpha \Rightarrow \beta$);
2. существует такая γ, что γ выводима из α и β непосредственно выводима из γ ($\alpha \Rightarrow^* \gamma$ и $\gamma \Rightarrow \beta$).

Если цепочка вывода из α к β содержит одну и более промежуточных цепочек, она имеет специальное обозначение α =>+ β. Если количество шагов известно, можно его указать непосредственно у знака выводимости. Например, выражение $\alpha =>^3 \beta$ означает, что β выводится из α за три вывода.

Вывод называется окончательным, если на основе цепочки β, полученной в результате вывода, нельзя больше сделать ни одного шага вывода. Иначе говоря, вывод законченный, если цепочка, полученная в его результате пустая, либо содержит только терминальные символы грамматики.

Язык L, заданный грамматикой G**(VT, VN, P,** S**) –** это множество всех сентенциальных форм грамматики G. Язык L, заданный грамматикой G обозначается как L(G). Алфавитом языка L(G) будет множество терминальных символов грамматики **VT**, поскольку все конечные сентенциальные формы грамматики – цепочки над алфавитом **VT**.

Тогда очевидно, что две эквивалентные грамматики должны иметь, по крайней мере, пересекающиеся множества терминальных символов. (Как правило, эти множества совпадают).

Вывод называется *левосторонним*, если в нем на каждом шаге вывода правило применяется к крайнему левому нетерминальному символу в цепочке. Аналогично, вывод называется *правосторонним*, если в нем на каждом шаге вывода правило применяется к крайнему правому нетерминальному символу в цепочке.

Примером левостороннего вывода является: S => Y => TY => TT => YTT => TTT, а правостороннего S => T => TX => T5 => T6. Вывод F => R является одновременно и лево- и правосторонним.

Встречаются выводы, которые нельзя отнести ни к левосторонним, ни к правосторонним, например TFT => TFFT => TFFF => FFFF => FFTFF.

Для грамматик типов 2 и 3 (КС-грамматик и регулярных грамматик) всегда можно построить левосторонний или правосторонний выводы. Для грамматик других типов это не всегда возможно, так как по структуре их правил не всегда можно выполнить замену крайнего левого или крайнего правого нетерминального символа в цепочке.

Деревом вывода грамматики G**(VT,VN, P,** S**),** называется граф, который соответствует некоторой цепочке вывода и удовлетворяет следующим условиям:

каждая вершина графа обозначается некоторым символом грамматики $A \in \mathbf{VN} \cup \mathbf{VT}$**;**

корнем дерева является вершина, обозначенная целевым символом грамматики S;

листьями дерева являются вершины, обозначенные терминальными символами грамматики или символом пустой цепочки λ,

если некоторый узел дерева обозначен нетерминальным символом $A \in \mathbf{VN}$, а связанные с ним узлы – символами $b, b_2 ... b_n$; $n > 0$, $0 < i < n$, $b_i \in (\mathbf{VN} \cup \mathbf{VT} \cup \{\lambda\})$, то в грамматике G**(VT,VN, P,** S**),** существует правило $A \rightarrow b_1 b_2 \ldots b_n \in P$.

Из определения следует, что по структуре правил дерево вывода в указанном виде всегда можно построить только для грамматик типов 2 и 3 (контекстно-свободных и регулярных). Для грамматик других типов дерево вывода в таком виде можно построить не всегда (либо же оно будет иметь несколько иной вид).

Для того чтобы построить дерево вывода, достаточно иметь только цепочку вывода. Дерево вывода можно построить двумя способами: сверху вниз и снизу вверх. Для строго формализованного построения дерева вывода всегда удобнее пользоваться строго определенным выводом: либо левосторонним, либо правосторонним.

При построении дерева вывода сверху вниз построение начинается с целевого символа грамматики, который помещается в корень дерева. Затем в грамматике выбирается необходимое правило, и на первом шаге вывода корневой символ раскрывается на несколько символов первого уровня. На втором шаге среди всех концевых вершин дерева

выбирается крайняя (крайняя левая – для левостороннего вывода, крайняя правая – для правостороннего) вершина, обозначенная нетерминальным символом, для этой вершины выбирается нужное правило грамматики, и она раскрывается на несколько вершин следующего уровня. Построение дерева заканчивается, когда все концевые вершины обозначены терминальными символами, в противном случае надо вернуться ко второму шагу и продолжить построение.

Построение дерева вывода снизу вверх начинается с листьев дерева. В качестве листьев выбираются терминальные символы конечной цепочки вывода, которые на первом шаге построения образуют последний уровень (слой) дерева. На втором шаге в грамматике выбирается правило, правая часть которого соответствует крайним символам в слое дерева (крайним правым символам при правостороннем выводе и крайним левым при левостороннем). Выбранные вершины слоя соединяются с новой вершиной, которая выбирается из левой части правила. Новая вершина попадает в слой дерева вместо выбранных вершин. Построение дерева закончено, если достигнута корневая вершина (обозначенная целевым символом), а иначе надо вернуться ко второму шагу и повторить его относительно полученного слоя дерева.

Все известные языки программирования имеют нотацию записи «слева направо», компилятор также всегда читает входную программу слева направо (и сверху вниз, если программа разбита на несколько строк). Поэтому для построения дерева вывода методом «сверху вниз», как правило, используется левосторонний вывод, а для построения «снизу вверх» – правосторонний вывод.

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

Рассмотрим некоторую грамматику G ({+, – , *, /, (,), x, y}, {S}, P, S):

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

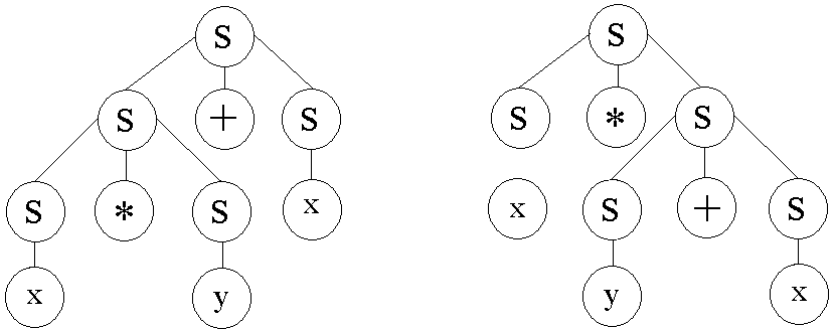
Грамматика определяет язык арифметических выражений с четырьмя основными операциями: сложение, вычитание, умножение, деление и скобками.

Для цепочки, принадлежащей данному языку, x*y+x можно построить два варианта левостороннего вывода:

S => S+S => S*S+S => x*S+S => x*y+S => x*y+x

S => S*S => x*S => x*S+S => x*y+S => x*y+S

Каждому из этих вариантов будет соответствовать свое дерево вывода:



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

К сожалению, доказано, что не существует алгоритма, позволяющего проверить однозначность и эквивалентность грамматик. Однако, неразрешимость проблем эквивалентности и однозначности грамматик в общем случае не означает, что они не разрешимы

вообще. Для многих частных случаев эти проблемы решены. Например, для КС-грамматик существуют правила определенного вида, по наличию которых во всем множестве правил грамматики G**(VT, VN, P,** S**)** можно утверждать, что она является неоднозначной. Эти группы правил имеют следующий вид:

1) $A \rightarrow AA \mid \alpha$
2) $A \rightarrow A\alpha A \mid \beta$
3) $A \rightarrow \alpha A \mid A\beta \mid \gamma$
4) $A \rightarrow \alpha A \mid \alpha A\beta A \mid \gamma$

здесь $A \in \mathbf{VN}$; $\alpha, \beta, \gamma \in (\mathbf{VN} \cup \mathbf{VT})^*$.

Если в КС-грамматике встречается хотя бы одно правило любого из приведенных вариантов, то доказано, что такая грамматика точно будет неоднозначной. Однако если подобных правил во всем множестве правил грамматики нет, это совсем не означает, что грамматика является однозначной. То есть отсутствие правил указанного вида – необходимое, но не достаточное условие однозначности грамматики.

Существуют условия, при выполнении которых грамматика заведомо является однозначной. Они справедливы для всех регулярных и многих классов контекстно-свободных грамматик. Эти условия, напротив, являются достаточными, но не необходимыми для однозначности грамматик.

Контрольные вопросы

1. Что такое сентенциальная форма грамматики?
2. В чем заключается отличие левосторонних и правосторонних выводов?
3. В чем заключается однозначность грамматики?
4. Дайте определение полностью выводимой цепочки.
5. По каким правилам строится дерево вывода?
6. Выберите неверное утверждение:

а) однозначность – это свойство грамматики;

б) построение дерева вывода заканчивается, когда все концевые вершины обозначены терминальными символами;

в) проблема эквивалентности грамматик разрешима алгоритмически;

г) для того чтобы построить дерево вывода достаточно иметь только цепочку вывода.

Задание

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

P: E → if b then E else E; | if b then E; | a

Не строя цепочек вывода, показать, что грамматика является неоднозначной. Проверить это, построив некоторую цепочку вывода.

2. 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

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

Тема 3. 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, $\delta(a,q)=R$, $a \in V$, $q \in Q$, $R \subseteq Q$,

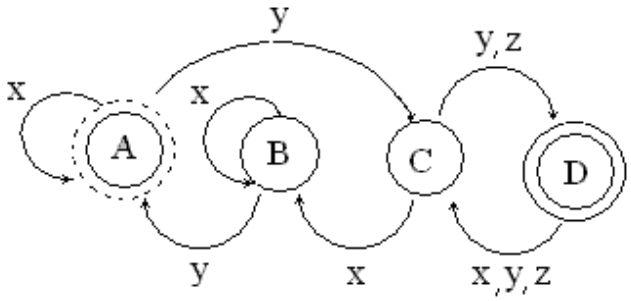
q_0 – начальное состояние автомата ($q_0 \in Q$),

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

Для реализации конечного состояния построить граф или таблицу переходов.

2. Построить регулярное выражение для описания переменных языка C++.
3. Привести к автоматному виду простейшую регулярную грамматику: G({"a","(","*",")","{","}"},{S,C,K},P,S) (символы a,(,*,),{,} из множества терминальных символов грамматики взяты в кавычки, чтобы выделить их среди фигурных скобок, обозначающих само множество):

P: S → C*) | K}

C → (* | Ca | C{ | C} | C(| C* | C) K → { | Ka | K(| K* | K) | K{

Если предположить, что a здесь — это любой алфавитно-цифровой символ, кроме символов(,*,),{,}, то эта грамматика описывает два типа комментариев, допустимых в языке программирования Паскаль.

Тема 4. 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 5 Работа с потоками

В системах с общей памятью, включая многоядерные архитектуры, параллельные вычисления могут выполняться как при многопроцессном выполнении, так и при многопоточном выполнении. Многопроцессное выполнение подразумевает оформление каждой подзадачи в виде отдельной программы (процесса). Недостатком такого подхода является сложность взаимодействия подзадач. Каждый процесс функционирует в своем виртуальном адресном пространстве, не пересекающемся с адресным пространством другого процесса. Для взаимодействия подзадач необходимо использовать специальные средства межпроцессной коммуникации (интерфейсы передачи сообщений, общие файлы, объекты ядра операционной системы).

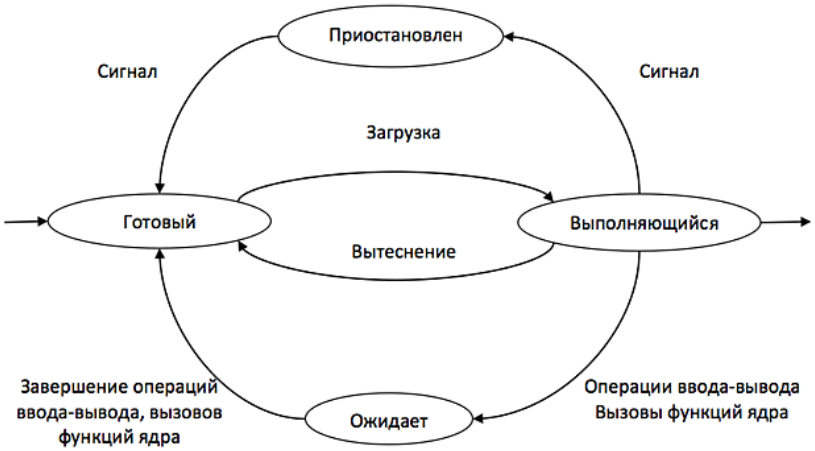
Потоки позволяют выделить подзадачи в рамках одного процесса. Все потоки одного приложения работают в рамках одного адресного процесса. Для взаимодействия потоков не нужно применять какие-либо средства коммуникации. Потоки могут непосредственно обращаться к общим переменным, которые изменяют другие потоки. Работа с общими переменными приводит к необходимости использования средств синхронизации, регулирующими порядок работы потоков с данными.

Потоки являются более легковесной структурой по сравнению с процессами ("облегченные" процессы). Поэтому параллельная работа множества потоков, решающих общую задачу, более эффективна в плане временных затрат, чем параллельная работа *множества* процессов.

Поток состоит из нескольких структур. Ядро потока – содержит информацию о текущем состоянии потока: приоритет потока, программный и стековый указатели. Программный и стековые указатели образуют *контекст* потока и позволяют восстановить выполнение потока на процессоре. Блок окружения потока - содержит заголовок цепочки обработки исключений, локальное хранилище данных для потока и некоторые структуры данных, используемые интерфейсом графических устройств (GDI) и графикой OpenGL. Стек пользовательского режима – используется для передаваемых в методы локальных переменных и аргументов. *Стек* режима ядра - используется, когда код приложения передает аргументы в функцию операционной системы, находящуюся в режиме ядра. *Ядро* ОС вызывает собственные методы и использует стек режима ядра для передачи локальных аргументов, а также для сохранения локальных переменных.

Каждый *поток* может находиться в одном из нескольких состояний. *Поток*, готовый к выполнению и ожидающий предоставления доступа к центральному процессору, находится в состоянии "Готовый". *Поток*, который выполняется в текущий момент времени, имеет статус "Выполняющийся". При выполнении операций ввода-вывода или обращений к функциям ядра операционной системы, *поток* снимается с процессора и нахо-

дится в состоянии "Ожидает". При завершении операций ввода-вывода или возврате из функций ядра *поток* помещается в *очередь* готовых потоков. При переключении контекста *поток* выгружается и помещается в *очередь* готовых потоков.



Переключение контекста

1. Значения регистров процессора для исполняющегося в данный момент потока сохраняются в структуре контекста, которая располагается в ядре потока.
2. Из набора имеющихся потоков выделяется тот, которому будет передано управление. Если выбранный поток принадлежит другому процессу, Windows переключает для процессора виртуальное адресное пространство.
3. Значения из выбранной структуры контекста потока загружаются в регистры процессора.

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 экзамена.

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

Экзамен проводится по билетам. Билет включает два теоретических вопроса. 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 на экзамене учитывается: полнота ответа на поставленный вопрос, точ-

ность формулировок, логичность ответа, умение делать выводы, выявлять закономерности, соблюдение норм литературной речи и использования специализированной терминологии. Высшего балла заслуживает ответ, удовлетворяющий всем этим требованиям.

Перечень вопросов для самостоятельного изучения:

1. Автоматы с магазинной памятью (определение, формы записи, классификация).
2. Методы генерация кода
3. Основные методы оптимизации кода

ЛИТЕРАТУРА

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

2 Гордеев, А.В. Операционные системы. Допущено Министерством образования РФ) – СПб: Питер, 2009. – 416с.

2 Гунько А.В. Системное программное обеспечение [Электронный ресурс]: конспект лекций/ Гунько А.В.— Электрон. текстовые данные. – Новосибирск: Новосибирский государственный технический университет, 2011. – 138 с. – Режим доступа: http://www.iprbookshop.ru/45020. – ЭБС «IPRbooks»

3 Малявко А.А. Системное программное обеспечение. Формальные языки и методы трансляции. Часть 1 [Электронный ресурс]: учебное пособие/ Малявко А.А. – Электрон. текстовые данные. – Новосибирск: Новосибирский государственный технический университет, 2010. – 104 с. – Режим доступа: http://www.iprbookshop.ru/45017. – ЭБС «IPRbooks»

4 Малявко А.А. Системное программное обеспечение. Формальные языки и методы трансляции. Часть 2 [Электронный ресурс]: учебное пособие/ Малявко А.А. – Электрон. текстовые данные. – Новосибирск: Новосибирский государственный технический университет, 2011. – 160 с. – Режим доступа: http://www.iprbookshop.ru/45018. – ЭБС «IPRbooks»

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

6 Олифер, В.Г. Сетевые операционные системы / В.Г. Олифер, Н.А. Олифер СПб.: Питер, 2010. – 539 с.

СОДЕРЖАНИЕ