Московский Государственный Университет
имени М. В. Ломоносова

Факультет вычислительной математики и кибернетики

Л. Е. Карпов

Системы программирования

Учебное пособие

Москва
2009

Печатается по решению Редакционно-издательского совета факультета вычислительной математики и кибернетики МГУ им. М. В. Ломоносова

Рецензенты
д.ф.-м.н.
д.ф.-м.н.

Карпов Л. Е.

К **Системы программирования:** Учебное пособие. – М.: Издательский отдел факультета ВМК МГУ (лицензия ИД № 05899 от 24.09.2001), 2009 – XXXXXX с.: ил.
ISBN

Настоящее пособие является дополнением к ранее выпущенному несколькими изданиями пособия по курсу “Системы программирования”, который читается на факультете ВМ и К МГУ им. М. В. Ломоносова с середины 1990-х годов. В течение всего времени в курсе излагались основы построения систем программирования и тенденции их развития, однако, в ранее выпущенных пособиях эти темы, давшие название всему курсу, отражения не получили. Настоящее пособие восполняет этот пробел.

В основе курса – изучение комплексной системы программирования, обеспечивающей поддержку всего жизненного цикла программных продуктов, начиная от их проектирования и заканчивая их сопровождением в процессе эксплуатации.

УДК
ББК

ISBN

Оглавление

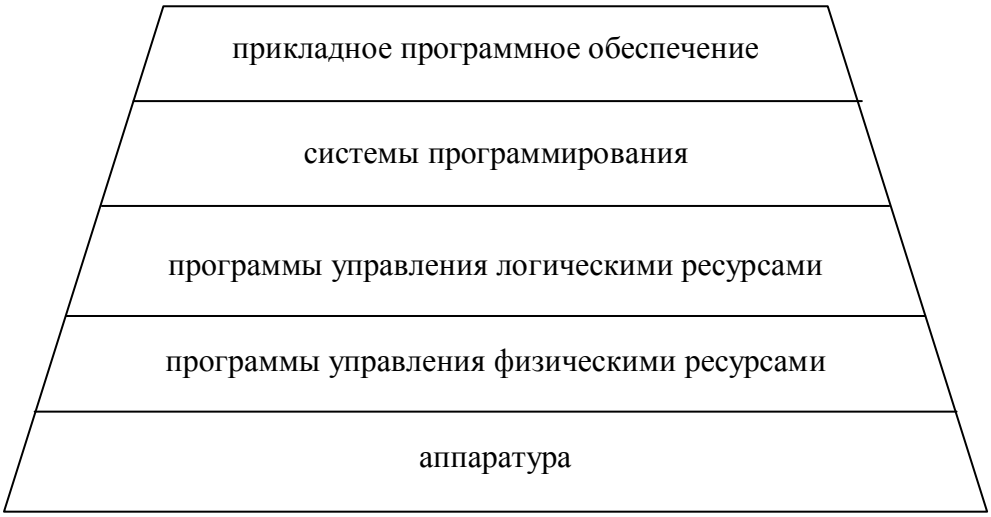
1. Жизненный цикл программного продукта 5
   1.1. Этапы жизненного цикла 6
   1.2. Основные требования к системам программирования 11
   1.3. Основные компоненты систем программирования 12
2. Классическая система программирования 16
   2.1. Общая схема работы систем программирования 17
   2.2. Интегрированная среда разработки 17
3. Компоненты классической системы программирования 21
   3.1. Редакторы текстов 21
      3.1.1. Виды текстовых редакторов 21
      3.1.2. Лексический анализ “на лету” 23
   3.2. Трансляторы, компиляторы, интерпретаторы 24
      3.2.1. Схемы работы трансляторов 25
      3.2.2. Смешанная стратегия трансляции 26
   3.3. Компилятор, как основной компонент системы программирования 27
      3.3.1. Общая схема работы компилятора 27
         3.3.1.1. Основные компоненты компилятора и фазы компиляции 27
         3.3.1.2. Однопроходный компилятор 30
      3.3.2. Задачи семантического анализа 33
         3.3.2.1. Проверка контекстных условий 34
         3.3.2.2. Дополнение внутреннего представления 35
         3.3.2.3. Проверка правил программирования 35
         3.3.2.4 Разнесение имен по пространствам именования 36
      3.3.3. Внутреннее представление программ 37
      3.3.4. Оптимизация в компиляторах 41
         3.3.4.1. Машинно-независимая оптимизация 43
         3.3.4.2. Машинно-зависимая оптимизация 48
      3.3.5. Основные методы динамического распределения памяти 51
      3.3.6. Генерация кода 57
   3.4. Редакторы связей: назначение, принципы работы 60
   3.5. Загрузчики: основные функции, принципы работы 62
   3.6. Техника работы с библиотеками 63
      3.6.1. Статические библиотеки 63
      3.6.2. Динамически загружаемые библиотеки 64
      3.6.3. Основные типы библиотек 66
         3.6.3.1. Библиотеки функций, процедур и макроопределений 67
         3.6.3.2. Библиотеки классов 68
         3.6.3.3. Библиотеки компонентов 69

3.6.3.4. Критерии проектирования стандартных библиотек 69
3.7. Средства конфигурирования 72
3.8. Системы управления версиями программных комплексов 73
3.9. Средства отладки и тестирования программ 75
3.10. Профилировщики 79
3.11. Справочные системы 80
4. Краткий обзор современных систем программирования 82
4.1. Компонентный подход и визуальное программирование 82
4.2. Системы программирования компании Borland 83
4.2.1. Turbo Pascal 83
4.2.2. Delphi 85
4.2.3. C++ Builder 88
4.3. Системы программирования компании Microsoft 88
4.3.1. Visual Basic 89
4.3.2. VBA 90
4.3.3. Visual C++ 90
4.3.4. Концепция .NET и C# 91
4.4. Системы программирования ОС UNIX и Linux 93
4.5. Проект GNU 95
4.6. Системы программирования компании IBM 98
4.6.1. Комплексная система программирования Rational Software 98
4.6.2. Интегрированная среда разработки Eclipse 99
4.6.3. Системы программирования ЭВМ zSeries 100
5. Разработка распределенных программ 104
5.1. Системы клиент-сервер 104
5.2. Технологии COM/DCOM 107
5.3. Брокеры объектов CORBA 109
5.4. Серверы приложений и сетевые службы 112
6. Средства автоматического грамматического разбора 115
6.1. Построение лексических анализаторов по регулярным выражениям 115
6.2. Автоматизация построения синтаксических анализаторов 119
Литература 123

Автор выражает глубокую признательность Ирине Анатольевне Волковой и Тамаре Васильевне Руденко, советы которых помогли ему при подготовке настоящего пособия.

1. Жизненный цикл программного продукта

В иерархии программно-аппаратного обеспечения системам программирования отводится место между программами управления логическими ресурсами и прикладным программным обеспечением:



Определение: ***системой программирования называется комплекс программных средств, предназначенных для поддержки программного продукта на протяжении всего жизненного цикла этого продукта***.

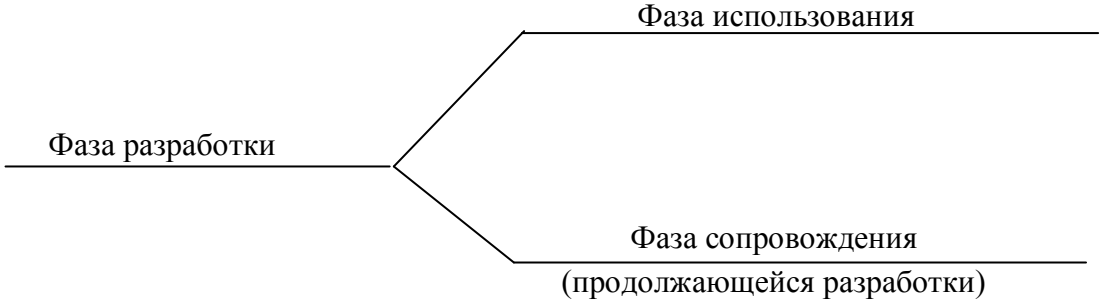
Для обозначения результатов труда программистов обычно использование следующих терминов: *программа*, *программный продукт* и *системный (или интегрированный) программный продукт*. ***Программа*** создается для решения отдельной задачи автором программы и используется в некоторой конкретной операционной среде. Программа неотделима от ее автора. ***Программным продуктом*** называется такая программа, которая работает без авторского присутствия в рамках некоторого набора операционных сред. Программный продукт может исполняться, тестироваться и модифицироваться без участия автора (он отчужден от автора). Качество программного продукта должно быть существенно выше качества обычной программы. Для программных продуктов разрабатывается документация, необходимая для пользователей, чтобы они могли работать с программным продуктом в целях решения собственных задач, и разработчиков, модифицирующих продукт. Программный продукт должен быть *настраиваемым*, причем эта настройка должна выполняться путем задания некоторых параметров настройки.

Системный (интегрированный) программный продукт есть комплекс программных продуктов (пакет). Примером интегрированного программного продукта может служить пакет Microsoft Office, включающий в себя около десятка программных продуктов, обладающих согласованными интерфейсами. Способы задания параметров, режимов работы и действий пользователя во всех компонентах одинаковы или похожи, хотя каждый компонент обладает собственными специализированными средствами. Между компонентами легко организовать передачу данных. Например, подготовив

сложную электронную таблицу (Excel), ее легко презентовать в наглядном виде (PowerPoint).

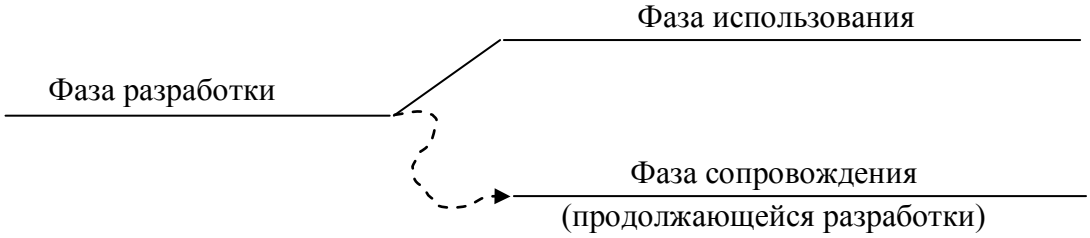
1.1. Этапы жизненного цикла

Работа с программой и над программой (а значит и использование системы программирования) продолжается на протяжении всей жизни программ, которая у любой из них состоит из трех фаз – *фазы разработки*, *фазы использования* и *фазы сопровождения*:

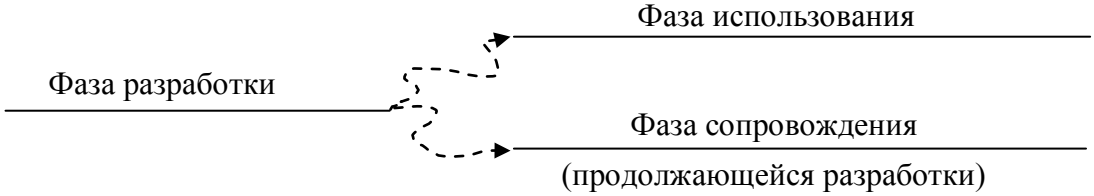


Разработка может вестись коллективом разработчиков новых продуктов. Сопровождение может передаваться другому коллективу, может быть другой организации. Использование программных продуктов часто ведется совсем другими людьми. Фаза разработки предшествует двум другим фазам, которые проходят во времени параллельно друг другу. Для больших и длительно используемых программ фазу сопровождения иногда называют *фазой продолжающейся разработки*. Необходимость сопровождения объясняется двумя причинами. Во-первых, в большой программе всегда имеется некоторое количество ошибок, которые не выявляются при тестировании. Во-вторых, программа должна развиваться. Появляются новые потребности, пожелания, к вычислительным машинам подключаются новые виды внешних устройств, с которыми программа должна научиться взаимодействовать.

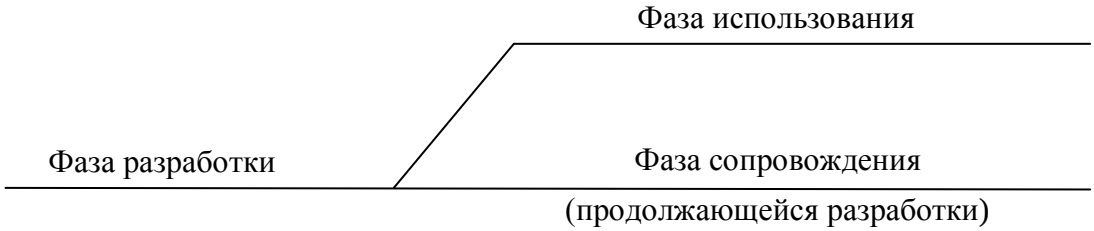
Неправильный подход к организации процесса разработки, применение неподходящих для этого средств разработки и систем программирования может привести к искажению жизненного цикла получающихся программ, к возникновению патологического цикла с разрывом между разработкой и сопровождением:



Такого рода процессы часто возникают, если разработчики программного обеспечения пренебрегают правилами и стандартами разработки. С этим можно было бы примириться, если бы разрывы между разработкой и сопровождением не приводили к разрыву между разработкой и использованием:



Приведенные схемы соответствуют, так называемым, одноразовым разработкам. Гораздо интереснее и продуктивнее такие отступления от классической модели жизненного цикла, в которых фаза сопровождения становится непосредственным продолжением разработки, при этом разработчик сам сопровождает свои программы:



Процессы разработки и сопровождения включают в себя этапы:

- *Анализ (определение) требований*
- *Проектирование*
- *Написание текста программ (программирование, "кодирование")*
- *Компоновка или интеграция программного комплекса*
- *Верификация, тестирование и отладка*
- *Документирование*
- *Внедрение*
- *Тиражирование*
- *Сопровождение, повторяющее все предыдущие этапы*

Этап ***постановки задачи и определения и анализа требований*** во многом не формализован, но он влияет на всю разработку и качество конечного продукта. На этом этапе необходимо выяснить потребности конечного пользователя. Часто для этого приходится создавать общий с заказчиком словарь терминов – систему понятий, посредством которой можно будет общаться с пользователями. Выработанная система понятий должна использоваться для описания объектов автоматизации, их сходства с другими объектами и их своеобразия, то есть отличий от объектов, остающихся за рамками осуществляемого проекта разработки программного обеспечения.

На первом этапе создаются материалы различных видов: от простого текста до частично формализованных описаний требований. Чтобы добиться хотя бы частичной формализации, разрабатываются специальные программные средства, в частности, языки описания требований. Эти языки могут быть разными:

- Таблицы решений, отображающие связь входных данных с выходными. Фактически таблицы содержат условия, налагаемые на возможные входные данные, и эффекты, которые эти данные оказывают на поведение программы и выходные данные. Для их построения входные данные разбиваются на группы, представители которых обрабатываются программным продуктом практически одинаково.
- Функциональные диаграммы, представляющие собой графы с узлами, соответствующими входным данным и накладываемым на них (или их сочетания) условиями и/или ограничениями. В диаграммах также описывается эффект обработки соответствующих данных.
- Языки спецификаций, применяемые для формулирования требований (язык CLU – один из первых таких языков, к этим же языкам относится язык диаграмм взаимодействия MSC, язык SDL и другие).

Результатом первого этапа являются сформулированные требования, то есть внешняя спецификация, *описание системы с точки зрения пользователя*.

Проектирование есть сложный процесс проектирования всей системы в целом. Часто этот этап разбивается на два подэтапа: проектирование структуры системы (проектирование "*в большом*"), и проектирование совокупности взаимосвязанных подсистем (проектирование "*в малом*"). На этапе проектирования осуществляется процесс, который называют *управлением сложностью*.

Как и все сложные системы, сложные программы имеют иерархическую структуру, а уровни их иерархии отражают различные уровни абстракции, следующие друг из друга, но не тождественные друг другу. Эти уровни соответствуют взаимозависимым подсистемам, которые сами могут иметь сложную, иерархическую структуру. На этапе проектирования широко применяется метод "*разделяй и властвуй*". При проектировании сложных программных комплексов их обычно стараются разложить на некоторое число относительно небольших подсистем, каждую из которых можно отладить независимо от других ("автономно"). Основная сложность проектирования – определить основания для разделения общей системы на составляющие ее подсистемы. Выбор подсистем часто зависит от разработчика и существенно влияет как на сложность процесса проектирования, так и на конечное качество программного продукта. Такое разложение называется *декомпозицией* программы.

Декомпозицию одной и той же системы можно проводить, основываясь на разных ее свойствах. *Алгоритмическая декомпозиция* разделяет программную систему по алгоритмам, в ней используемым. Такой метод декомпозиции ассоциируется со структурным программированием и методом проектирования "сверху-вниз". Каждый отдельный полученный в результате модуль системы выполняет какой-либо один этапов работы общего системного процесса. Данные в таком подходе играют пассивную роль.

Объектно-ориентированная декомпозиция связана с выделением отдельных объектов, которые способны воспринимать направляемые им сообщения и отвечать выполнением тех или иных ответных действий. Алгоритмы в такой модели теряют свою независимость, они превращаются в операции над выделенными объектами. Активную роль в таком подходе приобретают данные программ, то есть объекты. Эти объекты обладают не только информационной составляющей (значениями), но и действиями.

Если алгоритмическая декомпозиция концентрируется на последовательности происходящих событий, то декомпозиция по объектам фиксируется на этих объектах и на событиях, происходящих в системе и вызывающих действия. Сам процесс, выполняемый в программе, описывается в терминах посылки сообщений от объекта другим объектам. В последнее время объектно-ориентированный подход рассматривается, как более предпочтительный. Объектно-ориентированные системы более открыты и легче поддаются модернизации. На основе объектно-ориентированной декомпозиции сложные программные системы удается строить путем эволюционного развития небольших подсистем.

Результатом работы второго этапа являются

- схема иерархии подсистем (или модулей, для алгоритмической декомпозиции),

- функциональность и интерфейсы каждой подсистемы (первые два результата возникают от проектирования “в большом”), то есть их внешние спецификации,
- внутреннее представление (структура) данных для каждого отдельного модуля программы,
- алгоритмы, обрабатывающие данные (последние два результата – от проектирования “в малом”) в каждом отдельном модуле программы.

Написание текста чаще называется программированием или кодированием, этот этап заканчивается, когда в наличии оказывается текст программы на некотором языке программирования. Для написания текста программ чаще всего выбирается язык, соответствующий методам, выбиравшимся при анализе и формировании требований к программному продукту и проведении декомпозиции строящейся системы. В частности, объектно-ориентированная декомпозиция особенно полезна при последующем программировании на объектно-ориентированном языке.

Верификацией программы называется процесс ее проверки на правильность. Существуют специальные теоретические методы верификации программ, но наиболее широко применяются эвристические методы верификации, основанные на тестировании. ***Тестирование программы*** есть процесс обнаружения дефектов в созданных программах, а ***отладка программ*** есть выявление причин дефектов, а также их устранение. Во время тестирования выявляется несоответствие программы исходным требованиям и спецификациям по тестам, то есть программам с заранее известными ответами. Среди методов наиболее известны методы “черного” и “белого” ящиков. Отладка начинается после обнаружения дефектов. Иногда на этапе отладки приходится устранять не только причины дефектов, но и дефекты в данных, возникшие при работе неверных программ.

Результатом тестирования и отладки является доказательство, что все выявленные на данном комплекте тестов ошибки исправлены, а других ошибок не обнаружено.

Этап ***компоновки*** представляет собой интеграционный процесс комплексирования (комбинирования), то есть связывания отдельных частей программы в одну большую систему программного обеспечения. Обычно после компоновки проводится этап ***комплексного тестирования*** программных систем, на котором проверяется правильность взаимодействия всех автономно разработанных составных частей между собой. Правильная организация процессов верификации, тестирования и *документирования* имеет особое значение для последующего *внедрения*, *тиражирования* и *сопровождения* создаваемых программ.

Внедрением называется работа по привлечению заказчика к использованию созданного программного продукта. На этом этапе возникает множество организационных проблем по обучению пользователей, тестированию работоспособности и устойчивости программы при работе в конкретной организации и конкретной операционной среде. Если организация заказчика представляет собой разветвленную сеть рабочих мест, проводится ***тиражирование*** продукта.

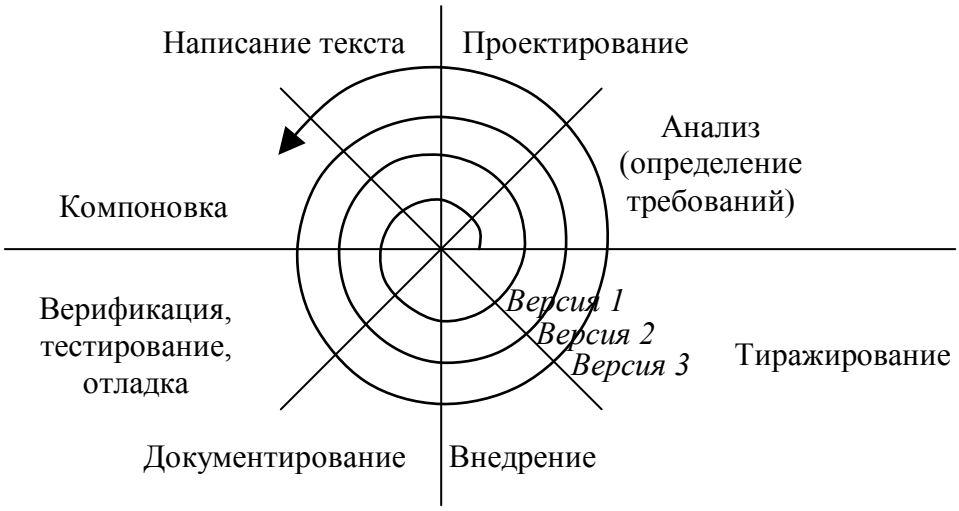
Во время ***сопровождения*** программного продукта можно наблюдать продолжение контактов пользователей и разработчиков. Сопровождение является зеркальным отражением разработки и внутри себя содержит все этапы фазы разработки (от определения требований до документирования).

В идеальном случае процесс разработки может иметь вид последовательно выполняемых слабо связанных между собой этапов. Такая схема разработки называется *нисходящей*, и она редко встречается на практике. Чаще встречается *каскадная* схема, в которой можно наблюдать замыкание процесса и проведение повторного анализа требований после проведения его тестирования. Каскадная схема является одним из вариантов *итеративных* схем разработки программного обеспечения.

Буквальное следование каскадной схеме разработки приводит к существенному запаздыванию получения результатов. Согласование результатов с пользователями производится только в точках, планируемых после завершения каждого этапа работ, требования к программному обеспечению зафиксированы в техническом задании на все время разработки. Это приводит к тому, что пользователи могут вносить замечания и пожелания по совершенствованию системы только после того, как работа над ней будет полностью завершена. Каскадная схема хорошо пригодна для моделирования процессов разработки такого программного обеспечения, для которого с самого начала удается достаточно точно и полно сформулировать все требования, с тем, чтобы затем предоставить разработчикам свободу выбора наиболее подходящих технических методов реализации. Реально процесс разработки программного обеспечения никогда не бывает простым, чаще применяется *каскадно-возвратный* метод:

Для преодоления проблем каскадно-возвратного метода используется *спиральная* модель жизненного цикла, упор в которой делается на начальные этапы: определение требований, их анализ и проектирование. На этих этапах реализуемость технических решений проверяется путем создания *прототипов*. Каждый виток спирали

соответствует созданию фрагмента или версии программ, на нем уточняются цели и характеристики проекта, определяется качество проектирования, планируются следующие работы. Основная проблема спирального цикла – определение момента перехода на следующий этап. Для ее решения вводят временные ограничения на каждый из этапов жизненного цикла.



Технологические процессы, составляющие жизненный цикл любого программного продукта, стандартизованы. Международной организаций по стандартам ISO, институтом IEEE и другими организациями, в том числе структурами Правительства России, утверждены стандарты, описывающие процессы, виды деятельности и задачи жизненного цикла программ и программно-аппаратных систем, а также результаты, достигаемые с помощью различных видов деятельности. С помощью стандартов удается проводить оценку качества проводимых процессов и выискивать возможности для их улучшения. Например, международный стандарт ISO/IEC 15504 (SPICE) Standard for Information Technology — Software Process Assessment (оценка процессов разработки и поддержки программного обеспечения) определяет правила оценки процессов жизненного цикла. Этот стандарт определяет 5 категорий процессов, включающих 35 процессов и 201 вид деятельности. Например, процесс приобретения программного обеспечения включает такие виды деятельности, как определение потребности в программном обеспечении, определение требований, подготовку стратегии покупки, подготовку запроса предложений и выбор поставщика.

1.2. Основные требования к системам программирования

Важнейшим из требований, которые предъявляются системам программирования, является требование ***согласованности интерфейсов и непротиворечивости результатов работы*** компонентов этих систем. Именно это согласование превращает наборы системных программ в единую систему, нацеленную на решение своей основной задачи – поддержку единого процесса подготовки программ.

Другое требование – ***полнота набора системных компонентов***, является важным, но вторичным. В мире существуют немного систем программирования, которые обеспечивали бы поддержкой весь процесс проектирования, разработки и сопровождения программных продуктов. Однако имеется некоторый уже обязательный круг компонентов, лакуны в котором недопустимы. Невозможно представить себе

систему программирования, в которой отсутствовали бы трансляторы. Уже давно обязательным компонентом считается редактор связей (компоновщик), позволяющий объединять раздельно созданные модули в единую программу. Наличие системных библиотек также является обязательным требованием к составу систем программирования. Среди современных систем программирования уже трудно найти системы без интерактивных отладчиков и справочных систем. В то же время, отсутствие компонентов, ответственных за первые этапы проектирования программ – от фиксации первичных требований к разрабатываемому программному продукту до разработки подробных спецификаций и структурированных описаний программ, в настоящее время еще не считается существенным недостатком систем программирования, и многие из них обходятся без таких компонентов, оставляя их системам проектирования другого рода. Можно ожидать, что в будущем, по мере внедрения автоматизированных технологий разработки программного обеспечения, системы, предназначенные для автоматизации различных стадий общего процесса проектирования и разработки, будут объединяться в единые комплексы.

Все большую важность в последнее время приобретает ***требование удобства*** работы с системами программирования и отдельными их компонентами. Важными являются возможности по поддержке работы в различных режимах, а также по поддержке ведения в системе нескольких разных проектов разработки программного обеспечения. От систем программирования требуется поддерживать как режим отладки программ, так и режим получения наиболее эффективного варианта программ. Поддержка нескольких проектов позволяет пользователям систем сохранять в архивах сделанные ими настройки и установки режимов для ведущихся ими проектов разработки и быстро извлекать их оттуда, легко восстанавливая сохраненный контекст.

1.3. Основные компоненты систем программирования

Содержание работ, производимых на отдельных этапах, базируется на результатах других этапов, все этапы должны быть согласованы между собой. Для достижения этого согласования необходимо вести *базу данных проекта*, что позволяет сохранять информацию о проекте и причинах принятия тех или иных решений при проектировании, а также контролировать его *внутреннюю согласованность*: изменения в составе требований должны приводить к указаниям на места проекта, в которых эти требования использованы. Все изменения, которые вносятся в проект, должны быть *корректными*, то есть не должны нарушать общие требования к проектируемому программному продукту. Должна проверяться *непротиворечивость* принимаемых решений. Все изменения в такой базе проекта можно проводить только санкционированно (специалист, реализующий какие-либо требования, не имеет право их изменять).

Содержимое базы проекта используется не только системой программирования, но и находящейся и функционирующей рядом *системой управления проектом*. Так же, как и система программирования, система управления проектом ведет свою работу на протяжении всего жизненного цикла проекта. Среди задач системы управления проектом находятся

- Планирование работ (составление списка задач, длительности и графиков выполнения проекта, оценка затрат на выполнение проекта, распределение ресурсов, необходимых для решения задач, распределение ответственности по решаемым задачам).

- Выявление источников затруднений (*устранимых* и *неизбежных* рисков), которыми могут быть:
  - недостаточное вовлечение в проект высшего руководства разработчиков,
  - невозможность привлечения к работам над проектом будущих пользователей программного продукта,
  - нестабильные требования к проекту, нехватка знаний или опыта в коллективе разработчиков,
  - организационно-политические риски.

  Управление рисками заключается в идентификации рисков и в предупреждении рисков, то есть снижении степени их негативного влияния на выполнение проекта. Способы предупреждения рисков могут сводиться либо к попыткам полностью их избежать, либо к действиям по их преодолению.
- Контроль и координация календарного плана работ: работа с единой базой проекта, идентификация хода проекта по графикам и диаграммам, рассылка сообщений участникам проекта, генерация отчетов о выполнении работ.

Наиболее известным программным продуктом, позволяющим осуществлять управление разработкой проектами, является компонент пакета офисных программ Microsoft Office, который носит название *Microsoft Project*. Доступны и другие системы управления проектами, например, система *TimeLine*, представляющая собой очень хорошую систему для ведения единой базы проектов. Широко используются системы *SureTrack* и *Primavera Project Planner* компании Primavera.

На этапе анализа и формулирования требований (и планирования проекта) необходимо иметь возможность использовать формальные языки описания требований, проводить анализ этих требований и их непротиворечивости, осуществлять моделирование проектируемой системы, используя поведенческие модели, а также выявлять необоснованно завышенные требования к программному обеспечению.

Основными компонентами систем программирования, используемыми на этапе анализа требований являются *текстовые и графические редакторы*, а также программные *средства автоматического контроля непротиворечивости* таблиц решений, функциональных диаграмм, текстов на языках спецификаций.

На этапе проектирования также возникают текстовые и графические материалы, поэтому в системах программирования также необходимы *текстовые и графические редакторы*. Если используется объектно-ориентированный подход к проектированию и объектно-ориентированная декомпозиция, полезными оказываются средства, позволяющие разработчикам автоматически строить визуальные описания классов объектов, просматривать их и согласованно редактировать. Поскольку этап проектирования опирается на результаты первого этапа, уже на этом этапе широко используется база данных проекта.

Этап программирования (кодирования) многими ошибочно называется этапом разработки программ. На самом деле разработке программ посвящены все этапы в совокупности, программирование есть лишь одна из нескольких стадий разработки. На этапе написания программ (кодирования) в системах программирования желательно иметь

- Средства автоматизации (пусть частичной) написания программ.
  Некоторые системы программирования включают в свой состав компоненты, облегчающие процесс написания программ и автоматически генерирующие заготовки программ, которые впоследствии превращаются программистами в полноценные программы. Например, при создании программ, взаимодействующих друг с другом с помощью брокеров запросов объектов, требуется перед программированием составить описание интерфейсов программных компонентов. Такое описание создается на языке определения интерфейсов (Interface Definition Language, IDL), трансляция с этого языка ведется в выбранный язык программирования (часто используются Си++ и Java) специальным компилятором, формирующим согласованные друг с другом заготовки текстов программ, как для серверной части программы, так и для клиентской части.
- Помощь в автоматизации графического интерфейса пользователя (*Graphical User Interface – GUI*). Поскольку, независимо от решаемой задачи, сами пользовательские интерфейсы для разных задач могут быть похожими, полезны генераторы графических интерфейсов.
- Библиотеки
  - содержательная часть (средства ввода/вывода, наборы функций, для объектно-ориентированных языков – классы и компоненты как системы классов)
  - средства работы с библиотеками (например, работа с иерархиями классов), то есть библиотекари.
- Средства редактирования текстов (программ). Лучше иметь синтаксически ориентированные текстовые редакторы, которые
  - обеспечивают подсветку служебных слов,
  - обеспечивают работу с функциональной клавиатурой, позволяющей вводить сложные языковые конструкции одним нажатием на клавишу,
  - проверяют баланс скобок.

  Последнее важно не только при вводе сложных арифметических выражений, но и при работе с языками программирования, построенными на длинных конструкциях со сбалансированными скобками.
- Трансляторы (используются также на других этапах разработки, например, при тестировании и отладке).
- Средства компоновки больших программ, которые на основе отдельно транслируемых фрагментов (модулей) и компонентов библиотек создают исполняемые программы (редакторы связей).

Некоторое время назад в состав систем программирования включались загрузчики – программы, обеспечивающие настройку программ на адреса в памяти вычислительной машины и размещение программ в этой памяти, однако, сейчас загрузчики из компонентов систем программирования превратились в компоненты операционных систем.

Этап программирования во многом сформировал современное представление о том, какие системы программирования нужны для создания высококачественного программного обеспечения. При этом развитие систем программирования приводит к тому, что в их состав включается все большее количество компонентов, имеющих

отношение не только (и не столько) непосредственно к этапу программирования, но и к другим этапам.

Этап компоновки есть этап формирования полной программы или программного комплекса из некоторого количества автономно запрограммированных, автономно отлаженных и автономно протестированных компонентов. Для компоновки программ используется часть инструментария, применяемого также на этапе программирования (прежде всего, редакторы связей). Кроме того, для использования на этом этапе создается специальный инструментарий, в котором особую важность имеют средства контроля версий программных компонентов.

Важную роль в получении высококачественного программного продукта играют средства, применяемые ***на этапах отладки и тестирования программ***. В состав этих средств входят

- Отладчики, как очень простые, уже давно включенные в состав систем программирования, так и более сложные, позволяющие устанавливать точки останова, просматривать значения переменных, проводить прямую и обратную трассировку хода выполнения программ. Наиболее удобные отладчики работают в терминах классов, объектов, переменных, используя семантику и терминологию пользователей.
- Генераторы тестов, позволяющие формировать входные данные для трансляторов. Генераторы тестов также способны генерировать тесты, содержащие ошибочные конструкции, что позволяет проверять диагностические средства трансляторов.
- Средства автоматизации прогонов тестов, позволяющие избежать “ручного” запуска большого количества тестов. В случаях тестирования графических интерфейсов полезны средства запоминания (и последующего воспроизведения) действий пользователей по нажатию клавиш и открытию элементов управления интерфейсом (окон, форм).
- Средства автоматизации (хотя бы частичной) анализа результатов прогона тестов. Эти результаты могут быть сложными, поэтому уровень автоматизации зависит от тестируемого приложения.
- Средства анализа уровня тестового покрытия (применяются при тестировании программ методом “белого ящика”). Необходимо также проверять покрытие условий, решений и т. д.

Многие современные системы программирования предоставляют развитые средства ***этапа документирования*** создаваемых программных продуктов, сложность которых постоянно возрастает, что, в свою очередь, делает все более важной хорошую документацию. К документации программных продуктов следует относить не только пользовательскую документацию, предназначенную для фазы использования, но также техническую документацию для сопровождения программных продуктов.

Выполнение работ ***на этапах внедрения и тиражирования*** требует использования средств управления проектами, а также средств управления версиями программных продуктов.

2. Классическая система программирования

Практически невозможно встретить систему программирования, в которой последовательно и полно были бы представлены сразу *все* компоненты, обеспечивающие полноценную поддержку процесса создания программ *на всех* стадиях и этапах. Чаще всего отсутствуют средства первых этапов – подготовки требований и проектирования. Иногда системы программирования строятся без какого-либо интегрирующего звена. Первые операционные системы могли позволить пользователям-программистам запускать компоненты систем программирования только в диалоге, последовательно, одну за другой, или собирая приказы на запуск этих компонентов в одном пакетном задании.

Со временем были развиты сервисные средства операционных систем и появились командные процессоры, что позволило объединять последовательность вызовов системных программ в единые *командные файлы*. Это упростило работу по запуску компонентов систем и позволило перейти к еще более удобному способу работы – использованию специализированных командных процессоров, ориентированных на запуск компонентов систем программирования.

Специализированные командные процессоры (программы под обобщенным названием *координаторы make* или *shell*) представляли собой интерпретаторы, на вход которым подавались файлы, записанные на особом командном языке. На этом языке кодировалась вся последовательность действий, необходимых для порождения результирующего исполняемого файла. В программе для командного процессора (*Makefile*) перечислялись все используемые входные текстовые файлы, библиотеки программ, все порождаемые объектные файлы, а также параметры запуска отдельных компонентов систем программирования и правила обработки каждого отдельного файла. Удобны координаторы *make* тем, что позволяют отслеживать зависимости в больших наборах фалов, составляющих цельную программу, что упрощает процесс сборки проектов.

Любая современная система программирования содержит не только трансляторы, но и служебные библиотеки процедур, функций, макроопределений, классов, а также средства компоновки программных комплексов из отдельных составляющих – модулей

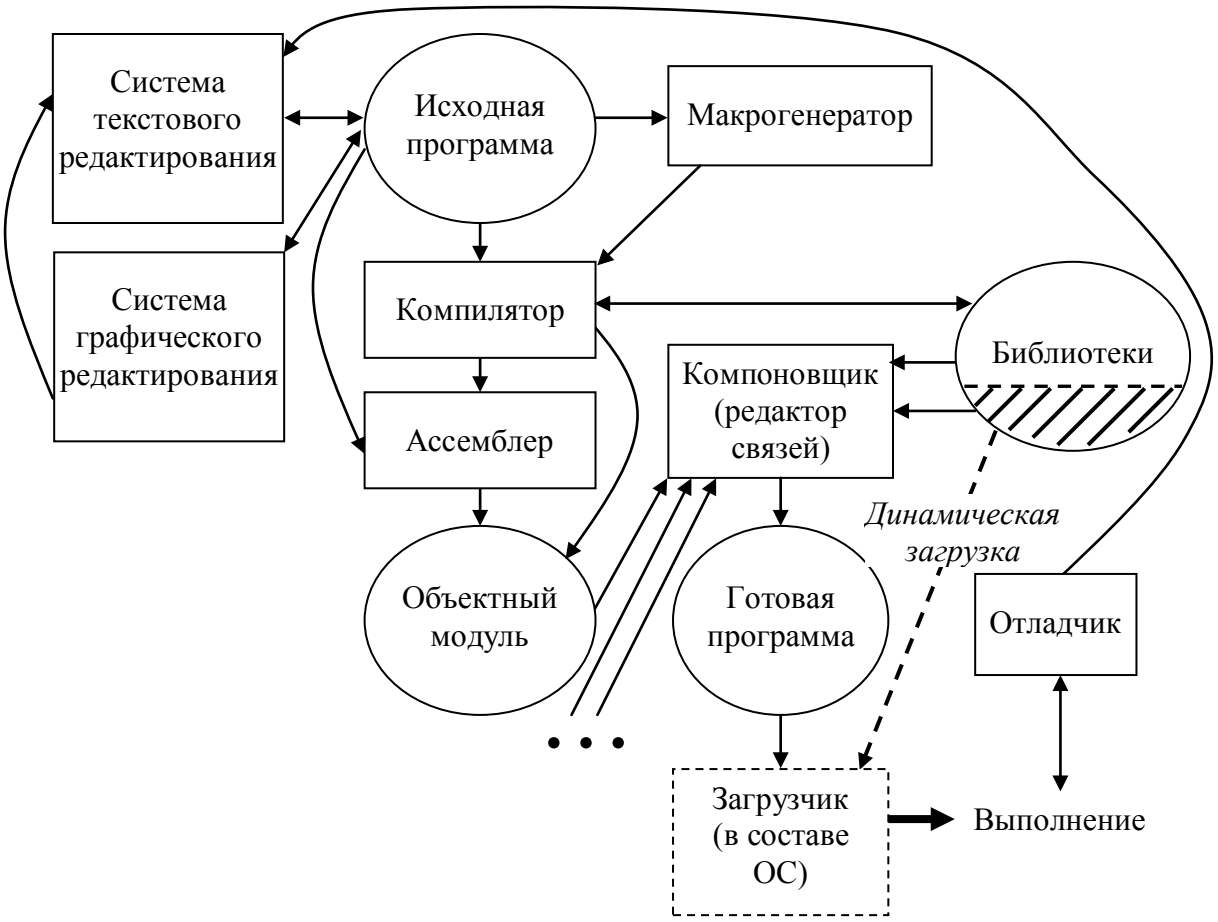
Современные системы программирования все чаще взаимосвязаны, а иногда и интегрированы с программными средствами, применяемыми на первых этапах жизненного цикла – от формулирования требований и их анализа до автоматизированного проектирования программ, включая в свой состав средства автоматизированного проектирования (CASE-технологии). Такие системы имеют следующие характерные особенности:

- поддержка единой базы проектов (репозитория);
- поддержка визуальных методов проектирования (графическое создание, редактирование, визуализация, печать отчетов по ходу выполнения проектов);
- использование унифицированного языка моделирования для фиксации решений, принятых при проектировании;
- наличие и интеграция всех средств поддержки для всех этапов жизненного цикла программных продуктов.

2.1. Общая схема работы систем программирования

Большинство систем программирования включают в свой состав ограниченный набор компонентов:

- средства интеграции компонентов системы программирования
- редакторы текстов
- компиляторы и ассемблеры
- библиотеки
- редакторы связей
- средства конфигурирования и управления версиями
- отладчики и средства тестирования
- средства тестирования
- профилировщики
- справочные системы.



2.2. Интегрированная среда разработки

Основной тенденций развития систем программирования является тенденция интеграции компонентов в единое системное окружение. Если первые системы программирования представляли собой простые наборы программ, объединенные только общим описанием структур данных, то в настоящее время из прежних систем программирования возникли интегрированные среды разработки.

Простейший способ управления сложными последовательностями вызовов системных программ, какими являются вызовы компонентов систем программирования, называется ***режимом работы с командной строкой*** оператора или операционной системы:

```
g++ -c -S -da -dp -dA b.cpp
```

Работая в этом режиме, программист подает операционной системе задания по вызову программ, указывая место размещения вызываемой программы, ее имя и параметры конкретного вызова. Существенным недостатком работы в таком режиме является необходимость постоянного присутствия квалифицированного пользователя системы на рабочем месте для постоянного запуска следующих и следующих компонентов.

В поисках преодоления этого недостатка разработчики систем программирования предложили ***концепцию командных файлов***.

В таких файлах заранее записывались директивы операционной системы, необходимые для активации всего тракта обработки программ – от запуска компилятора до записи в архив системы готовой программы или передачи ее на исполнение, как в таком файле, содержащем директивы операционной системы MS-DOS:

```
@echo off
@if exist t40.* del t40.*
@if exist kopu.?n del kopu.?n
@if exist kopu.?n0 del kopu.?n0
@for %%f in (*.a40) do ..\..\exe\ac40 -d-l –i%%f
@..\..\exe\rsv -it40.o40 -zt40.r40
@..\..\exe\nstr -k0x0 -p0x0 -it40.r40 -zt40.n40
@..\..\exe\Cpu -x0x86 -d-s0xfe000 -b0x4005ab -e0x400900 -zt40.n40 -k0x0 -f0x0 -ot40.se0
@del t40.?40
```

Особенно эффективна работа в режиме командного файла при выполнении массовых подготовок версий одной программы, отличающихся каким-либо важным параметром, например, паролем, блокирующим несанкционированный запуск готовой программы. Процесс подготовки большого числа почти одинаковых программ проводится в пакетном режиме работы операционной системы без оперативного контроля со стороны оператора и даже без его присутствия.

Все большее распространение получают *интегрированные системы программирования*. Интегрированная среда разработки программ объединяет в себе возможности текстовых редакторов и командного языка компилятора, а также некоторых других компонентов системы программирования, например, редактора связей и отладчика.

При работе в интегрированной среде разработчик программы освобождается от необходимости устанавливать параметры запуска компонентов системы при всяком запуске этих компонентов. Не требуется от пользователя и знания языка управления заданиями операционной системы. Работа в интегрированной среде ведется в терминах описания *программного проекта* и его характеристик. Пользователю дается удобный графический интерфейс, с помощью которого он может определять свои программные проекты, включать в их состав свои текстовые, ресурсные и библиотечные файлы,

устанавливать параметры запуска отдельных компонентов системы. Интегрированная среда автоматически осуществляет запуск необходимых компонентов (фактически создавая невидимый программисту командный файл), получает от этих компонентов результаты их работы и сообщает пользователю обо всех допущенных им ошибках на разных этапах подготовки программ.

Основное преимущество интегрированной среды – в удобстве работы ее пользователя. Все свои действия он выполняет непосредственно в окне редактирования исходного текста программы. Получив сообщение о какой-нибудь ошибке, пользователь, руководствуясь сопутствующими сведениями о месте этой ошибки в программе, имеет возможность исправить ее, не прерывая работу в интегрированной среде. Дополнительные удобства работы в таком режиме связаны с реальным объединением в одном технологическом процессе нескольких компонентов системы программирования – текстовых редакторов, компиляторов, редакторов связей, отладчиков.

Естественным следствием внедрения в практику интегрированных систем программирования явилось перенесение технологических приемов, использованных при их разработке на разработку того программного обеспечения, которая выполняется с их помощью. Программистам стали предоставлять графический пользовательский интерфейс, ставший вскоре после своего появления неотъемлемой составной частью многих современных операционных систем и графических оболочек. В состав систем программирования стали включаться библиотеки, обеспечивающие поддержку графического интерфейса. Описание библиотек выполнялось с помощью прикладных программных интерфейсов, позволявших стандартизовать взаимодействие с операционными системами.

Особенную помощь интеграция компонентов системы программирования приобретает при проведении отладки программ. Тесные связи между отдельными компонентами помогают отладчику так представить информацию о ходе выполнения программы, что отладка оказывается значительно более простым мероприятием, чем это обычно бывает.

Большинство систем программирования ориентированы на работу с каким-либо одним языком программирования. Тесная связь с языком позволяет включать в состав интегрированных систем полезные вспомогательные средства, которые способны проводить анализ текстов программ и находить в них определения объектов данных, их структур, классов. Например, в системе программирования Visual C++ компании Microsoft при работе с программой в одном из окон интегрированной системы демонстрируется полная иерархия классов, относящаяся к активному проекту. Выбрав любой из элементов иерархии, пользователь сразу видит в другом окне, предназначенном для отображения основного текста программы, тот фрагмент программы, где определяется выбранный элемент. При редактировании и отладке это позволяет легко отыскивать в тексте места определения объектов, проверять способы использования объектов на соответствие определениям, отлаживать не только процедуры обработки объектов, но и сами данные и их структуры.

Интегрированные среды имеются не только в системах Microsoft Windows для персональных компьютеров. В семействе систем под условным наименованием UNIX уже давно функционирует система оконного пользовательского интерфейса X Window System. В эту систему заложена идея разделения функций отображения информации на экране по принципу – функции клиента и функции сервера. Сервер выполняет саму

отрисовку, управляет дисплеем, а клиент выполняет вычисления и связывается с сервером при необходимости отобразить их результаты на экране. Взаимодействие клиентов и серверов осуществляется на основе сетевых протоколов, что значительно облегчает работу в многопользовательском и многоэкранном режимах, а также повышает уровень безопасности взаимодействия.

Многие программные компоненты системы UNIX получили возможность работать не только в режиме командной строки, но и в интегрированной среде X Window. Например, редактор Emacs имеет более удобную и чаще используемую модификацию XEmacs. Для интерфейса X Window созданы многочисленные диспетчеры окон, самым известным из которых является, по-видимому, диспетчер Motif, с помощью которого для работы с окнами интерфейса возникают дополнительные возможности.

В последнее время все большую популярность приобретает расширяемая интегрированная среда Eclipse. Это свободно распространяемый пользовательский интерфейс, обладающий развитым визуальным редактором, к которому могут подключаться как коммерческие, так и свободно распространяемые модули, реализующие различные функции.

Развитием интегрированных систем (сред) программирования являются системы программирования, интегрирующие компоненты, обеспечивающие поддержку на этапах разработки программ, предшествующих этапу кодирования, то есть на этапах, связанных с проектированием программ.

3. Компоненты классической системы программирования

Свойства систем программирования определяются не только компиляторами, но и многими другими компонентами этих систем, всей их совокупностью и способностью взаимодействовать друг с другом в процессе подготовки программ.

3.1. Редакторы текстов

3.1.1. Виды текстовых редакторов

В соответствии со схемой классической системы программирования можно выделить три начальных элемента процесса создания программы: редактор текста исходной программы, подсистема автоматизированного проектирования и редактор графических форм ведения диалога.

Текстовый редактор является наиболее частым начальным элементом процесса создания программы. Он позволяет готовить и вносить изменения в тексты исходных программ, однако в современных системах программирования его функции стали еще более широкими. Текстовые редакторы стали основой интегрированных сред разработки. Известным примером текстового редактора является редактор ***vi***, входящий в состав стандартной системы программирования операционной системы UNIX. Редакторы называются текстовыми, поскольку они предназначены для редактирования и хранения в архиве любой текстовой информации – от текстов программ до документации по вычислительной системе. Текстовые редакторы делятся на две категории по видам запуска, они бывают *пакетными* и *диалоговыми.*

Пакетные редакторы не требуют непосредственного присутствия программиста для своей работы. Они получают на вход исправляемый текст и пакетное задание на редактирование, в котором указано, какие фрагменты текста надо из текста исключить, какие переставить местами, какие фрагменты следует заменить другой информацией, которая также включена в пакетное задание. Указания могут даваться в терминах номеров строк (заменить строку 15 на текст "…") или с помощью контекста (перед строкой, в которой найден идентификатор "`int`" вставить строку с идентификатором "`long`").

Пакетные редакторы особенно удобны при пакетном формировании нескольких версий одних и тех же программ, отличающихся некоторыми важными параметрами, которые должны быть учтены непосредственно в тексте программы. Запуск пакетного редактора может осуществляться из командной строки или с помощью командного файла, но в любом случае файл с заданием на редактирование должен быть подготовлен заранее.

Одним из видов контекстного пакетного редактирования является ***макроподстановка***, то есть редактирование текста по найденным в нем шаблонам. Иногда такое редактирование текста встраивается непосредственно в компилятор, что имеет некоторые преимущества, связанные с повышением эффективности общего процесса обработки текста программы. Процесс макрообработки текстов концептуально делится на две фазы – первая из них это ввод макроопределений, вторая фаза состоит в обработке макровызовов. Вид макровызовов и макроопределений также вносит различия в макропроцессоры. В одних макропроцессорах (более традиционных) макроопределения напоминают определения функций и их формальных параметров, а макровызовы напоминают операции вызова функций с фактическими параметрами. Примерами таких макропроцессоров могут служить препроцессоры, встроенные в компиляторы некоторых языков программирования (PL/1, Си, Си++, различные языки ассемблера). Другие макропроцессоры считают макровызовом произвольную строку

текста, соответствующую некоторому заранее определенному шаблону, причем правила описания шаблонов могут достаточно сложными, а сами шаблоны могут быть параметризованными. Тем самым, при отождествлении входной строки с шаблоном возможно не полное совпадение, а совпадение с выделением фактических параметров.

Существует большое количество универсальных макропроцессоров или макрогенераторов. В некоторых разработках они использовались для повышения мобильности программного обеспечения, то есть его переносимости из одного операционного окружения в другое.

Диалоговые редакторы отличаются от пакетных редакторов тем, что для них готовить задание на редактирование не требуется. Пользователь указывает редактору, какой текст он собирается редактировать, далее непосредственно вводятся редактирующие приказы. В результате формируется исправленный текст, который можно снова записать в системный архив, использовав прежнее имя файла или задав новое имя, сохранив предыдущий вариант в архиве.

Диалоговые редакторы делятся на две категории: *строчные* и *экранные* редакторы.

При работе со строчными редакторами сначала задают номер строки, которая подлежит редактированию, а затем выдают редактирующие приказы, которые могут влиять только на заданную строку. Экранные диалоговые редакторы позволяют видеть на экране сразу несколько строк. Экранные редакторы – это самое удобное средство редактирования файлов, а строчные обычно применяются в условиях, когда устройство отображения информации не позволяет одновременно показывать сразу несколько строк текста.

Появление экранных редакторов серьезно повлияло на возможности, предоставляемые пользователям при редактировании. Пользователю теперь предоставляют набор как традиционных редакторских возможностей, основанных на редактировании строк по их номерам, которые обычно показываются среди другой служебной информации, так и контекстное редактирование. К этому добавлены возможности работы с блоками информации, причем блоки обычно бывают горизонтальными (построчными) или вертикальными, которые особенно удобны при редактировании форматированных текстов, например, таблиц. Среди экранных редакторов выделяются текстовые процессоры, примером которых является текстовый процессор Word, разработанный компанией Microsoft и входящий в состав системы офисной автоматизации Microsoft Office.

Появление интегрированной среды разработки позволило интегрировать в них и текстовые редакторы, точнее диалоговые экранные редакторы текстов. Редакторы тестов стали теснее взаимодействовать с компиляторами, а затем и с отладчиками программ. Теперь при вводе текстов программ с помощью редактора, эта программа, получив сведения о том, на каком языке программирования написан вводимый текст, осуществляет проверку правильности написания ключевых слов языка, выделяя их на экране особым шрифтом и цветом. Тем самым, текстовым редакторам передаются некоторые функции лексических анализаторов.

Некоторые системы программирования имеют особую структуру, предполагающую ввод исходной информации в виде графических объектов с помощью ***графического пользовательского интерфейса***. К таким системам относятся, например, системы, работающие с языками UML, Visual Basic. Графические образы

могут впоследствии автоматически преобразовываться в обычные текстовые программы.

3.1.2. Лексический анализ “на лету”

Суть лексического анализа “на лету” – в поиске и выделении лексем входного языка в тексте программы непосредственно в процессе ее создания разработчиком. Одновременно с вводом текста программы с помощью текстового редактора система программирования отыскивает в этом тексте лексемы по правилам того языка программирования, на работу с которым она в данный момент настроена.

В самых простых системах проведенный анализ используется только для выделения лексем в тексте с помощью графических средств дисплея. Например, литеральные константы могут выделяться одним цветом, служебные слова могут подчеркиваться, простые идентификаторы показываться каким-либо другим шрифтом и т. д. Более развитые системы строят таблицы идентификаторов и констант, которые передаются компилятору, входящему в систему программирования. Этим достигается интеграция текстового редактора с лексическим анализатором компилятора.

Таблицы, создаваемые в процессе ввода текста, могут использоваться для поиска сохраненных там лексем по типу или по некоторым информационным символам (например, по первым буквам). Такой поиск можно сделать контекстно-зависимым, то есть можно искать лексему именно того типа, который допустим в данном конкретном месте текста программы. Кроме самой лексемы, разработчику может быть предоставлена информация о ней, например, перечень доступных методов для типа или экземпляра класса, что избавляет от необходимости лишний раз заглядывать в документацию, и снижает вероятность ошибок при вводе текста. Помощь со стороны системы программирования, в которой текстовый редактор интегрирован не только с лексическим, но и с синтаксическим анализатором компилятора, может быть организована в виде *подсказок* и *гиперссылок*.

Подсказка возникает на экране в том случае, когда разработчик ввел какую-то часть исходного текста, определенную лексическим анализатором как начало некоторого заслуживающего внимания фрагмента. Подсказка может принимать форму *пояснения* или *варианта* дальнейшего ввода текста.

Пояснение дает разработчику представление о том, что может следовать дальше. Пользователь должен сам вводить этот текст, но при этом пояснение будет помогать ему. Например, при вводе имени функции может возникнуть подсказка с указанием типов ее параметров, тип текущего параметра может выделяться особым шрифтом.

Вариант текста предлагает разработчику некоторый шаблон, который можно вставить после уже введенного текста. В некоторых случаях указывается сразу набор шаблонов, из которого можно выбрать подходящий. Например, после ввода переменной, соответствующей некоторому классу, подсказка может предложить перечень методов и свойств этого класса, из которого можно выбрать нужный простым выбором нужной строки в перечне. Некоторые часто встречающиеся варианты вводимых текстов в ряде систем разрешается связывать с функциональными клавишами, нажатие которых приводит к автоматическому добавлению связанного текста к программе.

Подсказки не заменяют разработчика, они лишь немного помогают провести наиболее рутинную работу – ввод текста. Подсказки делаются лишь на основании лексического и синтаксического анализа, а этого не всегда достаточно для полного понимания ситуации, поэтому подсказки не всегда оказываются правильными. Однако

разработчик не обязан следовать подсказкам, он всегда может вводить тот текст, который считает необходимым ввести.

Гиперссылка позволяет переходить от одной части программы к другой. Например, по имени функции можно легко перейти в то место программы, где эта функция определяется, а по имени сложного типа данных перейти к его описанию.

Текстовые редакторы интегрируются также с отладчиками, для которых они позволяют расставлять в программе точки останова, показывать значения переменных во время приостановки работы программы и выполнять другие полезные действия.

3.2. Трансляторы, компиляторы, интерпретаторы

Конечной целью создания всякого программного продукта является достижение некоторого результата, способ получения которого закодирован в этой программе. Сам результат может быть получен только при работе аппаратуры вычислительной системы, которой для работы должна быть передана программа в своем исходном или переработанном виде, а также входные данные, требующиеся программе при ее работе.

Существует несколько вариантов взаимодействия программ с аппаратурой. Первый из этих способов почти не требует системы программирования и связан ***с кодированием программ непосредственно на машинном языке***. Такой подход становится приемлемым только в тех случаях, когда над всеми другими соображениями по поводу способов записи программ превалируют соображения эффективности. В настоящее время можно считать, что ни одна вычислительная система не воспринимает напрямую программы, подготавливаемые к исполнению людьми: все программы для получения результата их работы должны пройти предварительную обработку.

Второй вариант, в котором ***программирование ведется на языке программирования***, не совпадающем с машинным языком вычислительной системы, требует наличия системы программирования. В состав системы программирования включаются несколько компонентов, важнейшим из которых является компонент, ответственный за преобразование исходной программы к виду, в котором она может быть понята вычислительной системой. Эти компоненты называются *трансляторами*, то есть программами, которые переводят *исходную программу*, написанную на некотором исходном (входном) языке, в другую программу, эквивалентную первой. Получающаяся при этом программа тоже формулируется на некотором языке, называемом *объектным*.

Процесс перевода с исходного языка на объектный язык охватывает сразу три программы и называется трансляцией:

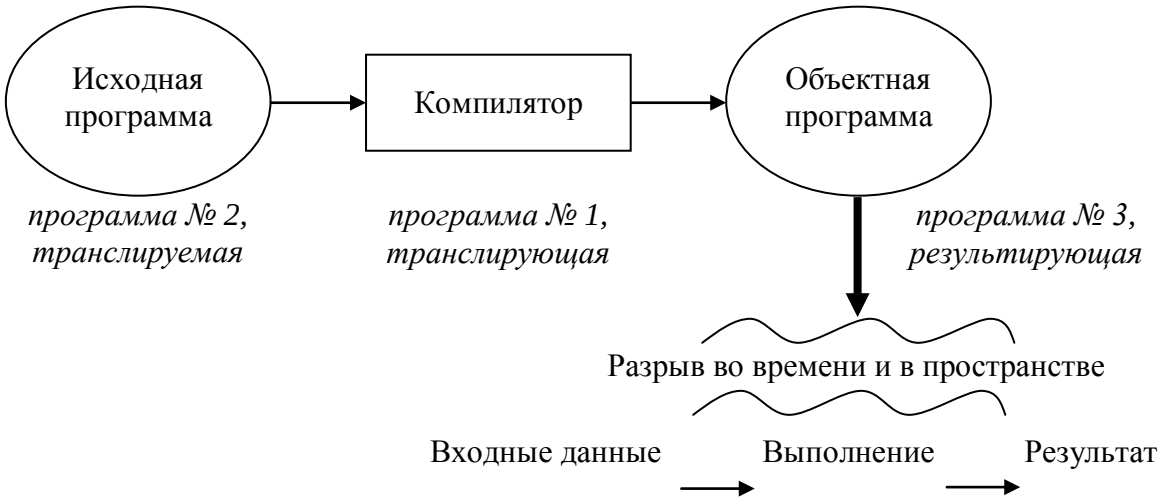
1. Во время трансляции вычислительная система выполняет программу транслятора (программа № 1, *транслирующая*).
2. Транслятор обрабатывает конкретную последовательность предложений входного языка. Такая последовательность должна удовлетворять набору синтаксических и семантических правил. В этом случае мы получаем право называть ее программой (программа № 2, *транслируемая*).
3. Результатом работы транслятора также является программа, строится она по синтаксическим правилам выходного языка, ее семантика определяется семантикой выходного языка (программа № 3, *результирующая*).

Исходная и результирующая программы являются *эквивалентными*. Эквивалентность программ означает совпадение их смысла с точки зрения (семантики) входного языка (для входной программы) и (семантики) выходного языка (для объектной программы).

3.2.1. Схемы работы трансляторов

Степень преобразования программ трансляторами может быть различной. Если исходная запись программы ведется на языках, близких к машинному представлению программ, такая обработка может быть относительно несложной. Программы, которые обрабатывают тексты на таких языках, называются *ассемблерами*. Несмотря на большую разницу в аппаратуре разных вычислительных машин, их языки ассемблера часто очень похожи друг на друга, отличаясь только представлением самих машинных команд. Для языков ассемблера разработан стандарт, в котором специально указано, что все подобные языки должны обрабатываться соответствующими ассемблерами на основе принципа "*один-в-один*".

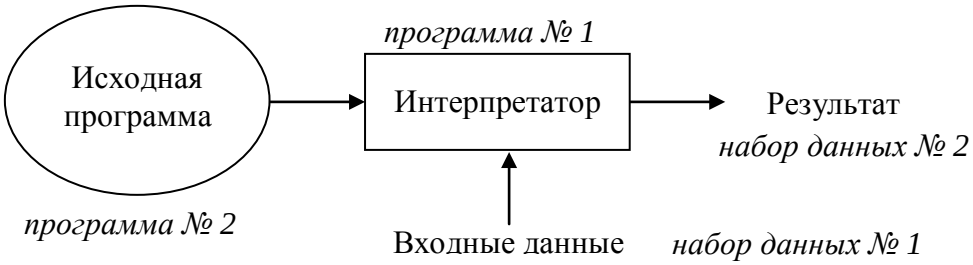
Другой вид трансляторов – *компиляторы*. Термин *компилятор* обычно используется вместо термина *транслятор* в тех случаях, когда исходным языком трансляции является язык программирования высокого уровня, например Паскаль или Си++, а объектным языком является автокод, язык ассемблера или машинный язык некоторой вычислительной машины. Вычислительная система, для которой ведется компиляция, называется *целевой вычислительной системой*. В это понятие входит не только архитектура аппаратных средств ЭВМ, но и операционная система, работающая на этой аппаратуре, а также набор динамически подключаемых библиотек, которые необходимы для выполнения объектной программы и, фактически, становятся ее частью:



Сам компилятор может работать в другом операционном окружении, нежели то, в котором будет выполняться откомпилированная им программа.

Третий способ использования вычислительной аппаратуры для получения результата программы связан с процессом, называемым ***интерпретацией языка***. Интерпретация выполняется программой, называемой *интерпретатором*. При интерпретации некоторой программы она размещается не в той области памяти вычислительной машины, которая предназначена для выполняемых программ, а там, где обычно размещаются исходные данные выполняемых программ.

В отличие от компилятора и ассемблера, *интерпретатор* некоторого исходного языка выполняет действия, которые этой программой предписываются (для этого интерпретатору необходимо передавать еще и входные данные программы). Интерпретатор не порождает объектную программу, которая впоследствии должна выполняться, а выполняет ее сам. Это и есть принципиальное отличие интерпретатора от компилятора. Итогом работы интерпретатора является результат, определяемый смыслом исходной программы, если исходная программа правильна синтаксически и семантически, либо сообщение об ошибке, в противном случае. В процессе интерпретации участвуют только две программы (программа интерпретатора и исходная программа) и два набора данных (входные данные программы и ее результат):



3.2.2. Смешанная стратегия трансляции

Иногда интерпретатор сначала производит преобразование исходной программы в некоторое внутреннее представление, которое затем программно интерпретируется. Такой подход называется *смешанной стратегией трансляции*, это наиболее часто возникающая на практике ситуация. Как и языки ассемблеров, внутреннее представление программ в интерпретаторах разрабатывается в таком виде, чтобы на второй фазе (фазе интерпретации) легко его расшифровывать и тратить минимум времени на анализ каждого отдельного предложения внутреннего языка при его выполнении. Именно поэтому *интерпретаторы также относят к трансляторам*. В этом смысле термин транслятор является самым общим и обозначает, как компиляторы и ассемблеры, так и интерпретаторы.

Некоторые интерпретаторы построены так, что исполняют исходную программу последовательно, по мере поступления программы на вход интерпретатора. Пользователю при этом не надо ждать завершения интерпретации, чтобы увидеть первые результаты работы программы, он может получать результаты постепенно, по мере работы интерпретатора. Не все языки программирования допускают такое построение интерпретаторов. Для того, чтобы это было возможно, язык должен одновременно допускать возможность существования однопроходного компилятора для этого языка. Это ограничение приводит к таким свойствам языков программирования, как необходимость описаний объектов данных до их первого использования в программе. Например, не могут интерпретироваться таким способом программы на языках программирования, если эти языки допускают использование некоторых объектов прежде, чем в тексте встретится описание этих объектов.

Скорость выполнения программы интерпретаторами во много раз меньше, чем при использовании компиляторов. Кроме того, при интерпретации исходная программа должна подвергаться разбору всякий раз при ее выполнении, а при компиляции разбор выполняется только один раз, после чего используется уже не исходный текст, а объектный файл с готовой программой. Однако интерпретаторы обладают

независимостью от архитектуры целевой вычислительной системы, в то время, как при компиляции готовые программы всегда ориентированы на эту архитектуру.

Разнородность оборудования, с которой постоянно приходится сталкиваться в таких сетях, например, в сети Интернет, препятствует использованию компиляторов, но способствует развитию систем, интерпретирующих тексты исходных программ, либо систем с двойной технологией – компиляции и интерпретации, в которых, в зависимости от требований пользователя исходная программа либо компилируется, либо интерпретируется. Наиболее известным примером интерпретируемого языка является язык разметки гипертекста (***H**yper**T**ext **M**arkup **L**anguage – **HTML***). Смешанная стратегия трансляции применяется в системах, работающих в сети Интернет, программы для которых пишутся на языке Java.

3.3. Компилятор как основной компонент системы программирования

На начальном этапе развития теории построения компиляторов их сравнивали между собой по количеству проходов по тексту исходной программы, которые выполнялись при компиляции программ. Это количество может меняться от одного до нескольких десятков, на их количество, кроме свойств исходного языка, могут влиять также свойства операционного окружения, в котором работает компилятор. *Проход* – это процесс последовательного чтения компилятором данных из внешней памяти, их обработки и записи результата во внешнюю память. Во время одного прохода может выполняться сразу несколько фаз компиляции, но случается, что одна фаза компиляции (например, синтаксический анализ) выполняется за несколько проходов.

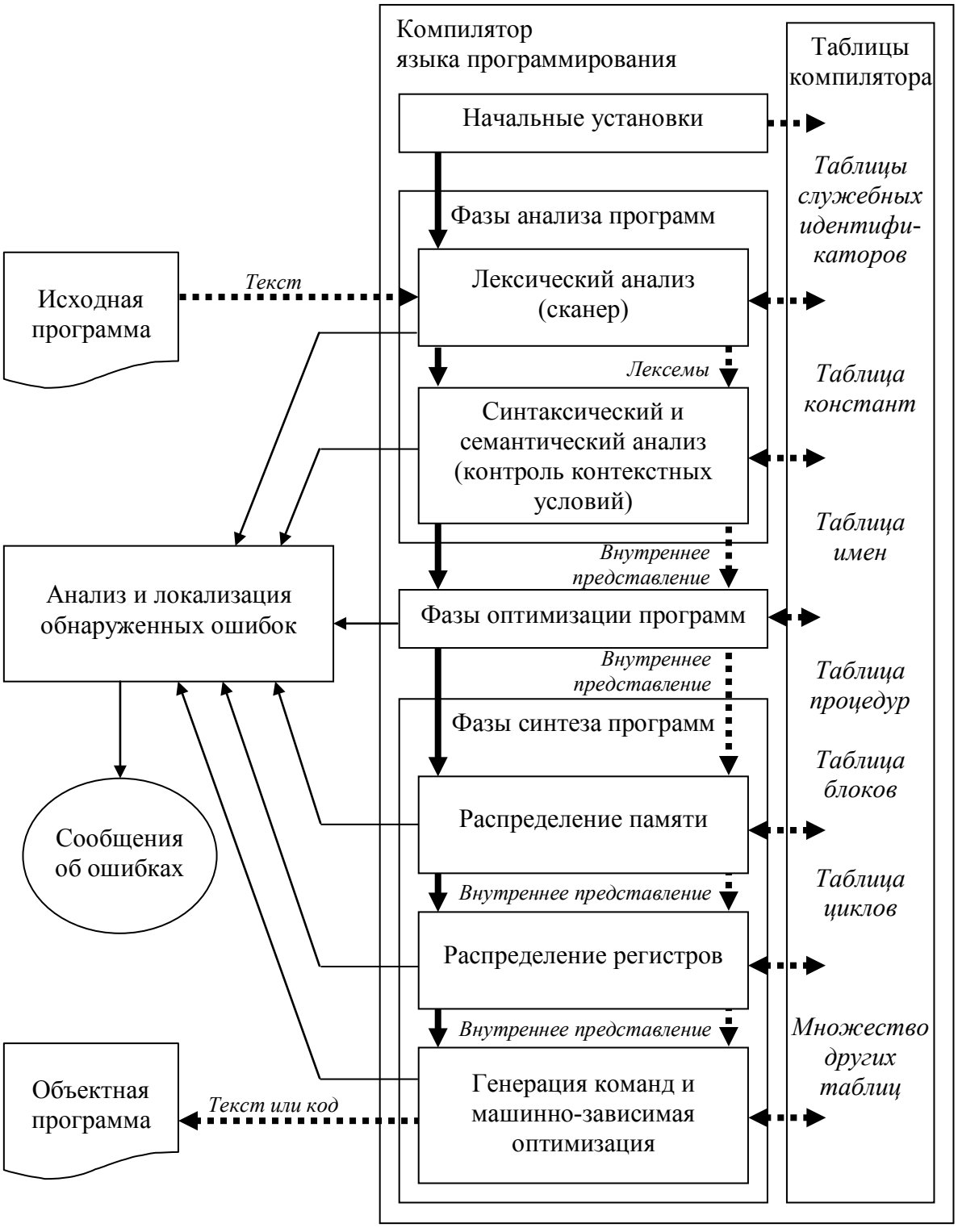
В настоящее время практически все компиляторы языков программирования просматривают сам исходный текст только один раз. Если же такого просмотра оказывается недостаточно, а его часто оказывается недостаточно, организуются дополнительные просмотры отдельных фрагментов программы, причем выполняются они не по исходному тексту, а по его внутреннему представлению. Чтобы построить такое внутреннее представление программы, в компиляторе строятся сложные таблицы, в которые заносятся сведения, извлекаемые из текста программы на первом и единственном его просмотре.

3.3.1. Общая схема работы компилятора

На следующей странице изображена схема работы компилятора языка программирования. Сплошные стрелки на этом рисунке указывают порядок работы компилятора, а пунктирные линии – потоки информации.

3.3.1.1. Основные компоненты компилятора и фазы компиляции

Информационные таблицы. При анализе программы из имеющихся в ней описаний, заголовков процедур, блоков, циклов и других структурных операторов извлекается информация, которая должна сохраняться для использования на последующих фазах компиляции. Вся эта информация размещается в информационных таблицах компилятора. Если при просмотре программы встречается некоторый идентификатор , в таблицах должны быть сведения о том, как он был описан и как использовался в программе. Конкретная информация зависит от исходного языка, от объектного языка и сложности транслятора. Кроме таблицы идентификаторов (“имен”) обычно строится таблица констант, в которую включаются все константы, использованные в программе, и заносятся их адреса в объектной программе.



Начальные установки. Перед началом работы с программой компилятору необходимо провести инициирующие мероприятия. К таковым относятся ввод и обработка режимов запуска компилятора и первичное заполнение таблиц исходной информацией.

Фазы анализа программ. На этапе анализа в компиляторе выполняется распознавание текста исходной программы и заполнение информационных таблиц. В результате формируется некоторое внутреннее представление программы, понятное

следующим фазам компиляции. Получая на вход исходную программу как последовательность символов входного языка (“*цепочку*”), компилятор должен проверить, принадлежит ли она входному языку, а также определить правила, по которым эта последовательность строилась. Анализ часто подразделяется на лексический, синтаксический и семантический анализ.

Лексический анализатор (сканер). Основная задача лексического анализатора – просмотреть весь текст исходной программы и выделить в нем лексемы (*минимальные лексические единицы или элементы текста программы, обладающие смыслом в рамках данного языка*). В обычных языках программирования лексемами являются числа (десятичные целые, вещественные), идентификаторы, служебные слова, разделители. Задачей лексического анализатора является замена разнообразных элементов текста стандартно выглядящими лексемами, которые в дальнейшем будет легче обрабатывать в других частях компилятора. Лексический анализ сопровождается исключением незначащих фрагментов текстов программ, например, комментариев. Для тех языков, в которых имеются макросредства, дополнительно выполняется расширение макровызовов. Подробнее задачи и проблемы лексического анализа рассматриваются в пособии “Формальные грамматики и языки. Элементы теории трансляции”.

Синтаксический и семантический анализаторы. Программа должна быть проверена на синтаксическую и семантическую правильность (должно быть проверено соблюдение контекстных условий), разделена на составные части, для каждой из которых должно быть сформировано внутреннее представление. В таблицы транслятора должна быть занесена вся информация, которую можно извлечь из обрабатываемой программы. Подробнее задачи и проблемы синтаксического и семантического анализа рассматриваются в разделе 3 3 2 и в пособии “Формальные грамматики и языки. Элементы теории трансляции”.

Внутреннее представление исходной программы. Внутреннее представление исходной программы в компиляторе в наибольшей степени зависит от той обработки, которой должна подвергнуться программа. Некоторые виды внутреннего представления больше подходят для фиксации структуры компилируемой программы, другие ориентированы на проведение оптимизирующих преобразований, третьи наиболее удобны при синтезе (генерации) результата компиляции. Более подробно внутреннее представление программ в компиляторах рассматривается в разделе 3.3.3 и в пособии “Формальные грамматики и языки. Элементы теории трансляции”.

Фазы оптимизации программ. Оптимизация – важнейшая задача компилятора. Языки высокого уровня, не связанные напрямую с особенностями конкретной аппаратуры, на которой должны выполняться программы, без оптимизации не могут использоваться для создания эффективных программ. Оптимизация программ может проводиться в интересах различных свойств программ. Обычно используют две стратегии оптимизации: оптимизация в целях повышения скорости работы программы и оптимизация в целях уменьшения размеров программ. Методы, используемые при реализации этих стратегий часто противоположны, хотя некоторые из них близки друг к другу. Более подробно проблемы оптимизации программ в компиляторах рассматриваются в разделе 3.3.4.

Фазы синтеза программ. Второй главной работой компилятора является генерация результирующей программы. На выходе компилятора должна быть построена последовательность символов (“*цепочка*”) выходного языка по тем правилам, которые предлагаются языком машинных команд или языком ассемблера. В

случае машинных команд распознавателем этой последовательности символов будет выступать целевая вычислительная система, для которой создается результирующая программа.

Распределение памяти и регистров. Даже в ассемблерах можно встретить фрагменты, выполняющие в том или ином виде распределение памяти и регистров. Тем более, подобные действия по формированию зон или блоков памяти, определению смещений в этих зонах, приписке регистров некоторым элементам данных, необходимы в компиляторах. При проведении таких действий производится компоновка данных в блоки, выравнивание данных на границы физических элементов памяти (байтов, слов, страниц), а также по регистрам специального назначения (векторным, регистрам устройства работы с вещественными числами). Более детально распределение памяти рассматривается в разделе 3.3.5.

Генерация команд и машинно-зависимая оптимизация. Этап генерации команд (кода) проводится в ассемблерах и компиляторах, значительно реже в интерпретаторах. На этом этапе производится окончательное преобразование внутреннего представления транслируемой программы к записи на машинном языке или на языке ассемблера.

В интерпретаторе (точнее в трансляторе со смешанной стратегией трансляции) эта часть заменяется программой, которая интерпретирует внутреннее представление исходной программы. Однако возникновение программы, готовой к интерпретации или выполнению в результате работы только компилятора, возможно не всегда. Многие современные языки, среди которых Си, Си++, Java, предлагают другую концепцию программы, основанную на понятии "*единицы трансляции*". Использование этих языков предполагает, что при запуске компилятора компилируется только некоторая часть полной программы, а остальные части добавляются к ней по мере готовности другими компонентами системы программирования, например, редактором связей.

В подобных случаях интерпретацию программы также нельзя проводить непосредственно после ее компиляции. Необходимо сначала подключить к программе недостающие фрагменты, одни из которых (может быть) надо сначала откомпилировать, а другие (может быть) добавить из набора уже откомпилированных программ, либо из библиотек.

Более детально проблемы генерации кода рассматриваются в разделе 3.3.6.

3.3.1.2. Однопроходный компилятор

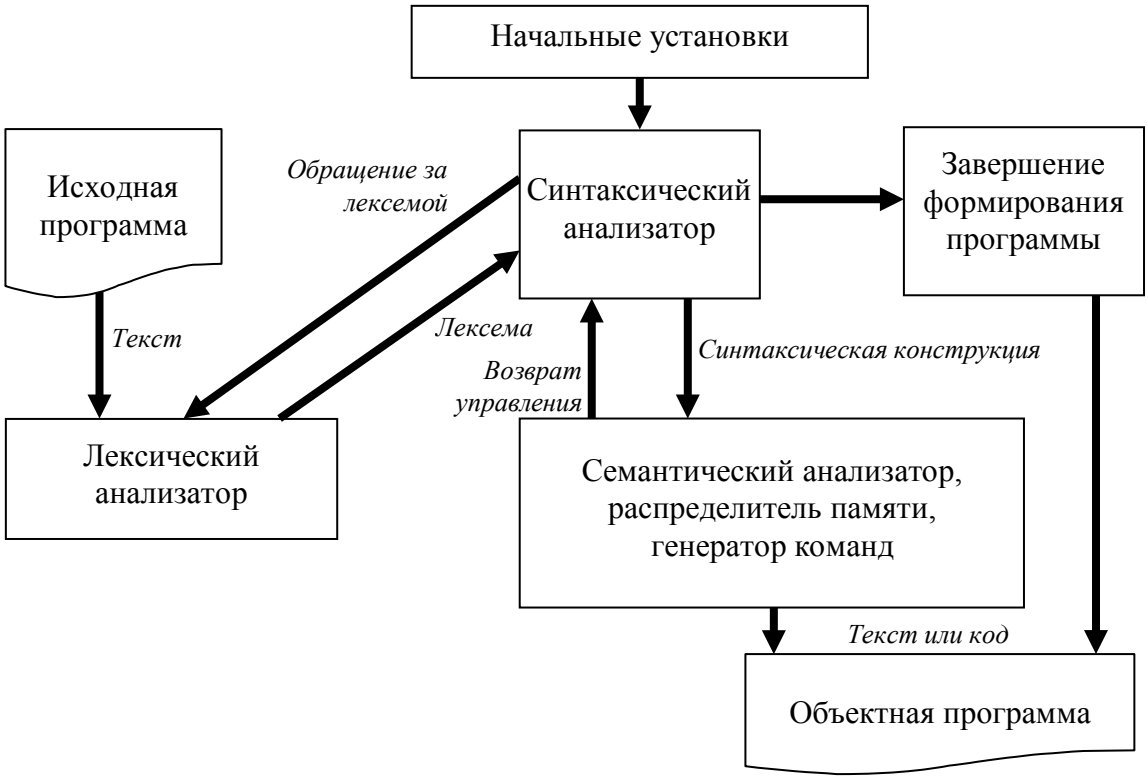
Разобранная схема работы компилятора является концептуальной. Многие компиляторы, однако, построены с отступлениями (иногда значительными) от рассмотренной схемы. Фазы компиляции могут разбиваться на отдельные составляющие или, напротив, объединяться друг с другом. Может даже меняться порядок их выполнения.

Выбор той или иной схемы определяется многими обстоятельствами. Одним из критериев является объем доступной оперативной памяти. Если памяти недостаточно, разработчикам приходится разбивать процесс компиляции на части, передавая информацию от одной части к другой через внешнюю память, в которую записывается промежуточное представление транслируемой программы. Существуют и другие критерии, например, планируемая скорость работы транслятора, степень оптимизации программ.

При выполнении каждого прохода компилятору доступна вся информация, накопленная в информационных таблицах на предыдущих проходах. При выполнении

очередного прохода компилятор может также вновь обратиться к исходному тексту программы. Пользователю становятся доступными только результаты самого последнего прохода – в виде объектной программы, сформированной компилятором, никакие промежуточные результаты компиляции пользователю не видны.

Поскольку процедуры чтения из внешней памяти и записи на внешнюю память имеют относительно невысокую скорость, разработчики компиляторов всегда стремятся уменьшить количество проходов в своих компиляторах. Для языков программирования, которые строились с учетом возможного упрощения процесса трансляции, удается строить такую схему построения компилятора:



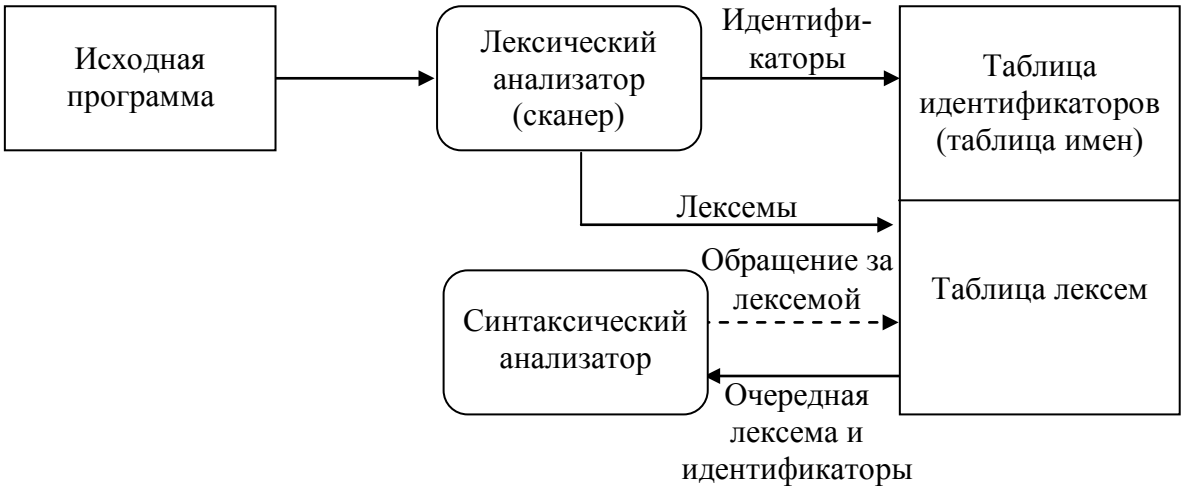
Лексический анализатор в данном случае работает как *сопрограмма* для программы синтаксического анализатора. Аналогично может работать программа семантического анализа и генератора команд.

В современных компиляторах лексический и синтаксический анализаторы – это взаимосвязанные части общего процесса. Возможны два принципиально различных метода организации взаимосвязи лексического и синтаксического анализа – последовательный и параллельный.

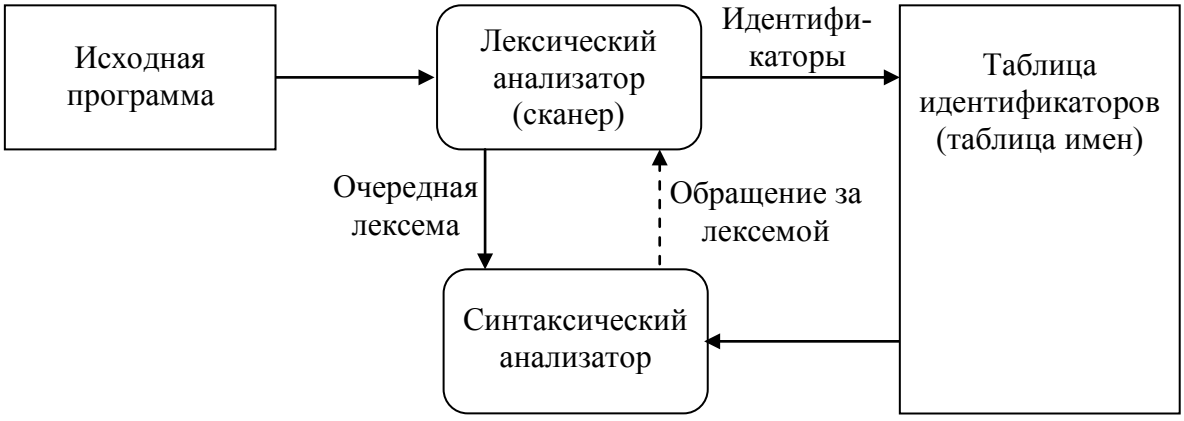
При последовательном варианте лексический анализатор просматривает весь текст исходной программы от начала до конца и преобразует его в таблицу лексем. Таблица лексем заполняется вся и полностью, компилятор использует ее для последующих фаз компиляции, но не изменяет. Если в процессе работы лексический анализатор не смог правильно определить тип лексемы, считается, что программа содержит ошибку. Получающийся в данном случае компилятор никогда не может быть однопроходным.

Последовательная работа лексического и синтаксического анализаторов представляет собой самый простой вариант их взаимодействия. Она проще в

реализации и в определенных условиях обеспечивает более высокую скорость работы компилятора, чем параллельное взаимодействие.



При параллельном варианте лексический анализ текста исходной программы выполняется поэтапно. Лексический анализатор выделяет очередную лексему в исходном тексте и сразу передает ее синтаксическому анализатору. После того, как синтаксический анализатор успешно выполнит разбор очередной законченной конструкции исходного языка (обычно такой конструкцией является отдельный оператор исходного языка), лексический анализатор помещает найденные лексемы в таблицу лексем и в таблицу идентификаторов, а затем продолжает разбор дальше в том же порядке. Параллельное взаимодействие лексического и синтаксического анализаторов строится по такой схеме:



Преобразование входного языка. Вторая задача лексического анализатора есть *выполнение действий*, связанных с обнаружением и распознаванием той или иной лексемы. Фактически ее решение приводит к тому, что с конечным автоматом, лежащим в основе лексического анализатора, ассоциируют не только входной язык, но и выходной. Автомат должен не только распознать правильную лексему на входе, но и породить связанную с ней последовательность символов на выходе. Тем самым, конечный автомат превращается в *конечный преобразователь*. Однако лексический анализатор не может действовать, как простой преобразователь, его задача шире, чем только порождение цепочки символов выходного языка. Он должен уметь выполнять такие действия, как запись выделенной лексемы в таблицу лексем, поиск ее в таблице

идентификаторов или в таблице констант, а также запись нового имени или новой константы в соответствующую таблицу. Часто подобные действия выполняются непосредственно при обнаружении лексемы в исходном тексте.

Выбирая тот или иной способ представления таблиц в программах компилятора, следует руководствоваться следующими требованиями к ним:

- Структура таблиц должна обеспечивать эффективность поиска в таблицах;
- Структура таблиц должна обеспечивать эффективность вставок в таблицы (имеются в виду, как вставки новых элементов, так и вставки новой информации в ранее имевшиеся записи);
- Структура таблиц должна обеспечивать возможность динамического роста объемов таблиц.

Влияет на программу реального лексического анализатора и необходимость отслеживать возможные ошибки в тексте исходных программ. Анализатор должен принимать меры для максимально более полной локализации ошибок, причем не только лексических, но также и синтаксических и семантических.

3.3.2. Задачи семантического анализа

Семантический анализ пользуется всеми результатами предыдущих стадий компиляции. Со стороны лексического анализатора ему передаются все созданные таблицы (идентификаторов, констант и т. д.), а со стороны синтаксического анализатора – результаты синтаксического разбора конструкций языка. Эти результаты представляются на выходе синтаксического анализатора в одной из форм внутреннего представления программ в компиляторе. Обычно на этапе семантического анализа используются некоторые варианты синтаксических деревьев, построенных в результате синтаксического разбора. Такое древовидное представление программы удобно для проведения семантического анализа потому, что для анализа семантики компилируемой программы важно знать именно общую структуру этой программы.

Семантический анализ в свою очередь тоже может разделяться на отдельные стадии. Одна из них вполне может совмещаться с синтаксическим анализом и проводится параллельно с ним. Другая стадия выполняется позднее, когда завершен синтаксический анализ последней конструкции программы и начинается подготовка к генерации объектной программы. Первая часть выполняется после завершения синтаксического анализа очередной конструкции входного языка (процедуры, функции, блока операторов и т. п.) на основе имеющихся в информационных таблицах данных. Вторая часть связана с проведением полного семантического анализа всей программы.

Независимо от выбранного способа реализации основная работа семантического анализатора связана

- с проверкой соблюдения во входной программе семантических соглашений и контекстных условий входного языка,
- с включением во внутреннее представление компилируемой программы дополнительных операторов, связанных с семантикой входного языка,
- с проверкой семантических (смысловых) норм языка, напрямую не связанных с входным языком и его синтаксисом.

Проверки, выполняемые семантическим анализатором, называются *статическими проверками*. Они отличаются от *динамических проверок*, выполняемых в процессе работы программы.

3.3.2.1. Проверка контекстных условий

Проверка семантических соглашений и контекстных условий заключается в сопоставлении входных цепочек исходной программы с требованиями семантики языка программирования. Такие соглашения есть в каждом языке, проверять их на этапе синтаксического анализа невозможно. Обычными видами статических проверок в компиляторах являются

1. *Проверки типов*. Компилятор обязан сообщить об ошибке, если в программе предписано выполнить некоторую операцию с несовместимым с ней операндом, например, произвести сложение указателей.
2. *Проверки управления*. Передача управления за пределы синтаксических конструкций должна производиться только в разрешенные места программы. Например, в языках Си и Си++ оператор выхода из цикла `break` может встречаться только внутри операторов цикла или перебора. Любое другое его употребление должно приводить к сообщению об ошибке.
3. *Проверки единственности*. В определенных ситуациях объект может употребляться только один раз. Например, во многих языках программирования, где есть оператор перебора, все метки в конструкции описания альтернативы `case` должны быть уникальными, элементы в перечислениях `enum` также не должны повторяться.
4. *Проверки, связанные с именами*. Иногда одно и то же имя должно использоваться дважды или большее число раз. Компилятор должен проверять, что во всех местах использовано одинаковое имя. Например, в языке Ада процедура, блок или цикл может иметь имя, которое должно находиться и в начале, и в конце синтаксической конструкции.

К типичным для многих языков контекстным условиям относятся такие семантические ограничения:

- *любое имя*, используемое в программе, должно быть описано, причем только один раз;
- в операторе присваивания *типы переменной и выражения* должны совпадать (либо относиться к некоторым семантически близким типам);
- в условном операторе и в операторе цикла в качестве условия возможно только *логическое выражение*;
- операнды операций отношения должны быть целочисленными (либо иметь какие-либо другие, но точно известные типы);
- тип выражения и *совместимость типов операндов* в выражении определяются по определенным для данного языка правилам; старшинство операций обычно задано синтаксическими правилами;
- *каждая метка*, на которую есть ссылка или переход, должна один раз присутствовать в программе (несколько вхождений одной метки легко проверяются во время анализа синтаксиса);
- единственность описаний идентификаторов рассматривается с учетом *блочной структуры программы*;

- при вызове функции *число* фактических *параметров и их типы* должны соответствовать числу и типам формальных параметров;
- обычно в языке накладываются ограничения на *типы операндов* любой операции, определенной в этом языке; на *типы левой и правой частей* в операторе присваивания; на *тип параметра цикла*; на *тип условия* в операторах цикла и условном операторе и т. п.

Конкретный состав подобных требований жестко связан с семантикой компилируемого языка. Например, требование обязательного описания идентификаторов в некоторых языках отсутствует. Некоторые языки допускают автоматическое преобразование типов несогласованных операндов выражений. Иногда в вызовах процедур и функций допускается указывать не все фактические параметры, остальные параметры получают значения по умолчанию.

По виду (синтаксической записи) большинства операторов нельзя утверждать их семантическую правильность или ошибочность. Семантический анализатор должен фиксировать все случаи нарушения семантических соглашений и условий, найденные им в программе, и выдавать сообщения о семантических ошибках в программе.

3.3.2.2. Дополнение внутреннего представления

Многие языки запрещают смешивать в выражениях операнды даже “близких” типов, например, использовать рядом операнды вещественных и целочисленных типов, знаковых и беззнаковых типов и т. д. Однако большинство реальных языков программирования допускает некоторые вольности при записи выражений, гарантируя, что в компиляторах будут предприняты меры по согласованию типов. Абсолютно аналогичная работа выполняется при обработке операций обращения к функциям и процедурам с параметрами.

Задача семантического анализатора состоит в поиске всех мест, где нужно выполнить подобные преобразования типов и вставить во внутреннее представление явные команды преобразования. Иногда эти преобразования тривиальны и состоят из одной-двух команд, а иногда они выполняются встроенными библиотечными функциями.

Преобразования типов могут производиться не только по отношению к простым арифметическим данным. Во многих языках существуют данные, позволяющие ссылаться на другие данные – указатели и ссылки. Иногда операция доступа к данным по указателю приводит к введению в программу целой серии команд по настройке внутренних регистров, выделению фрагментов данных, размеры которых не кратны целому числу машинных слов или байтов и т. д.

Все это означает, что действия, выполняемые при семантическом анализе компилируемой программы, существенным образом влияют на порождаемые компилятором объектные программы.

3.3.2.3. Проверка правил программирования

Многие современные компиляторы не только проверяют ограничения и требования, выставляемые семантикой языка, но также выполняют дополнительные проверки, способствуя выработке “правильного” стиля программирования. Во многих языках программирования действуют правила хорошего программирования.

- Каждая переменная или описанная константа программы должна быть использована в программе хотя бы один раз.

- Каждая переменная программы должна получить значение до своего первого использования (например, до использования в правой части оператора присваивания).
- Результат вычисления функции должен быть определен при любом ходе ее выполнения (например, независимо от выбора исполняемых ветвей в условных операторах).
- В программе не должно быть невыполняемых в принципе операторов. Каждый оператор программы должен иметь потенциальную возможность выполниться хотя бы один раз.
- Условные операторы (всех разновидностей) должны предусматривать возможность хода выполнения программы по любой из своих ветвей.
- Операторы цикла должны предусматривать возможность завершения цикла.

3.3.2.4. Разнесение имен по пространствам именования

Распределение использованных в программе идентификаторов по пространствам именования есть одна из важных задач этапа семантического анализа. Требования различных языков программирования, предъявляемые к именам используемых в программах объектов, распространяются на имена объектов, относящиеся к одним пространствам именования и видимости. С одной стороны это означает, что объекты нельзя использовать вне тех блоков, где они описаны и видны, с другой стороны, это ослабляет требования уникальности имен, поскольку объектам, относящимся к разным блокам, можно давать одинаковые имена, не внося при этом путаницы в программу.

На этапе лексического анализа разрешить все эти проблемы абсолютно невозможно – там уникальность имен определяется только их записью как последовательности букв, цифр и других разрешенных символов. На более поздних стадиях компиляции одинаково записываемые имена необходимо заменять на новые уникальные в пределах всей программы. В частности,

- имена локальных объектов блоков дополняются именами блоков (функций, процедур), в которых они описаны,
- имена внутренних (в терминах языка Си++ – статических) переменных и функций модулей программы дополняются именами самих этих модулей,
- имена процедур и функций, принадлежащих классам в объектно-ориентированных языках (Си++) или вложенных в другие процедуры и функции в процедурных языках дополняются именами этих классов или процедур,
- имена методов в описаниях классов дополняются именами, строящимися в зависимости от числа и типов их формальных параметров, это же относится и к именам перегруженных функций Си++.

Особой заботой компиляторов являются имена внешних (глобальных) объектов, которые остаются видимыми в объектной программе и могут обрабатываться другими компонентами систем программирования, например, редакторами связей. Для таких имен в конкретных системах программирования могут существовать собственные соглашения именования. Такие имена должны быть уникальными на уровне библиотек, в которые они могут попадать после завершения компиляции.

3.3.3. Внутреннее представление программ

Дополнительную сложность задаче компиляции придает тот факт, что из-за серьезных различий между входным и выходным языками провести непосредственное преобразование из одного языка в другой не всегда представляется возможным. Даже синтаксический анализ исходного текста приходится делать поэтапно, разбивая его на лексическую и собственно синтаксическую часть. Это позволяет использовать на каждом этапе свои грамматические правила разбора, существенно упрощая реализацию распознавателей, применяемых на этих этапах. Именно поэтому часто в компиляторах на некоторых стадиях обработки программ возникает некоторое промежуточное *внутреннее представление* компилируемой программы, которое лишь на завершающей стадии преобразуется в представление программы на выходном языке компилятора.

К основным свойствам языков внутреннего представления программ можно отнести такие:

- языки внутреннего представления позволяют фиксировать синтаксическую структуру исходной программы;
- текст на языках внутреннего представления можно автоматически генерировать во время синтаксического анализа;
- конструкции языков внутреннего представления должны относительно просто транслироваться в объектный код, либо достаточно эффективно интерпретироваться.

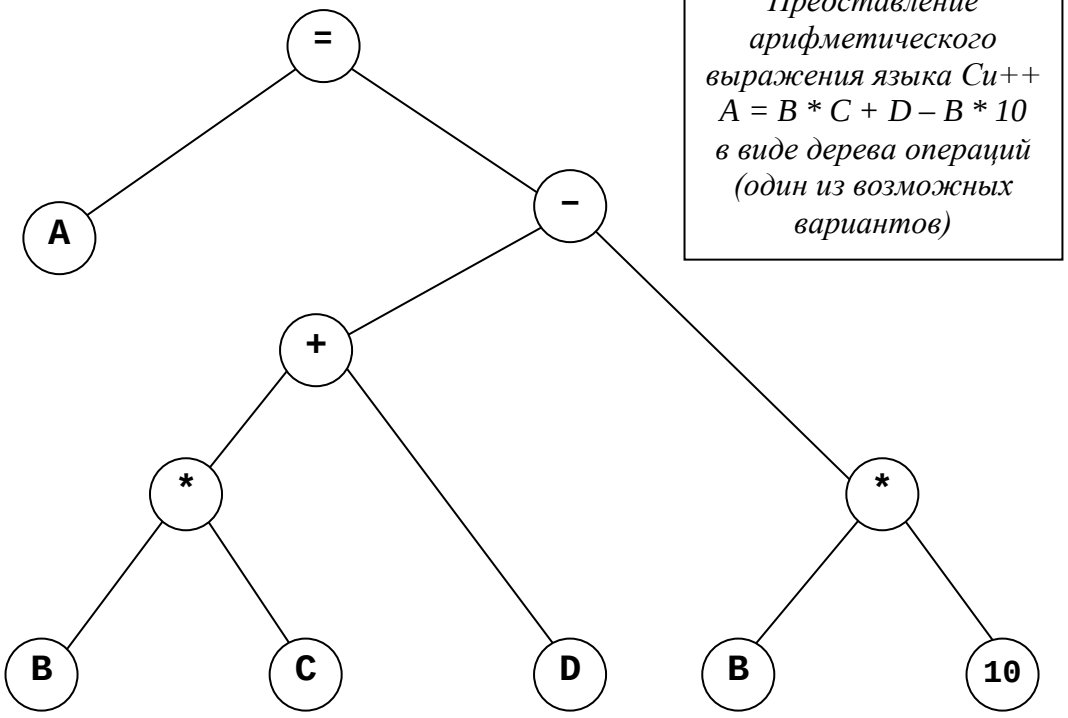
Легче всего синтаксическому анализатору организовать внутреннее (а для себя – выходное) представление в виде дерева синтаксического разбора. Однако дерево разбора содержит огромное количество избыточной информации: в нем присутствуют даже нетерминальные символы, не несущие никакой полезной информации. Что должно присутствовать во внутреннем представлении – это операторы и операнды исходной программы, записанные в более удобной для последующей обработки форме. Удобство это заключается, прежде всего, в том, чтобы было легко отличить оператор от операнда, в каком бы порядке они друг за другом ни следовали. В основе каждого из видов внутреннего представления программ лежит некоторый метод представления синтаксического дерева. В реальных компиляторах применяются следующие общепринятые способы внутреннего представления программ:

- связные списочные структуры, представляющие синтаксическое дерево;
- многоадресный код с явно именуемыми результатами (тетрады – ***оператор, операнд, операнд, результат***);
- многоадресный код с неявно именуемыми результатами (триады – ***оператор, операнд-результат***, ***операнд***), иногда этот способ представления модифицируется и используется последовательность косвенных триад (сами триады хранятся в отдельной таблице, формируемая программа представляет собой последовательность ссылок на таблицу);
- инфиксная запись (операции записываются между своими операндами, как в обычной записи арифметических выражений);
- префиксная запись (операции записываются перед своими операндами, как в записи вызова процедуры или функции с параметрами);
- постфиксная запись (операции записываются после своих операндов);
- язык ассемблера целевой или абстрактной машины.

В одном компиляторе может использоваться любая из этих форм, обычно выбираются несколько разных видов внутреннего представления программ для разных стадий компиляции. Если от компилятора не требуется проведения серьезной оптимизации программ, выбирается схема, в которой синтаксический разбор, семантический анализ и генерация объектной программы совмещены в одном проходе компилятора. В такой схеме внутреннее представление программы существует только на концептуальном уровне, выражаясь только в последовательности шагов преобразований.

Связные списочные структуры. Списки – это структуры представления программ, которые можно наиболее просто и эффективно строить на этапе синтаксического анализа. Обычно списки используются для представления синтаксических деревьев – таких структур, вершинами которых являются операции, а листьями – операнды. Как правило, листья связаны с записями в таблицах идентификаторов и констант. Структура синтаксических деревьев наиболее точно отражает синтаксис языка программирования, на котором была написана исходная программа. Обычно перед построением синтаксических деревьев в грамматиках языка избавляются от цепных правил вида $A \rightarrow B$, где A и B –нетерминальные символы.

В тех случаях, когда синтаксическому дереву соответствует последовательность операций, порождающая в результате команды объектной программы, такое дерево называют *деревом операций*. Дерево операций строится непосредственно и автоматически из дерева вывода, порождаемого синтаксическим анализатором. Для этого из него удаляются цепочки нетерминальных символов, а также узлы, не несущие семантической нагрузки при генерации объектной программы (например, скобки, меняющие порядок операций):



*Представление арифметического выражения языка Си++ A = B * C + D – B * 10 в виде дерева операций (один из возможных вариантов)*

Грамматика исходного языка не имеет никакого влияния на то, какой узел в дереве будет операцией, а какой операндом. Влияние на это имеет не синтаксис, а семантика языка программирования.

Деревом операций обычно пользуются на внутренних стадиях компиляции, предшествующих генерации объектной программы. Оно отражает общую структуру программы и связь операций между собой. Имея представление программы в виде дерева, удобно проводить преобразования, связанные с перестановкой фрагментов программы и переупорядочением операций. Многие оптимизирующие компиляторы выбирают именно древовидные структуры для внутреннего представления программ.

Недостатком деревьев является сложность их преобразования в линейную последовательность команд объектной программы.

Многоадресный код с явно именуемым результатом (тетрады). Тетрады представляют собой запись операций в форме четырех составляющих: операции, двух операндов и результата операции:

<операция>(<операнд1>,<операнд2>,<результат>)

Тетрады составляют линейную последовательность команд, как в следующей записи арифметического выражения `A=B*C+D-B*10` в виде последовательности тетрад:

1	*	B	C	T1
2	+	T1	D	T2
3	*	B	10	T3
4	-	T2	T3	T4
5	=	T4	?	A

При вычислении выражения, записанного в форме тетрад, они вычисляются последовательно одна за другой, без каких-либо приоритетов. Если какой-то из операндов (или оба) в тетраде отсутствует (как в унарной операции), он заменяется признаком пустого операнда. Результат тетрады никогда опущен быть не может. Порядок вычисления тетрад может быть изменен только явно с помощью специальных тетрад, вызывающих переходы по последовательности тетрад вперед или назад.

Тетрады записываются в линейной последовательности, поэтому их легко преобразовать в последовательность команд объектной программы, либо в язык ассемблера, но в отличие от языка ассемблера, тетрады не зависят от архитектуры вычислительной системы, для которой ведется компиляция, и являются машинно-независимым представлением программ. Некоторые сложности при работе с тетрадами могут возникать потому, что у вычислительных машин редко встречаются трехадресные системы команд.

Многоадресный код с неявно именуемым результатом (триады). Триады представляют собой запись операций в форме из трех составляющих: операции и двух операндов:

<операция>(<операнд1>,<операнд2>)

Особенностью триад является то, что один или оба операнда в триаде могут быть ссылками на другую триаду в том случае, если в качестве операнда данной триады выступает результат выполнения другой триады, например, так в виде триад можно записать арифметическое выражение `A=B*C+D-B*10`:

1	*	B	C
2	+	^1	D
3	*	B	10

```
4  -    ^2    ^3
5  =    A     ^4
```

Ссылка на триады в реальных компиляторах обычно реализуется настоящим указателем, что делает триады списочной структурой и облегчает процессы преобразования триад. Тем самым, назвать триады полностью линейной структурой нельзя. В то же время триады можно рассматривать и как линейную последовательность, если результаты вычислений операций триад хранить во временно выделяемой памяти. Такое свойство триад приводит к необходимости использования в компиляторах специальных алгоритмов распределения памяти для хранения промежуточных результатов, поскольку в отличие от тетрад, какие-либо явные временные переменные в триадах не используются.

Триады требуют меньше памяти для представления программ и имеют в этом преимущество перед тетрадами, к тому же триады ближе к двухадресным машинным командам, чем тетрады, поэтому их легче преобразовывать к окончательному виду объектной программы. В особенности триады удобны для трансляции в объектный код таких вычислительных машин, в командах которых первый операнд часто хранится в одном из регистров.

Инфиксная запись. Запись операций и операндов в традиционном виде применяется только в программах, подаваемых на вход компиляторов. Это наиболее удобная для людей, но наименее удобная для автоматической обработки запись.

Префиксная запись. Префиксная запись иначе называется польской записью или прямой польской записью. В такой записи операторы (операции) предшествуют своим операндам. Некоторое неудобство прямой польской записи, которое привело к использованию обратной записи, состоит в том, что операторы в ней следуют не в том порядке, в каком они должны выполняться в вычислительной машине:

```
= A - + * B C D * B 10
```

Постфиксная (инверсная, обратная, суффиксная) запись. В отличие от прямой польской записи инверсная польская запись (ПОЛИЗ) обладает следующими важными свойствами:

- Идентификаторы в обратной польской записи следуют в том же порядке, в каком они следуют в инфиксной записи.
- Операторы в обратной польской записи следуют в том порядке, в каком они должны вычисляться (слева направо).
- Операторы следуют непосредственно за своими операндами.

Обратная польская запись является наиболее удобным видом представления программ в компиляторах. Эта запись не требует учитывать приоритеты операций, в ней не используются скобки. ПОЛИЗ наиболее удобна при трансляции арифметических выражений:

```
A B C * D + B 10 * – =
```

Обычно в компиляторах для представления программ в виде ПОЛИЗ разрабатывается специальное представление не только арифметических операций, но и всех других исполняемых операторов, что позволяет полностью автоматизировать процесс преобразования стандартного представления на исходном языке

программирования в ПОЛИЗ. У этой записи имеются и недостатки. Программы, представленные в виде ПОЛИЗ трудно поддаются анализу, поэтому оптимизирующие компиляторы создают ПОЛИЗ после проведения глобальных оптимизирующих преобразований, когда требуется преобразовать уже оптимизированную древовидную структуру в линейную. Работа с обратной польской записью в компиляторах подробно рассмотрена в пособии “Формальные грамматики и языки. Элементы теории трансляции”.

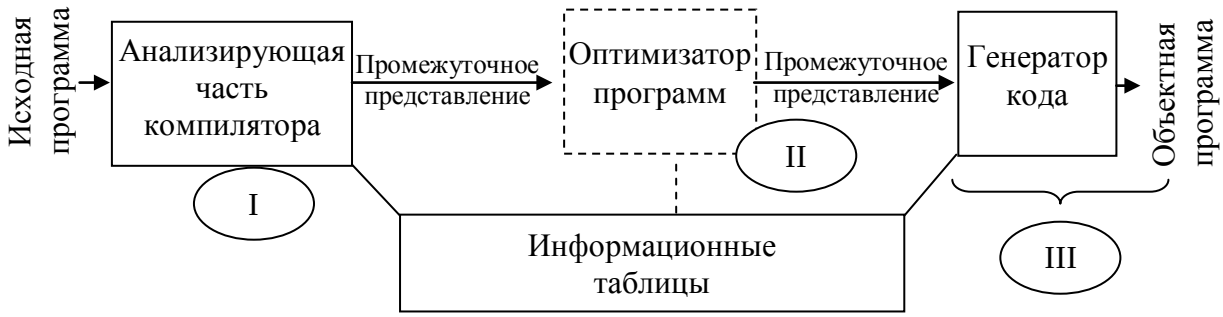
Язык ассемблера и машинные команды. По определению компилятора, как частного случая транслятора с языка программирования высокого уровня на машинный язык или язык ассемблера, понятно, что представление программы в виде машинных команд или ассемблерной записи является обязательным. Однако некоторые компиляторы преобразованием программ к такому виду не заканчивают свою работу, а лишь продолжают ее. Именно в таком виде наиболее удобно проводить машинно-зависимую оптимизацию, при которой легче всего учесть семантические особенности выполнения отдельных связок команд и получить дополнительный выигрыш в производительности программы.

Иногда в компиляторах, даже очень сложных и проводящих глубокую оптимизацию, тоже возникает потребность на некотором этапе преобразования транслируемой программы перевести программу к виду, более приближенному к ее окончательному представлению. При этом запись на языке ассемблера может не сразу оказаться наиболее удобным представлением. В таких случаях часто используют представление программы с помощью ***псевдокода***. От языка ассемблера он отличается тем, что в нем могут не приниматься во внимание некоторые архитектурные особенности целевой машины, в частности, для псевдокода является обычным предположение о том, что объектная машина обладает неограниченной памятью и бесконечным числом регистров общего назначения. В псевдокодах более вольно используются форматы генерируемых команд, а все уточнения об ограничениях делаются на этапах распределения памяти и регистров, а также при проведении машинно-зависимой оптимизации, когда уже точно становится известна окончательная последовательность команд, реальные номера регистров и адреса операндов в памяти.

3.3.4. Оптимизация в компиляторах

Переход от трансляции всей программы как целого к трансляции последовательности относительно независимых операторов (в синтаксическом плане в контекстно-свободных грамматиках операторы действительно не зависят друг от друга) приводит к потере информации о взаимосвязи этих операторов. Описанные ранее методы анализа программ и генерации этих программ на других языках позволяют решить главную задачу компиляции – отыскать эквивалентное представление исходной программы в терминах выходного языка. Вторая задача – поиск *эффективного* эквивалента серьезно отличается от первой и требует других подходов и методов решения. Под оптимизацией программ имеется в виду обработка, связанная с переупорядочением и изменением операций в компилируемой программе в целях получения более эффективной объектной программы.

Оптимизация программ – вынужденная мера, прибегать к которой приходится потому, что компилятор не в состоянии выполнить семантический анализ всей исходной программы как единого объекта, оценить и понять смысл программы. Оптимизация программ проводится в компиляторах в различных местах:



Первичную оптимизацию может проводить сам пользователь (пометка *I*). Некоторые системы программирования предлагают поддержку пользовательской оптимизации, в частности, имеют в своем составе профилировщики, помогающие выявить те фрагменты программ, которые для своего выполнения требуют максимальной доли времени работы программы. В целом оптимизация на уровне исходной программы может дать наибольший эффект для улучшения технических характеристик программы. К таким техническим характеристикам относят обычно две: объем памяти, необходимый для выполнения объектной программы (для хранения данных и самих команд программы), и скорость выполнения программы (ее быстродействие). Очень часто сокращение используемых в программе данных приводит к увеличению времени работы программы, а попытки повысить быстродействие приводят к увеличению используемой памяти. Поэтому часто для оптимизации выбирается один, главный критерий, либо некоторый интегрированный критерий, основанный на сбалансированном подходе к достижению многих целей одновременно.

При выборе используемых в компиляторе оптимизирующих преобразований руководствуются следующими критериями:

- все преобразования должны быть *эквивалентными* (для всех наборов данных, даже неправильных). Эквивалентные преобразования сохраняют семантику исходной программы;
- “стоимость” преобразования должна быть сопоставима с затрачиваемыми усилиями и полученными эффектами. Время компиляции от включения оптимизирующих преобразований всегда растет, но иногда это не очень важно. Важнее, чтобы при оптимизации не вносились дополнительные ошибки, на исправление которых затрачиваются дополнительные усилия,
- в результате преобразований программы в среднем должны “улучшаться” (почти для всех допустимых данных), лишь на каких-то (редко встречающихся) комбинациях данных допускается обратный эффект (ухудшение характеристик).

Некоторые преобразования считаются удовлетворяющими этим критериям и рекомендуются для включения в любой компилятор. Другие преобразования, основанные на сложном анализе потока управления и потока данных, а также на использовании результатов этого анализа при модификации программ, требуют дополнительного анализа представительного набора реальных программ на данном языке программирования. Различаются два основных вида оптимизации:

- *Машинно-независимая оптимизация*, то есть проведение преобразований исходной программы (в форме некоторого внутреннего представления), не зависящих от выходного языка компилятора (без учета конкретных свойств объектной машины). Эта оптимизация обычно выполняется на специально выделенной фазе компиляции (пометка *II* на рисунке),
- *Машинно-зависимая оптимизация*, то есть преобразование программы на выходном языке компилятора. Этот вид оптимизации обычно проводится одновременно с генерацией объектной программы или уже после этой генерации на дополнительной стадии (пометка *III* на рисунке).

Машинно-независимая оптимизация получила такое название потому, что проводимые в рамках этого процесса преобразования не зависят от архитектуры вычислительной системы, для которой предназначена объектная программа. Для проведения таких преобразований разработан целый ряд формальных математических методов. При проведении преобразований машинно-зависимой оптимизации может оказаться необходимым учитывать аппаратные особенности вычислительных систем – число и способ организации взаимодействия центральных процессоров, иерархию устройств памяти, количество и размеры регистров, а также многое другое.

Обычно оптимизирующим преобразованиям подвергается внутреннее представление программы, а не текст на исходном языке. Во-первых, операции, необходимые для реализации высокоуровневых операций становятся на языках внутреннего представления программ более явными, что облегчает их обнаружение и оптимизацию. Например, исходный оператор `s=s+a[i]*b[i]` скрывает, что вычисление адресов для элементов `a[i]` и `b[i]` содержит общие подвыражения `sizeof(тип)*i`. Во-вторых, внутреннее представление может оказаться относительно независимым от объектной машины, что делает оптимизатор достаточно устойчивым к изменениям при переносе на другую машину.

3.3.4.1. Машинно-независимая оптимизация

Основные преобразования машинно-зависимой оптимизации выполняются для отдельных выражений, линейных участков программ, циклов, вызовов процедур и функций.

Оптимизация однократно выполняемых участков программы практически не оказывает влияния на быстродействие программы и может сказываться только на объеме занимаемой программой памяти, поэтому наиболее тщательно всегда оптимизируется самый внутренний цикл программы. Для проведения оптимизации программы делят на ***линейные участки***, то есть на *выполняемые по порядку последовательности операций*. Линейные участки имеют *один вход и один выход*. Линейные участки имеются в любой программе и чаще всего содержат последовательности вычислений, состоящие из арифметических выражений и операторов присваивания значений переменным программы. Ни одна операция линейного участка не может быть выполнена большее число раз, чем смежные с нею операции. Для ***линейных участков*** проводятся следующие преобразования:

- вычисление выражений из констант на стадии компиляции,
- арифметические преобразования,
- устранение общих подвыражений (избыточных вычислений),

- удаление ненужных присваиваний и других операций, распространение копий значений,
- перестановка независимых смежных участков программ,
- удаление недостижимых фрагментов программы,
- оптимизация вычисления логических выражений.

Вычисление выражений из известных операндов (свертка операций):

- непосредственное использование констант программистом:

```
A = sin (2 * 3.14 * B);
```

- возникновение констант-операндов после макрорасширений,

```
#define Pi 3.1415926
A = sin (2 * Pi * B);
```

- возникновение констант-операндов в результате компиляции языковых конструкций, например, многомерных массивов:

```
int a [10][10][10], b [10][10][10], c [10][10][10];
a [3][4][i] = b [8][3][k] * c [3][2][j];
a' [((3 * 10) + 4) * 10 + i] :=  b' [((8 * 10) + 3) * 10 + k] *
                                 c' [((3 * 10) + 2) * 10 + j];
```

Компилятор должен выполнить вычисления и внести записи о новых литеральных константах в таблицу констант, как если бы эти константы были введены самим программистом. Более сложные варианты алгоритмов свертки принимают во внимание известные им значения переменных (например, сразу после присваивания) и даже функций.

Арифметические преобразования. Компилятор может изменять характер и порядок следования операций на основании известных алгебраических и логических тождеств, например, заменять выражение `A=B*C+B*D` выражением `A=B*(C+D)`. Некоторые операции могут заменяться более "простыми", что делает их выполнение более эффективным:

```
x := y ** 2        =>     x := y * y;
x := y * 2         =>     x := y + y;
```

Устранение общих подвыражений (избыточных вычислений). Операция линейного участка может оказаться избыточной, если ранее на этом же линейном участке уже выполнялась идентичная операция, и никакой операнд данной операции не был изменен в промежутке между двумя идентичными операциями.

Удаление ненужных присваиваний и других операций. Если на некотором линейном участке между двумя операциями присваивания какой-либо переменной значений (одинаковых или разных, не имеет значения), не было ни одного оператора, в котором использовалось бы первое значение переменной, это присваивание является бесполезным и может быть удалено из программы без изменения ее смысла.

Иногда по наличию операторов присваивания одним переменным значений других переменных удается исключить использование некоторых переменных, заменив их использованием их копий (метод носит название "распространение копий"). В таких случаях происходит как экономия времени исполнения программ, так и экономия

памяти, отведенной под хранение данных программы. Например, после присваивания `f:=g` можно вместо переменной `f` использовать переменную `g`, а присваивание просто исключить из программы. В наилучшем случае переменная `f` совсем станет ненужной, значит, и память для нее тоже распределять не придется. Подобные преобразования становятся особенно актуальными при компиляции автоматически сгенерированных программ, работающих с многочисленными переменными и "цепными" присваиваниями

Перестановка независимых смежных участков программ. Иногда компиляторам удается таким образом переставить следующие друг за другом операции, что без изменения смысла программы удается применить какие-либо другие преобразования. Например, имея выражение `A=2*B*3*C` можно преобразовать его перестановкой в `A=(B*C)*(2*3)`, а затем можно вычислить значение подвыражения из констант. Даже если выражение из констант получить не удается, перестановка операций может привести к экономии временных переменных, которые порождаются компилятором для хранения промежуточных результатов вычислений. Например, непосредственное вычисление выражения `A=(B+C)+(D+E)` может потребовать, по крайней мере, одной временной переменной для хранения промежуточного результата сложения `B` и `C`. Если же провести перестановку операций, эта переменная будет не нужна, а результат останется прежним: `A=((B+C)+D)+E`.

В некоторых случаях перестановка операций может приводить к потере точности вычислений. Некоторые типы операндов могут сделать переупорядочение выражений невозможным. Например, перестановка целочисленных операций в выражении `I/J*K` может привести к неверному вычислению выражения `10*3/8`.

Удаление недостижимых фрагментов программы часто требует глобального анализа программы для определения "достижимости". В основе такого анализа лежат идеи теории графов, связанные с анализом потока управления и потока данных. Простейший вариант – удаление недостижимых фрагментов, выделяемых с помощью препроцессорных операторов.

Оптимизация вычисления логических выражений. Особенностью оптимизации логических выражений является то, что для получения их значений не всегда требуется проводить вычисление всего выражения до конца. Иногда по результату первых же операций можно заранее определить окончательный результат. Например, операцию логического сложения можно не проводить, если известно, что один из ее операндов имеет значение "истина". Если это разрешается правилами языка, компиляторы так строят внутренние представления логических выражений, чтобы их вычисления прекращались сразу же, как только значение всего выражения становится предопределенным. Аналогичные рассуждения относятся и к арифметическим выражениям, но умножения на `0` встречаются гораздо реже, чем логическое "И" со значением "ложь".

Оптимизация ***передачи параметров и вызовов функций*** проводится на основе двух подходов: прямой подстановки тел функций в основной текст программы и передачи параметров не с помощью общего стекового механизма, а через глобальные переменные, которые впоследствии связываются с регистрами центральных процессоров.

Прямая подстановка функций в основной цикл программы может привести к существенному увеличению скорости работы программы, но одновременно и к увеличению размеров программы (если функция вызывается из нескольких разных

мест). Этот метод приводит к сокращению времени на передачу параметров и возвращаемого результата, на команды передачи управления, захвата памяти в стеке и другие вспомогательные операции. Некоторые языки (Си++) разрешают программистам явно указывать функции, которые желательно реализовать подстановкой (`inline`).

Передача параметров через регистры процессора относится к машинно-зависимой оптимизации. Отказ от универсального стекового механизма в определенных случаях может приводить к значительному снижению времени работы программы, но передача параметров через регистры зависит от количества доступных регистров в вычислительной системе и от используемого в компиляторе алгоритма их распределения.

Важность данного метода постоянно возрастает с ростом возможностей вычислительной аппаратуры, но метод имеет ряд выраженных недостатков. Во-первых, он сильно зависит от особенностей конкретной архитектуры вычислительной машины. Во-вторых, процедуры, оптимизированные таким образом, не могут включаться в общие библиотеки, поскольку используют нестандартный метод получения параметров. В-третьих, использовать метод не удается, если в теле функции используются операции вычисления адресов параметров.

В некоторых языках программирования программисты имеют возможность явно указывать, какие переменные следует размещать на регистрах (в Си++ – с помощью ключевого слова `register`). Подобные указания имеют для компиляторов рекомендательный характер, но часто помогают получить хорошо оптимизированные программы.

Оптимизация циклов. *Циклом* в программе называется любая последовательность участков программы, которая может выполняться повторно. Циклы необязательно должны оформляться с помощью операторов цикла, и чтобы их обнаружить, используется граф управления программы. Обычно циклы содержат в себе один или более линейных участков, где производятся вычисления, поэтому для них могут применяться все методы оптимизации линейных участков. Для оптимизации циклов разработаны и специальные методы:

- вынесение инвариантных вычислений из тела цикла,
- замена операций с переменными цикла,
- слияние, расщепление и развертывание циклов.

Вынесение инвариантных вычислений из тела цикла сводится к вынесению за пределы цикла тех операций, операнды которых не изменяются в процессе выполнения цикла. Например, цикл

```
for (i = 0; i < limit – 2: i ++) A [i] = B * C * A [i];
```

может быть заменен последовательностью операций

```
D = B * C; k = limit – 2;
for (i = 0; i < k: i ++) A [i] = D * A [i];
```

при условии, что значения `B` , `C` и `limit` не изменяются в теле цикла. При этом умножение `B*C` будет выполнено только один раз, а не 10, как в исходном варианте.

Для вычислительных машин с векторной архитектурой оптимизация циклов становится машинно-зависимой. Например, для некоторых векторных архитектур

снижение времени выполнения программы иногда можно получить, не проводя вынесение вычислений из циклов, а внося их туда: в таких архитектурах оказывается эффективнее провести повторные вычисления с помощью векторных регистров, чем нарушать работу векторного конвейера выполнением операции со скалярной переменной.

Замена операций с переменными цикла производится на основе понимания того, что с каждым шагом цикла значение переменной цикла меняется на один "*шаг цикла*". Многие алгоритмы устроены так, что в ходе вычислений некоторые величины оказываются пропорциональными номеру итерации. При этом в программах производится умножение этих величин на значение переменной цикла. Такие переменные называются индуктивными. Все такие переменные заменяются компилятором на одну, значения других вычисляются с помощью коэффициентов. Например, последовательность операторов

```
S = 10; for (i = 0; i < N; i ++) A [i] = i * S;
```

может быть заменена последовательностью операций

```
S = 10; T = 0; for (i = 0; i < 10; i ++) T = T + S, A [i] = T;
```

Вычисление значения `A[i]` тоже может потребовать индуктивной переменной, так как изменение значения переменной цикла на `1` может приводить к изменению адреса элемента массива на величину `sizeof(A[0])`, значит, `&A[i]` ≡ `A+sizeof(A[0])*i`.

В тех вычислительных системах, в которых время выполнения операции умножения превышает время выполнения сложения, удается добиться немалого эффекта. Иногда оказывается возможным отказаться от переменной цикла, как в следующем примере:

```
S = 10; for (i = 0; i < N; i ++) R = R + F (S), S = S + 10;
```

В этом примере две индуктивные переменные, но переменную цикла можно просто исключить:

```
S = 10; M = S + N * 10; while (S <= M) R = R + F (S), S = S + 10;
```

Таким преобразованием за счет введения дополнительной переменной `M` удалось исключить `N` операций сложения для переменной `i`.

Слияние и развертывание циклов – это два различных варианта преобразований: слияния двух смежных или вложенных циклов в один и замена цикла на последовательность операций (часто линейную). Слияние смежных циклов с независимыми внутренними операторами S_1 и S_2 позволяет снизить накладные расходы на организацию циклической работы:

```
for (i = 0; i < n; i ++) { S1 }
for (i = 0; i < n; i ++) { S2 }
```

⟹

```
for (i = 0; i < n; i ++) { S1; S2 }
```

Замену циклов последовательностями операций можно выполнять для циклов, кратность которых известна уже на стадии компиляции.

Расщепление цикла может оказаться полезным в случае наличия в теле цикла условных операторов:

```
for (i = 0; i < n; i ++)                    if (x < y)
    { if (x < y)    { S1; }                     for (i = 0; i < n; i ++) { S1; }
      else          { S2; }   }         else  for (i = 0; i < n; i ++) { S2; }
```

Развертывание цикла позволяет в определенных случаях уменьшить число проверок условий завершения цикла, а также создать предпосылки для последующего распараллеливания выполнения операций:

```
for (i = 0; i < n; i ++)                    for (i = 0; i < n; i += 2)
    { A [i] = B [i] * C [i]; }                  { A [i] = B [i] * C [i];
                                                  A [i+1] = B [i+1] * C [i+1]; }
```

Кажущиеся правильными преобразования не всегда ведут к построению эквивалентной программы. Например, цикл

```
for (i = 1; i < 100; i ++) { … A = i * B; … }
```

может быть преобразован к виду:

```
for (i = 1; i < 100; i ++) { … A = A + B; … }
```

однако это преобразование будет правильным только для целочисленных переменных *A* и *B*. Если в теле цикла использованы вещественные переменные, то при их последовательном суммировании может накапливаться погрешность, которая при значительном числе итераций может стать недопустимо большой.

3.3.4.2. Машинно-зависимая оптимизация

Машинно-зависимые методы оптимизации ориентированы на конкретную архитектуру вычислительной системы, то есть на совокупность аппаратных и программных составляющих, а также взаимосвязи между ними. Некоторые аспекты методов машинно-зависимой оптимизации имеют общий характер и применяются многими разработчиками. К таким аспектам относятся:

- учет регистровой структуры вычислительной аппаратуры,
- удаление излишних команд,
- оптимизация потока управления и удаление недостижимых участков программ,
- снижение “стоимости” программы,
- использование машинных идиом,
- слияние, дробление и развертывание циклов, иногда требующееся из-за технических особенностей аппаратуры,
- учет векторных и конвейерных свойств архитектуры.

Одним из важнейших аспектов является учет распространенной особенности многих вычислительных архитектур, строящихся на программно доступных регистрах. Среди этих регистров одни могут быть специально выделены для выполнения определенных задач (управление стеком), а другие представляют собой *регистры общего назначения*. Выполнение операций над регистрами производится существенно быстрее, чем над элементами памяти, к тому же часто над элементами памяти, кроме

операций пересылки, вообще нельзя выполнять никаких операций, а требуется предварительная загрузка их содержимого на регистры. Все это ставит перед разработчиками почти всех компиляторов ***задачи распределения регистров и оптимизации их использования***. Эта задача в общем случае является NP-полной, но в каждом конкретном случае удается найти приемлемое решение.

Простейшим методом распределения регистров является их *"жесткое" распределение*, например, только для хранения фактических параметров процедур и/или важнейших переменных. Такой выбор, с одной стороны, упрощает разработку компилятора, с другой стороны, ограничивает эффективность использования регистров.

Более сложным является распределение *на основе анализа графа потока управления*. Граф потока управления строится из узлов, которыми являются базовые блоки программы (последовательности команд, имеющие один вход и один выход), и дуг, соответствующих переходам от одного базового блока к другому при наличии некоторых входных для базового блока данных. Результаты вычисления некоторых выражений, вычисляемых в базовых блоках, оказываются при этом внутренними (промежуточными), некоторые другие результаты переходят в смежные блоки. Такие результаты и пытаются хранить на регистрах.

Распределение на основе раскраски графа взаимодействия регистров проводится так:

- число регистров полагается равным числу переменных в программе.
- два узла (регистра) соединяются дугой, если два регистра должны хранить некоторые значения одновременно.
- граф раскрашивается так, чтобы никакие соседние узлы не получили одинаковый цвет, при этом число цветов соответствует числу реально имеющихся регистров. Если цветов не хватает, узлы итеративно удаляются, причем максимально долго остаются на регистрах переменные, используемые во внутренних циклах программы.

Поскольку при выполнении вычислений регистров может не хватать, содержимое некоторых из них приходится выгружать в память (независимо от выбранной стратегии их распределения). Выгруженное значение может понадобиться в последующих вычислениях, и его придется снова читать из памяти. Это значит, что при вычислениях встает проблема выбора того регистра, содержимое которого можно выгрузить с минимальными потерями в производительности.

Компилятор должен анализировать полученную им программу и выяснять, какое из значений ему понадобится для дальнейших вычислений и когда оно понадобится. Обычно алгоритм выбора регистра для выгрузки работает так, что им выбирается регистр, содержимое которого понадобится позднее других (это не всегда оптимально, но несложно определяется).

Аналогичным образом производится оптимизация специальных регистров – сумматоров, индексных регистров, регистров базирования и других. Проблема распределения регистров усложняется тем, что некоторые вычислительные и/или операционные системы накладывают дополнительные ограничения на использование регистров. Например, для некоторых операций иногда требуется сразу пара регистров, причем имеется требование к четности номера первого регистра из этой пары.

При генерации команд на основе внутреннего представления отдельных операторов программы довольно часто возникают ситуации, когда в общем потоке возникают лишние команды. Например, после записи содержимого некоторого регистра в память может сразу следовать команда загрузки этого регистра из того же элемента памяти (проблема "*считывания после записи*"). Вторая команда оказывается излишней, а компилятор старается ***удалить*** из созданной последовательности команды все ***излишние команды.***

При проведении оптимизации каждой команде присваивается некоторая характеристика, называемая ее "*стоимостью*" (с точки зрения времени выполнения команды, использования аппаратных ресурсов и т. п.). Компилятор стремится ***снизить совокупную стоимость программы***, то есть заменить "дорогие" команды более "дешевыми", но дающими тот же результат. Подобные преобразования могут, например, приводить к замене операции возвышения в степень операциями умножения, а в определенных случаях – даже операциями сдвига влево. Качество оптимизации оказывается особенно высоким, если удается заменять не отдельные "дорогие" команды, а связки команд.

В каждой вычислительной машине могут оказаться аппаратно реализованные команды, удобные для выполнения специфических операций (***машинные идиомы***). Можно встретить системы команд с аппаратными возможностями по вычислению тригонометрических функций, с одновременным вычислением частного и остатка в целочисленном делении, с выполнением некоторых команд в режиме автоувеличения или автоуменьшения, при которых к операнду прибавляется (или вычитается из него) единица до или после использования его значения в операции. Например, режимы автоувеличения и автоуменьшения очень удобны для организации работы со стеком, а также при выполнении операций вида $i:=i+1$.

Все большее развитие получают архитектуры, ориентированные на ***параллельные или векторные вычисления***. Очень часто в таких архитектурах удается *совместить одновременное выполнение* нескольких *операций*. Перед компиляторами это ставит задачу перераспределения последовательности вычислений так, чтобы рядом оказывались операции, не зависящие друг от друга (в противном случае, их нельзя будет выполнять параллельно). Для всей программы в целом решить такую задачу невозможно, но некоторые участки программы могут быть хорошо оптимизированы. Например, в архитектуре с одним потоком вычислений операцию $A+B+C+D+E+F$ надо выполнять в порядке $(((((A+B)+C)+D)+E)+F$. Если же вычислительная машина имеет два потока вычислений, лучше организовать вычисления так: $((A+B)+C)+((D+E)+F)$. Тогда операции $A+B$ и $D+E$, а также сложение с использованием их результатов могут выполняться параллельно.

При рассмотрении полезности оптимизирующих преобразований нужно учитывать, что наибольший выигрыш возникает при участии самого программиста, причем возникает он еще на этапе выбора алгоритма, который реализуется в программе. Верный выбор алгоритма всегда способствует получению эффективной программы. Например, замена алгоритма сортировки вставками алгоритмом быстрой сортировки может привести к изменению времени сортировки N элементов с $2.02N^2$ на $12\log_2 N$. Для реальной программы сортировки 100 чисел это изменение уменьшает совокупное время работы в 2.5 раза, для 100000 чисел снижение времени работы реальной программы – в 1000 раз!

3.3.5. Основные методы динамического распределения памяти

Во время распределения памяти компилятор ставит в соответствие лексическим единицам исходной программы адрес, определяет их размер и приписывает им атрибуты области памяти, необходимой для этой лексической единицы. О лексических единицах речь идет потому, что выбор области памяти и распределение памяти в ней проводятся не только для объектов данных, но и для выполняемых фрагментов программ – операторов, блоков, функций и процедур. Область памяти – это совокупность объединенных между собой элементов памяти, причем логика объединения задается семантикой входного языка.

Семантика всех программ подразумевает, что при выполнении программ области памяти будут необходимы для хранения:

- кодов пользовательских программ;
- данных, необходимых для работы этих программ;
- кодов системных программ, обеспечивающих поддержку пользовательских программ в период их выполнения;
- записей о текущем состоянии процесса выполнения программ (например, записей об активации процедур).

Области памяти, в которых проводится распределение, обладают двумя характеристиками – характеристикой использования этой памяти в программе и характеристикой способа ее распределения. По способу использования области памяти делятся на *глобальные* и *локальные*, а по способу распределения – на *статические* и *динамические*.

Наиболее проста ***стратегия статического распределения памяти***, в соответствии с которой память автоматически распределяется в статических областях компилятором. Размеры объектов, размещаемых в этих областях, никогда не меняются,

как и та часть адресов этих объектов, которая указывает их положение внутри области. Единственное, что может измениться – это начальный адрес самой области, но и это изменение происходит до выполнения программы, перед загрузкой ее в память.

Стратегия динамического распределения выбирается в тех случаях, когда на стадии компиляции не удается определить положение объекта в некоторой области памяти и/или его размер. При динамическом распределении возможно применение различных дисциплин, наиболее известны из которых ***стековая дисциплина распределения*** и ***дисциплина произвольного распределения*** (распределение в “куче”).

В случае выбора стековой дисциплины необходимо выделить некоторый фрагмент свободной памяти, на котором при запросах ресурсов памяти будет моделироваться работа со стеком областей памяти в стиле “первым освобождается последний из ранее размещенных фрагментов памяти”. Дисциплина произвольного распределения памяти, по-существу, означает отсутствие какой-либо дисциплины в этом распределении. Захват фрагментов памяти может осуществляться по произвольным запросам, так же производится и освобождение памяти. Распределение в соответствии с дисциплиной произвольного распределения память может происходить явно самим разработчиком программы, а может осуществляться компилятором автоматически, то есть неявно.

Выбор той или иной стратегии и дисциплины основывается на следующих критериях:

- эффективность начального распределения памяти;
- эффективность восстановления статуса “свободной памяти”;
- эффективность уплотнения свободных участков областей памяти (эффективность объединения свободных фрагментов во фрагменты суммарного размера).

Исходными данными для распределения памяти являются информационные таблицы компилятора, полученные на фазах анализа исходной программы. Эта информация собирается на основе обработки операторов описания объектов данных программы и пополняется при компиляции исполняемых операторов программы на основе семантических правил входного языка. Именно поэтому семантический анализ также должен предшествовать фазе распределения памяти.

С учетом выделенных объектов для хранения в памяти наиболее целесообразно выбирать

- для кодов пользовательских программ, кодов системных программ, буферов ввода/вывода:
  - статические области памяти и стратегию статического распределения.
- для данных разной природы, необходимых для работы различных программ:
  - статические области памяти и стратегию статического распределения – для глобальных, статических (не глобальных, а собственных) переменных, констант, внутренних структур данных, возникающих при трансляции некоторых языковых конструкций, например, таблиц виртуальных функций для полиморфных классов;
  - динамическую стратегию со стековой дисциплиной – для распределения памяти под локальные переменные;

- динамическую стратегию с дисциплиной произвольного распределения – для переменных, создаваемых по явному запросу с помощью специальных операторов или библиотечных функций (операторы new в Паскале и Си++, функция malloc() в Си), а также для переменных, размеры которых меняются в процессе выполнения программы (объекты типа vector из библиотеки STL).

- для записей о текущем состоянии процесса выполнения программ, а также для записей о входах в блоки операторов, которые можно рассматривать в качестве процедур без параметров.

  Например, для записей об активации процедур, которые содержат всю необходимую информацию для обеспечения выполнения процедур и возврата в точки вызова, в частности, фактические параметры, локальные, в том числе временные переменные, адреса возврата, значения регистров в момент входа в процедуру, ссылки на подобные записи объемлющих и вызвавших процедур. После завершения работы процедуры ее текущая запись об активации разрушается. Для записей активации можно выбирать
  - статическую стратегию с выделением фиксированных зон в памяти для каждой нерекурсивной процедуры и каждого блока нерекурсивных процедур.
  - динамическую стратегию со стековой дисциплиной (для языков программирования, поддерживающих рекурсивные процедуры).

Распределение памяти нужно проводить не для всех элементов программы. Многое определяется в этом процессе реализацией компилятора и особенностями вычислительной архитектуры. Например, если в системе команд машины имеется формат непосредственных данных (данных, размещаемых непосредственно в самой команде), то распределять память для некоторых целочисленных (а иногда и вещественных) констант в статической памяти не обязательно, они будут размещены в самой программе. Некоторые объекты данных программы могут при распределении памяти получить одинаковые адреса (например, две одинаковые строковые константы или две разные локальные переменные, никогда не используемые одновременно).

При распределении памяти для элементов данных может вызывать определенные трудности учет требований выравнивания адресов некоторых объектов данных на границы аппаратно поддерживаемых элементов данных – слов, двойных слов, параграфов и страниц. Требования выравнивания адресов могут приводить к потерям памяти, а также к трудностям при совместной работе программ, написанных на разных языках программирования и обрабатываемых разными компиляторами, имеющими разные стратегии распределения памяти. Потери памяти, возникающие из-за выравнивания, можно легко оценить, сравнив значения размеров сложных объектов (структур, массивов) с суммой размеров составляющих их элементов. Все эти проблемы обычно решаются в рамках каждой конкретной вычислительной системы по-разному (возможно даже хранение данных в упакованном виде с распаковкой их значений при попытке доступа к ним).

Глобальная область памяти выделяется один раз при инициализации объектной программы и доступна в этой программе все время, пока эта программа выполняется. ***Локальная область памяти*** выделяется в начале выполнения некоторого фрагмента работающей программы (оператора, блока, функции, процедуры). Использовать локальную память вне пределов ее видимости,

определяемых синтаксическими и семантическими правилами языков программирования, нельзя.

Распределение статической памяти (как глобальной, так и локальной) не вызывает особых сложностей, размеры объектов базовых типов данных точно определены для каждой вычислительной системы, компилятору надо лишь приспособить аппаратные элементы для хранения значений программных объектов, то есть выбрать оптимальный из многих возможных вариантов.

Наибольшие сложности вызывает ***распределение динамической памяти***, которая называется так потому, что ее размер не известен на стадии компиляции (независимо от того, глобальная это память, или локальная). Неопределенность размера динамической памяти не позволяет компилятору непосредственно зафиксировать ее адрес. Этого нельзя сделать даже символически, передав указание следующим компонентам системы программирования: редактор связей также не в состоянии будет определить этот адрес. Все, что может сделать компилятор, вставить в программу специальную подпрограмму, к которой можно будет обращаться для выделения места в динамической области памяти и освобождения этого места, когда потребность в нем исчезнет.

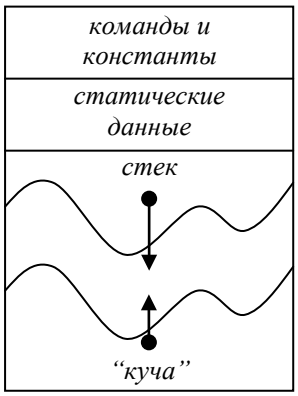
Обычно с динамическими областями памяти связаны многие операции с указателями и конструкторами экземпляров объектов (классов). При использовании динамических объектов памяти говорят о динамическом связывании объекта данных и области памяти.

В динамической области памяти могут располагаться как зоны, выделяемые пользователем (работающей программой), так и зоны, выделяемые компилятором, но и те и другие выделяются в процессе выполнения программы.

Пользователь может выделять и освобождать динамические области памяти, выполняя специальные операторы или обращаясь к библиотечным функциям (`new` и `dispose` в Паскале, `malloc()` и `free()` в Си, `new` и `delete` в Си++). Эти операторы и функции, в свою очередь, могут использовать возможности операционной системы, а могут производить распределение памяти самостоятельно в рамках статически выделенного большого участка памяти. В любом случае за распределение и освобождение памяти несет ответственность сам автор программы, а компилятор лишь вставляет в нужные места программы обращения к нужным функциям.

По-другому организована работа компилятора для работы с динамической памятью, ответственность за распределение которой возложена на компилятор. Потребность автоматически распределять динамическую память возникает, когда в программе используются такие операции над данными, которые требуют перераспределения памяти, а сами операторы перераспределения в программе отсутствуют. Примером такой ситуации может служить внешне простая операция конкатенации строковых переменных в языке Basic. В объектно-ориентированных языках программирования к такому перераспределению могут приводить некоторые операции над экземплярами объектов. Во всех таких случаях компилятор должен сформировать команды, обеспечивающие своевременное выделение памяти, необходимой для выполнения операции, а также последующее освобождение этой памяти.

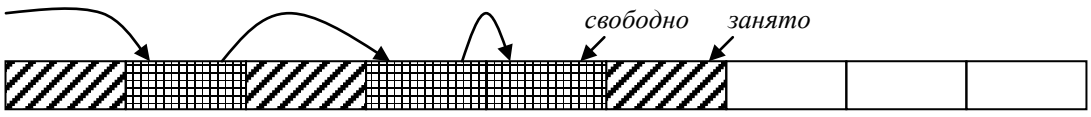
Общая структура размещения в памяти отдельных областей такова:



Из нее понятно, что для двух динамических областей, границы которых нельзя определить заранее, можно выделить общую зону и организовать в ней встречные направления роста стековой области и области с произвольной дисциплиной распределения. Точная граница между двумя динамическими зонами с разными стратегиями распределения памяти не предусматривается, вместо этого системная поддержка гарантирует проведение постоянного контроля за совокупным размером двух динамических зон.

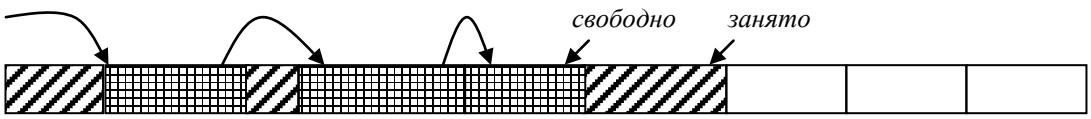
Технологии динамического распределения памяти. Техническая реализация методов управления динамической памяти в значительной степени зависит от тех потребностей, которые демонстрируют пользователи, определяя в своих программах объекты, память для которых должна выделяться на основе динамической стратегии.

В ответ на *явные* запросы пользователей, оформленные в виде явных обращений к системным функциям захвата памяти, обычно выделяются блоки одинакового размера:



По мере освобождения таких блоков начинает формироваться *список свободных блоков*, дальнейшие запросы пользователей приводят к выделению блоков из списка свободных, что позволяет снизить требования к наличной памяти вычислительной системы. Одинаковые размеры выделяемых блоков памяти делают проблему фрагментации памяти практически не актуальной. Единственной трудностью при данном подходе является выбор оптимального размера элементарного блока.

Второй вариант обработки явных запросов пользователей основан на более гибком подходе, при котором размер блоков не фиксируется заранее, а выбирается на основе параметров запроса и хранится в самом блоке:

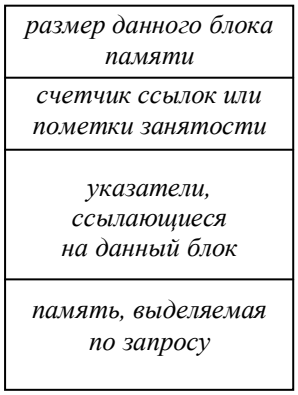


Это позволяет оставить почти без изменения работу системы при захвате памяти, но значительно усложняет ее при освобождении, а особенно при повторном выделении, которое может потребовать склеивания смежных блоков, а иногда и уплотнения занятых участков. Сложность процесса уплотнения связана с

необходимостью модифицировать все указатели на объекты, размещенные в перемещаемых блоках.

Неявное освобождение памяти проводится в случае автоматического управления распределением памяти, которое выполняет компилятор и программы системной поддержки в отсутствии явных операторов управления памятью в программе пользователя. Существенную роль в процессах освобождения и повторных захватах памяти начинает играть процедура “сборки мусора”, то есть поиска ранее освобожденных участков памяти, которые можно использовать повторно. Если в схемах, выбираемых для обработки явных запросов, практически не возникает никаких накладных расходов, то при неявном управлении памятью без больших накладных расходов памяти обойтись не удается.

Выделяемый блок памяти уже не может состоять из фрагмента, нужного пользователю и (может быть) дополнительного небольшого расхода на хранение размера блока. При управлении неявных запросов памяти в состав выделяемого блока приходится дополнительно включать специальный счетчик числа указателей, связанных с данным блоком, либо специальный признак занятости блока (“пометку”):

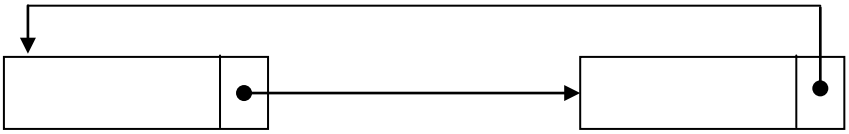


В блок обычно включается также полный список всех указателей, связанных с объектами данного блока. Работа с блоками, выделенными и освобождаемыми по неявным запросам, проводится по-разному, в зависимости от выбираемого алгоритма определения занятости блока.

Определение занятости блока с помощью счетчика ссылок основано на постоянном контроле числа указателей, имеющихся в программе и содержащих ссылки на объекты, размещенные в данном блоке памяти. Такой подход связан с контролем всех операторов присваивания вида

```
p = q;
```

где p и q – указатели на некоторые динамически созданные объекты. Перед выполнением присваивания значение счетчика ссылок блока, на который ссылается указатель p, уменьшается единицу, одновременно увеличивается на 1 значение счетчика ссылок блока, на который ссылается указатель q. Только после этого можно выполнять само присваивание. Блок считается свободным, если значение его счетчика ссылок стало равным нулю. Проблемой при таком подходе является обработка недостижимых циклических ссылок:



При работе программ возможно возникновение сложных списочных структур, создающих закольцованные последовательности ссылок. Такие ссылки учитываются при подсчете значений счетчиков ссылок блоков памяти, но сами структуры могут реально оказаться недостижимыми в программе, так как ни один из указателей, не входящих в созданное кольцо (внешних по отношению к этому кольцу), не ссылается ни на один из элементов списочного кольца. Такие недостижимые циклические ссылки становятся препятствием для освобождения ставшей ненужной памяти и создают реальные проблемы для систем управления памятью.

Определение занятости блоков памяти с помощью пометок позволяет избежать обозначенной проблемы циклических ссылок и необходимости отслеживания всех присваиваний указателям за счет организации сложного и затратного механизма “сборки мусора”. Этот механизм включается в момент, когда перед системой управления памяти возникает необходимость выделить фрагмент памяти такого размера, который невозможно обеспечить, не проведя уплотнения занятых блоков памяти. Алгоритм работает следующим образом:

- все ранее выделявшиеся блоки памяти помечаются как свободные;
- анализируются все указатели – переменные программы и помечаются занятыми все блоки, на которые они ссылаются;
- итеративно анализируются все указатели, хранящиеся в виде значений полей объектов, размещенных в блоках, помеченных, как занятые; процесс останавливается, когда новые занятые блоки перестают возникать;
- все недостижимые по указателям блоки памяти остаются свободными, занятая ими память может быть уплотнена; одновременно с уплотнением занятых блоков осуществляется модификация значений действующих указателей.

Процесс “сборки мусора” запускается системой управления памятью в произвольные моменты времени, что может влиять на процесс решения сразу многих задач, работающих в вычислительной системе в одно и то же время. Эффективная реализация этого процесса является одной из важнейших характеристик как систем программирования, так и программ системной поддержки.

3.3.6. Генерация кода

Завершающей стадией работы компилятора является генерация кода (команд, констант и т. д.) объектной программы. Генератор кода получает на вход промежуточное представление исходной программы и вырабатывает эквивалентную объектную программу, используя для этой цели всю информацию, накопленную к этому моменту в информационных таблицах компилятора.

Работа генератора не зависит от того, имеется ли перед фазой генерации дополнительная фаза оптимизации, или она отсутствует. Математически проблема генерации эффективной объектной программы является неразрешимой, но на практике

применяются достаточно эффективные эвристические подходы, дающие хорошие результаты, хотя и не абсолютно оптимальные.

На вход генератора кода поступает внутреннее, промежуточное представление компилируемой программы, полученное на начальных стадиях компиляции, осуществляющих анализ исходной программы и построение информационных таблиц компилятора. Обычно таблица информации используется генератором кода для определения тех адресов объектов программы, которые они будут иметь в процессе выполнения программы. Так как перед началом генерации кода уже выполнен синтаксический разбор, отсеяны очевидно ошибочные конструкции и получено полное внутреннее представление исходной программы, во внутреннем представлении программы эти объекты обозначаются специально построенными внутренними именами, по которым записи об объектах легко отыскать в информационных таблицах, а не теми именами, которые присвоил им программист.

Внутреннее представление, поступающее на вход генератора кода, может иметь различные формы. Одно из самых существенных предположений, которые делаются при разработке генераторов кода, состоит в том, в поступающем на его вход внутреннем представлении программы отсутствуют ошибки (хотя иногда генерация кода выполняется параллельно с семантическим анализом программ).

Реальные компиляторы готовят результаты своего труда в виде объектных программ, для кодирования которых могут использоваться такие виды языков:

- машинные коды в абсолютных адресах;
- перемещаемый машинный код;
- язык ассемблера объектной машины;
- другой язык программирования высокого уровня.

Абсолютный машинный язык имеет свои преимущества в том, что программы на таком языке могут помещаться в фиксированные места памяти машины и немедленно выполняться. Такие системы обычно используются в целях обучения языкам и методам программирования. Некоторые языки программирования (Алгол-60 и Паскаль) специально разрабатывались в расчете на использование подобной схемы компиляции. Для других языков больше подходят другие режимы работы компиляторов.

Генерация перемещаемого объектного модуля с внутренним кодированием на машинном языке обеспечивает возможность раздельной компиляции файлов, из которых строится текст программы. Множество перемещаемых объектных модулей могут затем связываться в единое целое и помещаться в память машины для совместного выполнения. Такое связывание выполняется редактором связей, а размещение в памяти – загрузчиком. Такой режим приводит к дополнительным затратам на связывание модулей, но раздельная компиляция имеет так много преимуществ (например, с ее помощью удается строить библиотеки программ), что компенсирует все эти затраты. Чтобы редактор связей мог провести свою часть работы по созданию полноценной самодостаточной программы, компилятор передает ему дополнительную информацию в объектных модулях.

Значительно облегчить процесс генерации объектной программы может использование в качестве выходного кода языка ассемблера объектной машины. Генерация текстов на языке ассемблера имеет особенный смысл при наличии готового ассемблера, который можно использовать для завершения работы по компиляции,

также полезно в составе многоязыковых систем программирования для связи компонентов, написанных на разных языках между собой.

Генерация программы на другом языке программирования часто используется при наличии компилятора с некоторого языка программирования. При такой поддержке другие языки можно транслировать в один, обеспечивая полный цикл обработки программ. Эта же технология используется для быстрого создания прототипа компилятора или при проведении процедуры "*раскрутки*", когда сначала создается компилятор для некоторого языкового ядра (ограниченного варианта языка), а затем последовательными переходами от версии к версии создается серия компиляторов для все более и более полных вариантов языка. Например, язык Фортран своим стандартом определен через ядро и дополнительные слои, которые можно отключать при компиляции, борясь за эффективность программ, но (может быть) в ущерб удобству программирования.

Любой генератор кода, независимо от формы внутреннего представления программ в компиляторе и вида выходного представления результатов компиляции должен в своей работе решить две главные задачи:

- выполнить распределение памяти для объектов данных и фрагментов компилируемой программы,
- выполнить поиск и подбор семантических эквивалентов конструкциям внутреннего представления программ.

Важной дополнительной функцией генератора кода в таком процессе, заметно усложняющей весь процесс генерации в целом, является проведение одновременно с генерацией поиска стандартных последовательностей операторов по некоторым шаблонам и локальной оптимизации потока формируемых команд.

Поиск по шаблонам помогает обнаруживать в программах некоторые часто используемые ресурсоемкие последовательности операторов, для которых в конкретной аппаратуре объектной машины могут иметься решения, существенно повышающие эффективность их выполнения. Например, для машин с конвейерной или векторной архитектурой очень важно распознавать в программах, так называемые ливерморские циклы. Ливерморские циклы представляют собой 24 программы (первоначально написанные на Фортране) из состава производственных программ, разработанных Ливерморской национальной лабораторией имени Лоуренса (США). Циклы являются вычислительными ядрами, характерными для трудоемких научных расчетов. Они включают в себя как общие математические операции (скалярное произведение и умножение матриц), так и сложные алгоритмы поиска и хранения (поисковый цикл Монте-Карло), например так на языке Си выглядит четвертый ливерморский цикл, построенный на базе фрагмента подпрограммы решения ленточных линейных уравнений:

```
void Loop4 (int n, REAL X [AR_1001], REAL Y [AR_1001])
{ for (int k = 6; k < AR_1001; k += (AR_1001 - 7) / 2)
     { int lw = k - 6;
       for (int j = 4; j < n; j += 5) X [k - 1] -= X [lw ++] * Y [j];
       X [k - 1] *= Y [4];
     }
}
```

Обнаружение подобной критической последовательности операторов позволяет вместо прямой подстановки обычной последовательности команд сформировать последовательность, специфическую для данного шаблона и конкретной вычислительной архитектуры. Такая замена может уменьшить время выполнения важной программы в десятки или сотни раз.

3.4. Редакторы связей: назначение, принципы работы

Конечным результатом компиляции является объектный модуль. За один запуск компилятора всегда порождается ровно один объектный модуль. Если какой-либо компилятор создает при компиляции с языка программирования текст на языке ассемблера, объектный модуль все равно создается, но работа по его созданию перекладывается на ассемблер. Дальнейшая работа над объектными модулями проводится *редактором связей*, которые иногда называются *компоновщиками*.

Основное назначение редактора связей – завершить ту часть работы, которая принципиально не могла быть выполнена компилятором, а именно, осуществить пр ивязку нескольких модулей друг к другу. Компилятор не мог выполнить такую привязку потому, что он всегда работает только с одной компилируемой программой, зная о других точнее программных компонентах только то, что они *должны существовать*, а также их *программные интерфейсы*.

В отличие от компилятора, который во время своего запуска обрабатывает только один объект (программный компонент), редактор связей в общем случае получает на вход сразу несколько объектных модулей. Эти модули могут быть получены редактором связей непосредственно от компилятора, а могут извлекаться им из библиотек, которые также указываются среди параметров запуска компоновщика.

Редактор связей должен заново прочитать все объектные модули (как откомпилированные, так и библиотечные), необходимые для формирования одной полной программы, и выявить в них все упоминания *внешних* объектов (процедур, функций, констант, переменных и т. д.). Внешними эти объекты являются по отношению к тому конкретному программному компоненту, в котором они упоминаются и используются, но не определяются или не реализуются. Задача редактора связей – отыскать среди всего набора объектных модулей те, которые определяют или реализуют внешние объекты других модулей. В конечном итоге должны быть обнаружены все определения и реализации всех внешних объектов. Если же некоторые внешние связи остались *неразрешенными*, то есть соответствующие им объекты не обнаружились ни в одном из объектных модулей, поданных на редактирование, редактор связей выдает сообщение об ошибке.

Редактор связей в состоянии провести и другой контроль – контроль соответствия между объектным модулем, в котором упоминается некоторое внешнее имя (используется объект с этим именем), и объектным модулем, в котором данный объект определен. В тех случаях, когда имена объектов совпадают, а семантика их использования различается, могут возникать трудно обнаруживаемые ошибки. Найти их удается только при отладке уже готовых программ. Избежать подобных ошибок при работе с библиотеками удается только, если точно следовать правилам работы с внешними компонентами, которые подробно разъясняются их поставщиками.

Задача редактора связей – сформировать области (разделы, секции) памяти, которые впоследствии смогут быть размещены в памяти вычислительной машины как единое целое, в виде цельных блоков. Области могут иметь инициирующие значения, а могут быть пустыми, то есть использоваться только для резервирования памяти.

Значениями, которыми инициируются разделы памяти, могут быть последовательности команд (программные разделы) или начальные значения статических объектов данных, в том числе констант. Для каждой области редактор связей вводит свой начальный адрес (реальное его значение определяется на более поздних стадиях, вплоть до записи программы в физическую память для выполнения). Истинные адреса начала областей обычно используются только в командах загрузки адресов на регистры базирования, поэтому только эти команды обычно остаются не до конца доработанными после завершения редактирования связей. Каждый объект программы, определенный в некотором объектном модуле относится редактором связей к одной из областей, для всех этих объектов редактор связей вычисляет их относительные адреса. После его работы останется только осуществить прибавление начального адреса загрузки области памяти. В некоторых вычислительных системах на редактор связей возлагается также обязанность проводить выравнивание адресов при компоновке областей памяти.

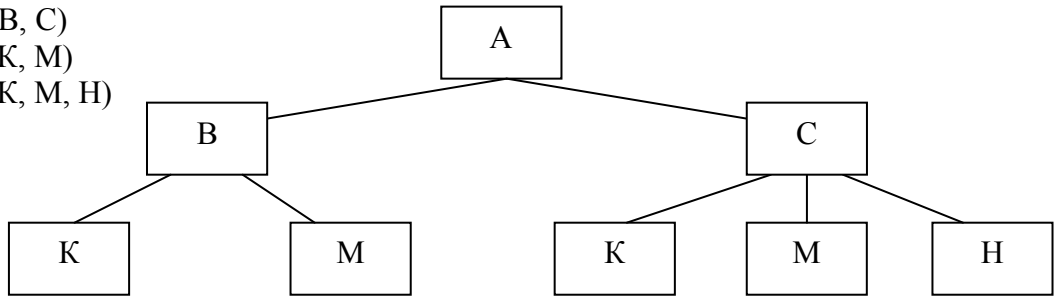
От редактора связей не зависит эффективность выполнения готовой программы. Все, что можно сделать для повышения этой эффективности, делается в период компиляции. Однако от редактора связей может зависеть эффективность использования памяти вычислительной машины, поскольку именно при редактировании связей определяется истинный размер готовой программы.

Самый простой редактор связей при обнаружении ссылки на некоторый объектный модуль (или даже при передаче некоторого объектного модуля на редактирование, независимо от того, имеются на его объекты ссылки в других объектных модулях или нет) просто вставлять в готовую программу все определенные в нем объекты, как программы, так и данные. В таком случае файл готовой программы (и размер, занимаемой ею памяти машины) будет максимальным, но в этом файле могут оказаться объекты, никогда в программе не используемые. В особенности часто подобная ситуация может возникать при работе с библиотеками, в которых компонуются множество семантически связанных процедур, из которых в реальной работе используется лишь некоторая часть. Чтобы избежать потерь памяти разработчики библиотек часто оформляют их не в виде одного большого библиотечного файла, а виде набора относительно небольших файлов, вставка которых в готовые программы не будет приводить к их существенному росту.

Однако лучшим выходом из этой ситуации является усложнение алгоритма работы редактора связей, который обладает всей информацией, необходимой для отбора тех объектов модуля, которые реально используются в программе. Неиспользуемые объекты могут исключаться из рассмотрения и в формируемые разделы памяти не попадать. В таком случае они не будут попадать и в файл готовой программы. Современные системы программирования стараются комплектовать именно такими редакторами связей, использование которых снижает нагрузку на используемую память машины.

Еще одним важным свойством редакторов связей является способность некоторых из них формировать *оверлейные* структуры, то есть структуры программ, имеющих одинаковые адреса в памяти, но никогда не размещаемые в памяти одновременно, а замещающие друг друга в процессе работы программ. Для того, чтобы редакторы связей могли формировать оверлейные структуры, им в качестве параметра запуска передают описание процесса смены модулей в памяти, выполняемое на некотором простейшем языке спецификаций:

Модуль A (B, C)
Модуль B (K, M)
Модуль C (K, M, H)



В приведенном примере модули *B* и *C* будут иметь в памяти машины одинаковые адреса. Также одинаковые адреса будут у группы модулей *K* и *M* и группы модулей *K, M* и *H*. Поскольку размеры модулей *B* и *C* могут отличаться друг от друга, начальные адреса разных групп модулей, располагающихся на одном уровне оверлейной структуры, в общем случае отличаются друг от друга, даже если в них входят одни и те же модули.

Оверлейные программы создаются для того, чтобы избежать ситуаций, в которых линейно загруженные модули, составляющие целую программу, не могут быть размещены в памяти машины из-за наличия физических ограничений. Оверлейные структуры позволяют загружать программы в память не целиком, а фрагментарно, имея в каждый момент выполнения программы в памяти те ее фрагменты, которые в этот момент необходимы. Если же обращения к оверлейному фрагменту не происходит, то и в память он не загружается.

3.5. Загрузчики: основные функции, принципы работы

Работа редактора связей заканчивается формированием собранной программы, для которой остались неизвестными лишь начальные адреса размещения разделов памяти. Ни компилятор, ни редактор связей не в состоянии знать, в какой именно области физической памяти будет размещаться программа в момент ее выполнения. Эти компоненты работают лишь с *относительными* адресами, которые отсчитываются от некоторой условной точки (обычно она совпадает с началом разделов памяти, отводимых для объектов самого первого модуля, поданного для компоновки редактору связей). Задача следующего преобразования – преобразовать условные адреса разделов памяти в истинные (*абсолютные*). Такое преобразование выполняется загрузчиками.

Загрузчики могут включаться в состав систем программирования, но чаще они оказываются составными частями операционных систем, поскольку выполняемые ими функции не только зависят от архитектуры вычислительной системы, в которой должна выполняться программа, но также и от конкретной физической конфигурации этой системы (в частности, от точного количества и размеров модулей памяти). Загрузчики обязательно входят в состав тех систем программирования, которые передают подготовленные компиляторами и редакторами связей программы не аппаратуре на исполнение, а программе интерпретатора для интерпретации.

Чтобы загрузчик (в какую бы систему он ни входил) мог выполнить свои функции, редактор связей вставляет в передаваемую ему программу специальную таблицу, с помощью которой можно определить все места в программе, где надо произвести модификацию условных или относительных адресов в абсолютные. Обрабатывая информацию, содержащуюся в этой таблице, загрузчик создает окончательное представление тех команд программы, которые еще не были к этому времени сформированы.

Формат таблицы трансляции адресов зависит не только от архитектуры вычислительной системы, но и от той операционной системы, которая должна управлять выполнением готовых программ. Это делает несовместимыми друг с другом программы, подготовленные в рамках разных операционных систем (например, программы, подготовленные для систем Windows и Linux, имеют разные по структуре таблицы трансляции адресов, хотя и те, и другие должны выполняться на одной и той же аппаратуре – процессоре персонального компьютера). В то же время вынесение загрузчика из состава систем программирования в состав операционных систем делает структуру этой таблицы более общей, не зависящей от конкретной системы программирования. В противном случае, программы, подготовленные системами программирования компании Borland, нельзя было бы выполнять в операционной системе, предназначенной для работы с системами программирования компании Microsoft.

Вынесение загрузчиков в операционные системы имеет еще один очень важный смысл. Современные вычислительные комплексы могут иметь сложные аппаратно управляемые структуры памяти. Методы трансляции адресов могут основываться на сегментной, страничной или сегментно-страничной организации памяти. Полное владение ситуацией может обеспечить только операционная система, причем только в момент непосредственного занесения программы в память. Это означает, что загрузчик системы программирования в принципе не способен решить все проблемы модификации адресов, поскольку он не может знать точных характеристик конфигурации аппаратных средств и состояния внутренних таблиц подсистемы управления памятью операционной системы в момент, когда программа начнет выполняться. Загрузчик, который выполняет трансляцию адресов в момент запуска программы, называется *настраивающим загрузчиком*.

3.6. Техника работы с библиотеками

Существенной особенностью систем программирования является наличие и номенклатура библиотек подпрограмм. По техническому составу ***библиотеки*** делятся на две категории: библиотеки функций исходного языка и библиотеки функций операционной системы, в составе которой должна будет работать обрабатываемая программа. Эта операционная система может отличаться от той, в составе которой функционирует сама система программирования.

Некоторую часть библиотеки, а именно описания входящих в библиотеки компонентов, необходимо передавать компилятору, чтобы во время компиляции программ, ссылающихся на библиотечные элементы, компилятор мог проверять, например, соответствие списков формальных и фактических параметров. Фактические параметры задаются в компилируемых программах, а формальные параметры описываются в заголовках библиотечных процедур. С другой стороны, библиотеки сами формируются компиляторами, поскольку реализации библиотечных элементов описываются на языках программирования.

Библиотеки делятся на *статические* и *динамические* по выбору того момента, когда система программирования извлекает из них элементы.

3.6.1. Статические библиотеки

Статические библиотеки фактически представляют собой процедуры и функции, встраиваемые внутрь программ, подготавливаемых системами программирования на этапе обработки этих программ. Все статические библиотеки подключаются к

программам, готовящимся к выполнению, ровно один раз в момент формирования редактором связей полной программы. После подключения компоненты статической библиотеки становятся неотъемлемой частью программы и в дальнейшем распространяются вместе с нею. Программа, к которой уже подключены компоненты статической библиотеки, приобретает независимость от самой этой библиотеки. Если в библиотеку будут внесены изменения, вносить какие-либо параллельные изменения в программу, использующую исправленную библиотеку не требуется. Связь программы и библиотеки после подключения этой библиотеки к программе разрывается. Это свойство статических библиотек является их несомненным преимуществом.

Другой стороной разрыва связи программы и библиотеки в момент их объединения является трудоемкость исправления программ в случае, если внесение изменений в библиотеку объясняется исправлением существовавшей в ней ошибки. Эта трудоемкость связана с необходимостью произвести повторную полную сборку программы из объектных модулей и исправленных библиотечных компонентов.

Статическое подключение компонентов библиотек к объектным модулям ведет к существенному росту объемов готовых программ. В современных условиях, когда библиотеки достигли высокой степени развития, доля компонентов библиотек в объеме готовых программ уже превышает половину этого объема. Поскольку некоторыми системными и стандартными библиотеками пользуются почти все программы, размеры непроизводительно используемой памяти, как в архивах программ, так и в памяти вычислительных машин при выполнении программ, могут быть очень большими.

3.6.2. Динамически загружаемые библиотеки

Компоненты динамических библиотек (динамически загружаемые компоненты и библиотеки) подключаются к программам *во время выполнения этих программ*. В этом состоит основное отличие динамических библиотек от статических. Компоненты динамических библиотек не связаны с программами, которые к ним обращаются, и распространяются отдельно от них.

Некоторые динамические библиотеки подключаются к программам в самом начале выполнения программы при ее записи в оперативную память вычислительной машины. Какой-либо связи с тем, будет в программе реальное обращение к компонентам библиотеки или нет, здесь не возникает. Другие библиотеки попадают в оперативную память только тогда, когда происходит реальное обращение к какому-нибудь одному из их компонентов (к какой-либо процедуре или функции из этих библиотек). Освобождение памяти для разных библиотек также происходит по-разному. Память может освобождаться либо после завершения выполнения всей программы, либо по указанию главной программы, которая просигнализирует, что некоторая библиотека больше использоваться ее не будет. Возможен и вариант, когда память, занимаемая динамическим подгруженным компонентом, будет освобождена непосредственно после завершения выполнения этого компонента.

Загрузка при обращении к компоненту библиотеки имеет определенные преимущества перед загрузкой при начале выполнения программы. Память вычислительной машины при таком методе загрузке библиотек используется более рационально, но этот метод сложнее для реализации. Он требует либо существенного усложнения операционной системы, либо внесения в программы специальных указаний для динамической загрузки, поскольку только разработчик программы может точно знать, когда потребуется загрузка в память той или иной библиотеки (или какого-либо

компонента библиотеки). Точно также только разработчик программы может точно знать, когда можно освободить память от используемой библиотеки.

Загрузка при начале выполнения программы менее экономно расходует оперативную память, но проще реализуется, поскольку может быть полностью автоматизирована, а задача такой динамической загрузки может быть решена компоновщиком программ (редактором связей). Для этого достаточно в заголовок программы добавить перечень всех используемых в программе библиотек. Имея такой перечень, который обычно передается компоновщику в качестве одного из параметров его запуска, он может обеспечить загрузку всех библиотек в память перед началом работы основной программы.

Современные системы программирования обычно допускают оба метода работы с динамическими библиотеками, а решение по выбору конкретного метода принимается разработчиком программы. В общем случае, для разных библиотек можно выбирать разные методы. Выбор зависит от размеров библиотек и количества обращений к их компонентам.

Загрузка динамических библиотек в оперативную память, освобождение этой памяти, а также связывание основной программы и ее данных с компонентами библиотек (программами и данными) выполняется *динамическим загрузчиком*. Этот загрузчик входит в состав операционной системы и контролирует все обращения к динамически загружаемым библиотекам и модулям, считает эти обращения, а также освобождает память, когда количество действующих обращений оказывается равным нулю. Передача функции загрузчика из системы программирования в операционную систему позволила обеспечить повторное использование библиотечных компонентов при обращении к ним нескольких различных программ. Если динамическая библиотека уже загружена в память, нет необходимости загружать ее повторно. Загрузчик способен отслеживать двойные обращения и обеспечивать доступ к библиотечным компонентам всем работающим программам.

Выполнение функций динамическим загрузчиком возможно только в том случае, если загружаемая библиотека или объектный модуль имеют определенную структуру, содержащую в себе информацию, необходимую для трансляции адресов. Большинство современных редакторов связей могут работать в режиме формирования таких библиотек и модулей. Непосредственная загрузка динамических объектов в память выполняется настраивающим загрузчиком, при этом их программная часть оказывается доступной всем исполняемым программам, а часть, содержащая данные дублируется для каждой исполняемой программы так, как будто она является единственной программой, использующей динамически загружаемые объекты.

Формат файлов динамических библиотек полностью зависит от операционной системы, он близок к формату исполняемых файлов. В эти файлы (так же, как и в статических библиотеках) включены описания компонентов, что позволяет компиляторам автоматически контролировать операторы обращения к библиотечным компонентам. В то же время каких-либо команд обращения к этим компонентам компиляторы для динамических библиотек вставить не могут. Вместо них в программы вставляются команды обращения к функциям операционной системы, которые обеспечивают обращения к компонентам динамических библиотек. Эти команды могут вставляться автоматически, что предполагает проведение динамической загрузки в момент запуска основной программы, либо “вручную” самим разработчиком программы, что обеспечит реализацию динамической загрузки в момент реального

обращения к библиотечным компонентам. Компоненты, к которым завершены все обращения, могут удаляться операционной системой из памяти машины. Вновь они попадают в память только после очередного реального обращения к ним.

Динамические библиотеки имеют множество преимуществ перед статическими. Их составляющие не должны включаться в состав исполняемых файлов программ, что значительно сокращает их размеры. Этого преимущества удается добиться за счет разработки специального механизма операционной системы, который выполняет подключение фрагментов одних программ к другим в момент их выполнения.

Известным недостатком динамических библиотек является зависимость программного обеспечения от библиотечных объектов, непосредственно не связанных с этим обеспечением, то есть зависимость программ пользователей от неизвестных им динамических библиотек. Для выполнения программ, основанных на динамическом подключении библиотек, необходимо принимать специальные меры, гарантирующие наличие всех необходимых библиотечных компонентов в составе той вычислительной системы, где предполагается исполнять эти программы. Одновременно возникает зависимость самой логики работы программ от подключаемых к ней библиотек. Изменения в библиотеках осуществляются системными средствами, не зависящими от разработчиков прикладного обеспечения, а эти изменения могут влиять на работу программ, которые сами не менялись перед этим продолжительное время.

Это означает, что использование динамических библиотек налагает обязательства, как на их разработчиков, так и на их пользователей. Разработчик прикладной программы должен тщательно придерживаться правил, предложенных разработчиками библиотек. В свою очередь, разработчики библиотек, внося в библиотеки изменения, должны предпринимать меры, чтобы эти изменения минимальным образом сказывались на прикладных программах, основанных на предыдущих версиях библиотеки. Одним из результатов таких мер является то, что в современных библиотеках имеются специальные библиотечные компоненты, которые позволяют точно определить версию данного библиотечного набора компонентов.

Описанное повышение эффективности использования памяти вычислительной системы происходит за счет увеличения времени выполнения программ, поскольку на динамическую загрузку приходится тратить дополнительное время.

3.6.3. Основные типы библиотек

Никакая современная система программирования не может обойтись без встроенных в нее библиотечных средств. Для широкого распространения библиотечных средств в программировании имеются, по крайней мере, две причины:

- необходимость оказывать поддержку программам во время их исполнения на вычислительной машине,
- потребность накапливать полезные программы и передавать их другим пользователям, не раскрывая деталей реализации алгоритмов, запрограммированных в них.

По функциональному наполнению все используемые в составе современных систем программирования библиотеки можно классифицировать следующим образом:

- библиотеки функций, процедур и макроопределений,
- библиотеки классов,
- библиотеки компонентов.

3.6.3.1. Библиотеки функций, процедур и макроопределений

Многие программы, прошедшие компиляцию, нуждаются в поддержке во время выполнения. Почти все языки программирования включают в себя некоторые элементы, реализация которых подразумевает, что во время выполнения программы должна обеспечиваться связь с операционной системой. Наиболее наглядным примером может быть пример реализации операций, связанных с вводом и выводом информации. В каждом языке программирования имеются операторы ввода/вывода, которые не могут быть реализованы никаким образом, кроме обращений к системным программам, собранным в библиотеку, представляющую собой коллекцию объектных модулей, сформированных заранее при разработке самого компилятора. Естественно, что нет никакой необходимости в том. чтобы программы, входящие в такие библиотеки, были написаны на том же языке программирования, с которого ведется компиляция. Однако необходимо, чтобы компоненты библиотеки были написаны с учетом того, что к ним могут делаться обращения из программ, написанных на определенном языке и компилируемых конкретным компилятором.

Библиотеки связаны не только с целевой машиной и целевой вычислительной системой, но даже с конкретным компилятором, которым выполнялось их формирование. Однако некоторые языки программирования имеют настолько близкую семантику вызовов процедур и представления данных, что компиляторы для них могут создавать программы, которые можно вызывать из программ, написанных на других языках.

Необходимость оказания системной поддержки программам, проходящим обработку в системах программирования, повлияло на первоначальное наименование библиотек, которые сначала назывались *библиотеками системных программ* или *библиотеками стандартных программ*.

Параллельно с разработкой системных библиотек началась работа по разработке библиотек прикладных программ, которые со временем превратились в *пакеты прикладных программ*, то есть в совокупности программ, позволяющих выполнить весь комплекс операций по обработке информации. Потребность создания прикладных пакетов существует для каждой прикладной области. Например, в области математических расчетов созданы многочисленные пакеты программ, лидером среди которых является пакет, принадлежащий международной *Группе Численных Алгоритмов* (*The Numerical Algorithms Group*). Прикладные программы, входящие в этот пакет, предназначены для подключения к программам пользователей, написанным на языке Си и разных вариантах языка Фортран (Фортран 77, Фортран 90, Фортран 95), причем для разных операционных систем и трансляторов (Compaq Alpha Tru64, IBM RS/6000, Intel Linux *pgf77*, Intel Linux *g77*, Silicon Graphics IRIX). В дополнение к обычным расчетным программам в пакет входит набор из 76 статистических расширений для проведения моделирования и применения мультивариативных методов непараметрической статистики для электронных таблиц Microsoft Excel для операционных систем Windows 95/98/NT/2000/XP.

В Научно-исследовательском Вычислительном центре МГУ создана библиотека численного анализа для использования с трансляторами *pgf77* и *pgcc* с языков Фортран-77 и Си, разработанными Portland Group/STM.

Созданы пакеты прикладных программ для автоматизации бухгалтерского учета и обработки финансовой информации (многие из них используются для работы с

программами, написанными на языке Кобол). Широко известны пакеты прикладных программ для управления базами данных (СУБД) и издательских систем.

В библиотеки могут включаться не только объектные модули системных программ, но и макроопределения. Поддержка, которая оказывается ими, осуществляется не на стадии выполнения программ, а на стадии их компиляции, поскольку макроопределения и соответствующие им макровызовы вставляются непосредственно в тексты компилируемых программ.

Современные библиотеки содержат также интерфейсную информацию, предназначенную для чтения автоматизированными системами программирования. Такая информация может использоваться при передаче компилятору сведений о составе входящих в библиотеку процедур и функций. Обрабатывая эти файлы, компилятор автоматически получает всю необходимую информацию о компонентах библиотеки, причем эта информация поставляется в терминах входного языка компилятора. Тем самым автор компилируемой программы избавляется от необходимости вставлять в свои тексты описания библиотечных компонентов – функций, процедур, а также констант и переменных.

3.6.3.2. Библиотеки классов

Следующим шагом в развитии библиотек оказалось создание библиотек классов для систем программирования, основанных на объектно-ориентированных языках программирования (Си++, Java). Библиотеки классов могут представлять собой

- совокупности независимых классов,
- иерархии классов,
- иерархии шаблонов классов.

Простые наборы описаний классов встречаются в системах программирования все реже и реже. Такие наборы могут оказаться полезными, но пользоваться ими нелегко, так как они обычно плохо структурированы.

Иерархические библиотеки оказывают своим пользователям лучшую поддержку. Часто в основу иерархий закладывается один наиболее общий класс (*Object*), а остальные классы строятся как производные от него. Однако такой подход может привести к возникновению проблемы “жирного интерфейса”. На более верхних уровнях иерархии должна собираться более общая функциональность. Таким образом, либо исходный базовый класс оказывается пустым (то есть присущей абсолютно всем элементам иерархии общей функциональности выделить не удается), либо строится абстрактный класс с огромным набором виртуальных функций.

Пустой базовый класс просто имитирует связь нижних уровней иерархии между собой, фактически же такая связь в данном случае отсутствует. Наличие виртуальных функций (иногда семантически далеких друг от друга) заставляет при формировании нижних уровней иерархии реализовывать их все, даже в тех случаях, когда для некоторого конкретного производного класса конкретные функции не могут быть реализованы сколько-нибудь эффективно в виду невозможности придать им полезную интерпретацию. Отсутствие реализации хотя бы одной виртуальной функции превращает класс в абстрактный, что ограничивает его использование.

Решают подобные проблемы созданием систем иерархий, то есть построением наборов иерархических деревьев (*“леса”*), которые между собой не связываются

никакими отношениями (по такому принципу построены библиотека STL и стандартная библиотека языка Си++).

Библиотеки шаблонов построены на основе параметрического полиморфизма, то есть параметризации типов, использования типовых параметров. Библиотека STL и стандартная библиотека языка Си++ представляют собой наборы иерархий шаблонов классов.

3.6.3.3. Библиотеки компонентов

Библиотеки компонентов представляют собой развитие понятия библиотек языков программирования на основе развития концепции классов. Компонентами таких библиотек обычно являются законченные программные модули, из которых достаточно легко строить наиболее типичные приложения, относящиеся к самым произвольным прикладным областям. Библиотеки компонентов могут включать в себя генераторы отчетов, компоненты для построения сводных таблиц, компоненты для построения графиков и диаграмм, компоненты для создания графических интерфейсов. Библиотечные компоненты имеют общее программное ядро и проектируются на базе единых архитектурных принципов, что облегчает их совместное использование, сокращает время обучения для разработчика.

Компоненты, включаемые в библиотеки, подчиняются правилам инкапсуляции, то есть имеют открытые реализованные интерфейсы, а детали реализации скрываются внутри библиотек и не видны пользователям. Часто компоненты поставляются в виде двоичных модулей, что позволяет сделать их более независимыми от конкретных систем программирования и использовать в распределенном системном окружении, но противоречит стремлению сделать эти компоненты настраиваемыми (*гибкими*). Для удобства использования над компонентами позволяется проводить операцию контейнеризации, то есть помещения в контейнеры, допускающие внешнее визуальное представление. Такие контейнеры поддерживают развивающуюся технологию визуального программирования (в стиле “*drag & drop*”).

Примерами библиотек компонентов являются распространяемые компанией Microsoft библиотеки COM (*Component Object Module*) и DCOM (*Distributed COM*), библиотеки различных компаний, построенные на основе стандарта CORBA (*Common Object Request Broker Architecture*), библиотеки, входящие в состав серверов приложений J2EE и .NET.

Суммируя, можно говорить о том, что все современные библиотеки делятся на две категории:

- библиотеки, связанные с конкретными системами программирования, и
- библиотеки, связанные с конкретными задачами, решаемыми с помощью вычислительных машин.

Такое деление никак не связано с технической реализацией библиотечных средств. Как системные (включая стандартные), так и прикладные библиотеки могут быть и статическими и динамическими.

3.6.3.4. Критерии проектирования стандартных библиотек

Стандартная библиотека языка программирования является в настоящее время обязательной частью системной библиотеки. Все современные языки программирования нуждаются в поддержке имеющихся в них средств, причем поддержка им необходима именно в период выполнения программ, полученных путем

компиляции с этих языков. При проектировании любой стандартной библиотеки необходимо принимать во внимание ее основное назначение: быть именно стандартной библиотекой, то есть библиотекой средств, необходимых *для каждой реализации* данного языка программирования. Чтобы библиотека могла оказывать поддержку всем пользователям этого языка, она должна (***требования по составу***):

- обеспечивать поддержку свойств языка, например, управление памятью и предоставление информации об объектах во время выполнения программ;
- предоставлять информацию о зависящих от реализации аспектах языка, например, о максимальных размерах целых значений;
- предоставлять функции, которые не могут быть написаны оптимально для всех вычислительных систем на данном языке программирования, например, функции вычисления квадратного корня `sqrt()` или пересылок блоков памяти `memmove()`;
- предоставлять программисту нетривиальные средства, на которые он может рассчитывать, заботясь о переносимости программ, например, средства работы со списками, функции сортировки, потоки ввода/вывода;
- предоставлять основу для расширения собственных возможностей, в частности, соглашения и средства поддержки, позволяющие обеспечить операции для данных, имеющих определяемые пользователями типы, в том же стиле, в котором обеспечиваются операции для встроенных типов (например, ввод/вывод);
- служить основой и теоретическим базисом других библиотек.

При проектировании стандартных библиотек следует учитывать и ограничения на включение в ее состав некоторых (может быть полезных) элементов. Если какое-нибудь средство не оказывается необходимым для обеспечения хотя бы одного из перечисленных свойств, оно должно оставаться за пределами стандартной библиотеки. Такие средства должны быть реализованы в рамках дополнительных библиотек, предназначенных для решения более конкретных задач.

Средства стандартной библиотеки должны (***требования по свойствам компонентов***)

- быть важными и доступными для программистов разной квалификации, в том числе для создателей других библиотек;
- использоваться (прямо или косвенно) всеми программистами для решения всех задач, для которых предназначена библиотека (структуры данных и алгоритмы для работы с ними должны иметь *общезначимый* характер – стек, очередь, список, …, сортировка, поиск, копирование);
- быть настолько эффективными, чтобы у пользователей библиотеки не возникало потребности заново программировать библиотечные средства (*эффективность* не должна уступать “ручному” программированию);
- быть независимыми от конкретных алгоритмов или предоставлять возможность указывать алгоритм в качестве параметра;
- оставаться элементарными, чтобы не терять эффективности из-за излишних усложнений или попыток совместить различные функции в одной;
- быть удобными, эффективными и *безопасными* (устойчивыми к неправильному использованию) в большинстве типичных случаев

использования (использование библиотеки не должно провоцировать ошибки, а наоборот, снижать их вероятность);

- обладать достаточной полнотой (*завершенностью*) в той своей функциональности, которая включаются в библиотеку, чтобы ни у кого не возникало желания что-то заменить или доопределить;
- хорошо сочетаться друг с другом;
- обладать удобной и безопасной системой умолчаний;
- поддерживать общепринятые стили программирования;
- обладать способностью к расширению, чтобы *работать с типами, определяемыми пользователем так же хорошо, как и со встроенными (базовыми) типами (сочетаемость с базовыми типами данных и базовыми операциями)*.

Классическим примером проектирования библиотечных средств является пример библиотечной функции сортировки. В стандартной библиотеке языка программирования Си эта функция (в языке Си функция сортировки реализует алгоритм быстрой сортировки и называется *qsort()*) получает в качестве параметра функцию сравнения сортируемых элементов, а не использует для сравнения какую-либо операцию языка Си, например, операцию *'<'*. Тем самым, удается добиться некоторой общности, то есть возможности сортировать некоторые объекты (доступные с помощью указателей, передаваемых функции сортировки в качестве других параметров) не только по возрастанию, но и по убыванию и, вообще, по произвольным критериям. Это максимально возможная общность для языка Си:

```
void qsort (const void * base, size_t nmemb, size_t size,
            int (* compar)(const void *, const void *));
```

Однако учитывая, что затраты на вызовы функции при проведении каждого элементарного сравнения элементов могут в некоторых случаях становиться слишком большими, надо стараться найти более эффективное решение. Ведь в каждом конкретном случае (для каждого конкретного типа данных) можно найти такое эффективное решение. Достигнуть большего удается только в языке Си++, где критерий сравнения для обобщенной функции сортировки *sort ()* реализуется с помощью параметра шаблона. Чтобы обобщенная функция сортировки *sort()*, которая должна упорядочивать целые числа по возрастанию, производила упорядочения целых чисел по убыванию, достаточно привлечь понятие функционального объекта:

```
class IntGreater
{ public:   bool operator()(int x, int y) const { return x > y; } };
int main ()
{ int x [1024];
                .............                  // Инициализация
  sort (&x [0], &x [1024]);                    // Обычное упорядочивание
  sort (&x [0], &x [1024], IntGreater ());     // Упорядочивание по убыванию
}
```

При таком использовании эффективность по скорости выполнения будет такой же, как и при написании сортировки целых чисел вручную, а реально может оказаться и выше за счет удачного выбора алгоритма сортировки. Введением функциональных объектов достигается даже больший эффект: функция *sort()* может сортировать

объекты любой природы и сложности по свойственным только им критериям сравнения, причем сам алгоритм будет внешне выглядеть самым обычным образом.

Некоторые требования кажутся противоречащими друг другу (например, требование элементарности и удобства), но если что-то можно сделать по-настоящему удобным, не следует отказываться от этого, под предлогом недостаточной элементарности. Именно такие соображения послужили причиной включения в стандартную библиотеку Си++ таких функций, как генератор случайных чисел. Эти же соображения заставляют отбрасывать требования элементарности, если они приводят к неясным или опасным умолчаниям.

3.7. Средства конфигурирования

Одним из важнейших свойств современной системы программирования становится возможность этой системы участвовать не только в процессе разработки программных комплексов, но и в процессе их сопровождения. Часто сопровождение программ осложняется тем, что при широком использовании программ и наличии многих пользователей в практическом использовании одновременно находятся различные версии этих программ. В это же самое время разработчики могут продолжать свою работу над созданием и отладкой очередных версий.

Широкое внедрение программных систем в различные прикладные области привело к разнообразию способов применения этих систем. Поиски путей снижения зависимости программного обеспечения от конкретных параметров окружения, в котором это обеспечение разрабатывается и функционирует, привело к попыткам выносить описания параметров окружения, а также режимов формирования и использования программ за пределы этих самых программ.

Внедрение средств управления конфигурацией шло постепенно и к настоящему моменту можно выделить четыре способа управления конфигурацией программных комплексов. К указанным вариантам управления конфигурацией относятся:

- конфигурирование из командной строки,
- использование командных файлов,
- работа в интегрированных средах с проектами программных комплексов,
- использование систем управления версиями программных комплексов.

Работа в режиме командной строки предполагает последовательное обращение к тем или иным компонентам системы программирования и передачу им параметров в виде последовательностей символов, входящих в эту командную строку. Такой способ работы подразумевает высокую квалификацию пользователя системы программирования, который должен точно знать все компоненты конкретной системы программирования, необходимые для формирования нужной ему программы, необходимые им параметры, а также правильную последовательность их вызова. Управление сборкой программы ведется в этом случае “вручную”. Пользователь должен помнить варианты сборки программы, режимы, в которых выполнялось их формирование, имена файлов, в которых размещены результаты работы системы программирования.

Помощь в снижении трудоемкости формирования программных комплексов оказывает возможность использования командных файлов, содержащих последовательности вызовов компонентов систем программирования. Их применение освобождает программистов от запоминания многих технических деталей, которые

нужно точно знать только непосредственно в момент создания командного файла. Все вариации конфигураций можно получать, используя развитые языки управления заданиями современных операционных систем, например, возможности условного вызова программ системы программирования или значения переменных системного окружения.

Примыкает к такому способу управления конфигурацией работа в интегрированных средах разработки программного обеспечения. Такие системы умеют работают с "*проектом программного комплекса*", который включает в себя все файлы комплекса (как с текстами программ, так и с библиотечными модулями), а также режимы их обработки. Интегрированные среды позволяют гибко переходить от отладочной конфигурации комплексов к оптимизированной конфигурации, что облегчает процесс сопровождения уже разработанных программ.

Перечисленные способы управления обладают своими преимуществами (простотой или удобством), но имеют один существенный недостаток. Их использование возможно только в тех случаях, когда над разработкой или сопровождением программного комплекса работает один программист (если таких программистов много, они должны работать последовательно друг за другом, сообщая друг другу обо всех сделанных ими изменениях в программах). Однако в последнее время все чаще приходится видеть, что разработкой программ (тем более их сопровождением) занимаются целые коллективы разработчиков. Иногда эти коллективы работают в распределенном режиме в глобальной вычислительной сети.

Реальную помощь таким коллективам при управлении конфигурацией формируемых программных комплексов могут оказать только специально разрабатываемые системы управления версиями программ. Эти системы способны вести централизованные базы данных программных проектов, к которым обеспечен коллективный доступ со стороны многих разработчиков одновременно.

3.8. Системы управления версиями программных комплексов

В настоящее время многие системы программирования начали включать в свой состав системы управления версиями. Например, в составе систем Visual Studio компании Microsoft, имеется система Visual SourceSafe, позволяющая создавать базы данных версий, включать файлы в состав версий программных проектов, отслеживать историю их изменений, сравнивать различные версии между собой. Имеется также определенный выбор систем управления версиями, которые не привязаны жестко к какой-либо системе программирования, а могут работать с любыми из них, ведя базы данных или репозитории файлов, составляющих законченные программные комплексы.

Такие автономные системы управления версиями обычно удобны тем, что позволяют вести управления версиями особенно сложных программных комплексов – распределенных. Эти программные комплексы не только работают, но и создаются в распределенном окружении, в котором разные программисты работают с разными системами программирования, создавая относительно независимые компоненты единой сложной программы. В этих условиях особенно важно, чтобы работа с репозиторием файлов велась в сетевом режиме с возможностью одновременного доступа с многих рабочих мест. Такой сетевой репозиторий начинает играть роль программного сервера, а рабочие места оказываются клиентами этого сервера. При этом клиенты могут быть самыми разнообразными, они могут работать с разным аппаратным обеспечением, с разными операционными системами, с разными системами программирования. Необходимо лишь, чтобы система управления версиями

могла взаимодействовать со всеми ними по единым, понятным для всех правилам. Именно так и работают лучшие и наиболее используемые системы управления версиями, которые способны поддерживать до нескольких десятков операционных систем и нескольких тысяч клиентов, одновременно работающих над созданием единого программного комплекса.

Среди коммерческих систем наиболее широко используются системы управления жизненным циклом программ Peforce SCM (*Software Configuration Management*) и IBM Rational ClearCase. В задачу этих систем входит создание и изменение конфигураций программ, их комплексирование, регистрация поставок, а также обеспечение повторного использования программ. На клиентских местах разрешается использовать все наиболее распространенные операционные системы (Windows, UNIX, Linux, mainframe z/OS), а также наиболее современные системы разработки, включая Rational Application Developer, WebSphere Studio, Microsoft Visual Studio .NET, Eclipse.

Широко известны и свободно распространяемые системы управления версиями, среди которых лидером является система CVS (*Concurrent Versions System*). Серверная часть системы может работать под управлением любого варианта операционной системы UNIX – FreeBSD, Linux и др. Клиентские части работают под управлением UNIX систем, а также системы Windows.

Система CVS поддерживает историю дерева каталогов (репозитория) с исходным кодом, работая с последовательностью изменений. Каждое изменение в файлах репозитория маркируется моментом времени, когда оно было сделано, и именем пользователя, совершившим изменение. Обычно человек, совершивший изменение, также предоставляет текстовое описание причины, по которой произошло изменение. Система CVS может отвечать на такие вопросы:

- Кто совершил данное изменение?
- Когда они его совершили?
- Зачем они это сделали?
- Какие еще изменения произошли в то же самое время?

Точно так же, как книгу выписывают из библиотеки, следует сначала получить из репозитория рабочее дерево каталогов. Большинство этих файлов – рабочие копии исходных текстов. Однако самый первый подкаталог имеет другое назначение. CVS использует его для хранения дополнительной информации о каждом файле в этом каталоге, чтобы определять, какие изменения внесены в них с тех пор, как их извлекли из репозитория.

После того, как рабочее дерево каталогов создано, можно редактировать, компилировать и проверять находящиеся в нем файлы. Так как каждый разработчик использует собственный рабочий каталог, изменения, которые делает каждый из них в своем каталоге, не становятся автоматически видимыми всем остальным. Когда изменения будут проверены, их надо *зафиксировать* в репозитории и сделать доступными остальным.

Перед тем, как фиксировать изменения, необходимо, чтобы исходные тексты были синхронизованы со всеми изменениями, которые сделали остальные члены группы. Перед фиксацией система запускает текстовый редактор и просит ввести описание изменений.

Система CVS позволяет узнать, какие изменения внесли другие разработчики. Журнальные записи выводятся на экран в обратном хронологическом порядке, исходя из предположения, что недавние изменения более интересны.

Система CVS обращается с добавлением и удалением файлов так же, как и с прочими изменениями, записывая такие события в истории файлов. Фактически, система сохраняет историю каталогов вместе с историей файлов, однако, система не считает, что все созданные файлы должны оказаться под ее контролем; это не так во многих случаях. Например, не требуется записывать историю изменений объектных и выполняемых файлов, потому что их содержимое всегда может быть воссоздано из исходных файлов.

Чтобы удалить файл из проекта, его помечают для удаления. Фиксация помеченного файла не уничтожает историю этого файла – к ней просто добавляется еще одна редакция (“не существует”). В репозитории по-прежнему хранятся все записи об этом файле, и к ним можно обращаться по желанию.

CVS объединяет изменения, сделанные разными разработчиками. Ситуация проста, если изменения были совершены в разных участках файла, но может быть изменена одна и та же строка, что называется *конфликтом*. CVS не понимает семантики программы, она обращается с исходным кодом просто как с деревом текстовых файлов. Если один разработчик добавляет новый параметр в функцию и исправляет все ее вызовы, пока другой разработчик одновременно добавляет новый вызов этой функции, и не передает ей этот новый параметр, что определенно является конфликтом (два изменения несовместимы), то система CVS не сообщит об этом. Ее понимание конфликтов строго текстуально.

На практике, однако, конфликты случаются редко. Обычно они происходят потому, что два человека пытаются справиться с одной и той же проблемой, плохо взаимодействуя между собой. Правильное распределение задач между разработчиками уменьшает вероятность конфликтов.

Многие системы контроля версий позволяют разработчику *блокировать* файл, предотвращая внесение в него изменений до тех пор, пока его собственные изменения не будут зафиксированы. Блокировки уместны в некоторых ситуациях, но их использование не всегда лучше, чем использование CVS без блокировок. Изменения обычно объединяются без проблем, а разработчики иногда забывают убрать блокировку, в обоих случаях явное блокирование приводит к ненужным задержкам. Более того, блокировки предотвращают только текстуальные конфликты – они ничего не могут поделать с семантическими конфликтами типа вышеописанного, когда два разработчика редактируют разные файлы.

В настоящий момент активно ведется разработка нового проекта *Subversion,* учитывающего положительные стороны системы CVS. Он будет распространяться с исходными текстами по свободной лицензии.

3.9. Средства отладки и тестирования программ

Любая полезная программа может содержать ошибки даже после ее передачи пользователям. Чтобы уменьшить число ошибок в программах, еще на этапе программирования (до объединения с другими компонентами создаваемого комплекса) разработчики проводят над своими программами цикл отладки:

- расставляют операторы выдачи промежуточных результатов работы программы,

- исследуют содержимое памяти, занятой командами или данными тестируемой (отлаживаемой) программы,
- применяют автоматизированные средства отладки и тестирования (двоичные и символьные отладчики).

Первый метод предполагает полностью “ручное” управление отладкой. Реализуя его, программисты сами определяют места в программе и номенклатуру переменных, значения которых будут выдаваться во внешний файл (или на дисплей). Это один из самых первых методов отладки программ. Он до сих пор используется при отладке, как небольших программ, так и крупных отладочных комплексов. Иногда программисты оставляют отладочные операторы в текстах программ и после передачи этих программ пользователям. В таких случаях отладочные операторы вставляются в условные операторы, которые выполняются при включении специального отладочного режима выполнения программ.

Возможности второго метода – получения информации из содержимого областей памяти, связаны со способностью многих операционных систем выдавать такую информацию в момент завершения работы программы, в частности, при возникновении неперехватываемой исключительной ситуации. Большинство компиляторов и редакторов связей в своей работе формируют таблицы, в которых объектам программ сопоставлены зоны памяти целевой вычислительной системы. Такие таблицы обычно называются таблицами перекрестных ссылок (иногда в них указываются не только места, где определяются объекты программы, но и номера строк текста, в которых эти объекты используются). По сведениям, содержащимся в этих таблицах легко сопоставить выдаваемую операционной системой информацию и значения переменных программы.

Однако в современных системах программирования наиболее удобным и часто используемым компонентом, обеспечивающим быструю отладку программ, является отладчик, то есть программный компонент, который позволяет выполнять основные задачи, связанные с отслеживанием хода выполнения объектной программы.

Применение отладчика объединяет возможности двух других методов отладки и дополнительно позволяет:

- проводить пошаговое выполнение отлаживаемой программы на основе шагов по машинным командам, строкам текста или операторам входного языка (интеграция с текстовым редактором);
- выполнять отлаживаемую программу до достижения ею одной из заранее заданных точек остановки (или до курсора текста);
- выполнять отлаживаемую программу до возникновения ситуации, в которой оказывается истинным некоторое логическое выражение над переменными и адресами программы;
- проводить трассировку и обратную трассировку работы программы;
- выдавать диагностические сообщения в терминах входного языка отлаживаемой программы;
- просматривать (а иногда и изменять) значения переменных программы и содержимое областей памяти, занятых программой;
- изменять текст отлаживаемой программы (с помощью тектсового редактора) и продолжать отладку без полной перекомпиляции.

Первоначально отладчики выполнялись в виде автономных программных компонентов и представляли собой *двоичные* отладчики. Такое название возникло из-за того, что они работали с двоичным представлением программ, в точности соответствующим тому представлению, которое имеют программы, исполняемые аппаратурой.

Символьные отладчики позволяют вести отладку в терминах исходного языка, а наибольшую отдачу от них удается получать в тех системах программирования, где символьные отладчики интегрированы в общую среду разработки программ. В настоящее время отладчикам поддержка оказывается, как системой программирования, так и аппаратурой вычислительных систем, в системы команд которых обычно вводятся специальные команды, облегчающие работу отладчиков.

В интегрированных средах разработки символьные отладчики получили возможность более тесного взаимодействия с другими компонентами систем программирования, прежде всего с текстовыми редакторами, компиляторами и редакторами связей. От текстовых редакторов требуется помощь при расстановке точек остановов и определении деления текста на отдельные строки при пошаговом исполнении, от компиляторов и редакторов связей потребовалась возможность предоставлять отладчикам доступ к таблицам имен и адресов, к описаниям областей видимости. В интегрированных системах таблицы компилятора не уничтожаются после завершения компиляции, а сохраняются и используются в процессе отладки непосредственно. Многие интегрированные отладчики позволяют проводить редактирование текста программы прямо в процессе отладки, что подразумевает еще более тесное взаимодействие текстовых редакторов, компиляторов и отладчиков.

Современные отладчики позволили существенно повысить качество разрабатываемого программного обеспечения, однако полностью проблему наличия ошибок в программах они не решили. Именно поэтому на фазе разработки программных комплексов обязательно проводятся мероприятия по тестированию программ, которое представляет собой *процесс сравнения результатов работы программ с заранее рассчитанными результатами выполнения тестовых примеров*. В отличие от отладки тестирование не выявляет причины дефектов в программах, а лишь обнаруживает эти дефекты, которые связаны с несоответствием программы исходным требованиям и спецификациям.

Результатом тестирования является вовсе не доказательство отсутствия ошибок в программе. Единственное, что можно утверждать, это, что все выявленные на данном комплекте тестов ошибки исправлены, а других ошибок не обнаружено.

Стратегия тестирования, или методы тестирования — это систематические методы, используемые для отбора тестов, которые должны быть включены в тестовый комплект. Стратегия является эффективной, если тесты, включенные в нее, с большой вероятностью обнаружат ошибки тестируемого объекта. Эффективность стратегии зависит от комбинации природы тестов и природы ошибок, на поиск которых эти тесты направлены. Так как программа изменяется при исправлении ошибок и росте ее функциональности, типы ошибок, находимые в ней, меняются со временем, и, следовательно, меняется эффективность стратегии. Теоретически возможно, что стратегия по отношению к специфическим программам совершенствуется во времени, на самом деле эффективность большинства стратегий со временем убывает.

Стратегия поведенческого теста основана на технических требованиях. Например: тест всех характеристик, упомянутых в спецификации, выполнение всех

тестов, вытекающих из требований к программам и т. д. Тестирование, выполняемое с помощью стратегии поведенческого теста, называется *поведенческим тестированием*. Поведенческое тестирование называется также *тестированием черного ящика*. Для поведенческого тестирования также используется термин *функциональное тестирование*. При поведенческом тестировании (в принципе, но не на практике) не обязательно знать, как объект сконструирован.

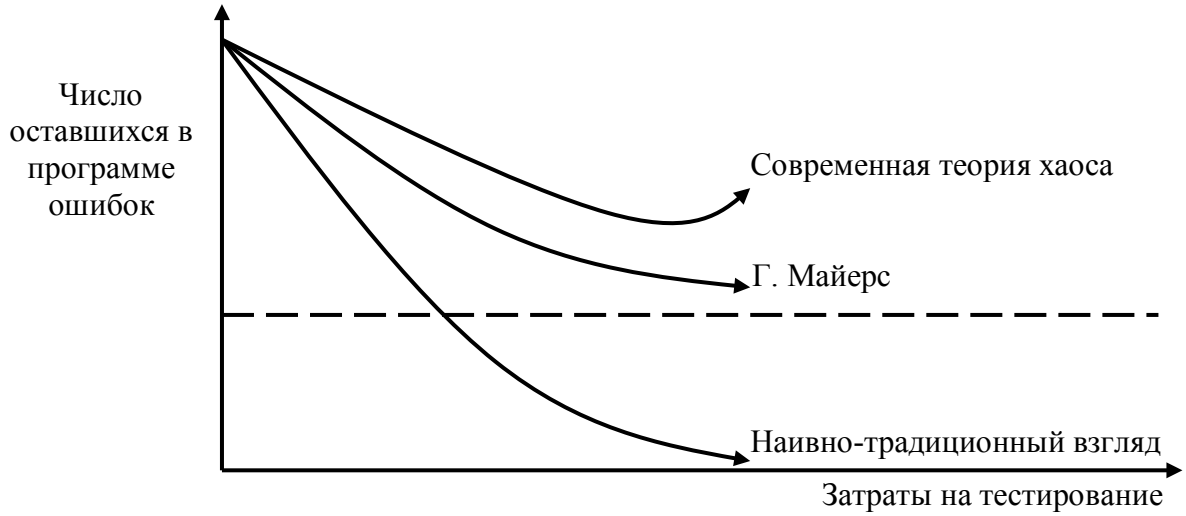
Стратегия структурного теста определяется структурой тестируемого объекта. Например: выполнение каждого оператора программы по меньшей мере один раз, выполнение каждой ветви программы по меньшей мере один раз, использование всех объектов данных, выполнение каждой команды объектной программы, полученной при компиляции. Тестирование, выполненное с помощью стратегии структурного теста, называется также тестированием прозрачного ящика или *тестированием белого ящика*. Стратегия структурного теста требует полного доступа к структуре объекта — то есть к исходной программе.

Стратегия гибридного теста является комбинацией поведенческой и структурной стратегий. Поведенческая, структурная и гибридная стратегии не противоречат друг другу, и ни про одну из них нельзя сказать, что она лучше других. Модули и низкоуровневые компоненты часто тестируются с помощью структурной стратегии. Большие компоненты и системы в основном тестируются с помощью поведенческой стратегии. Гибридная стратегия полезна на всех уровнях. Не существует лучшей стратегии, так как полезность стратегии зависит от природы тестируемого объекта, природы ошибок объекта и уровня знаний о программе.

Тестирование проводится не только на той стадии разработки программ, которая специально для этого предназначена, но и на предшествующих стадиях – при автономной отладке программ, еще до объединения их в единый программный комплекс. Такое тестирование следует называть *автономным*. Его обычно проводят сами разработчики, которые проверяют точное соответствие программы выданной им спецификации. Автономное тестирование детально проверяет каждый разработанный программный компонент. Проверка ведется с точки зрения разработчика: проверяется, насколько данный компонент соответствует своей спецификации. На этой стадии совершенно не принимаются во внимание аспекты взаимодействия данной программы с другими программами комплекса.

Комплексное тестирование призвано проверить все аспекты работы программы от правильности взаимодействия внутренних программных компонентов до правильности взаимодействия программного комплекса с его пользователями. Его необходимость подтверждается многочисленными исследованиями действующих программных систем. Если первоначально многие считали, что со временем число ошибок в программе может быть уменьшено до 0, то позднее было осознано, что некоторое число ошибок в программах может оставаться весьма длительное время. Исправление одной ошибки с большой вероятностью (от 2 до 50 процентов) вносит другую. Некоторые современные теории предсказывают, что по мере приближения к теоретическому минимуму числа ошибок исправления, вносимые в программы, могут приводить к появлению даже большего числа ошибок, чем было ранее:

Во время *пользовательского тестирования* результаты работы программы проверяются с прикладной точки зрения. На этом этапе необходимо проверить является ли разработанный программный продукт именно тем, что было заказано, все ли заказанные свойства реализованы, соответствует ли результат поставленным целям.



Техническое тестирование дает возможность проверить безопасную и эффективную работу созданной программы в нормальном и пиковом режимах ее использования. Функциональность на этом этапе проверяется только в смысле ее влияния на важнейшие технические параметры программы, например, на время реакции системы на запрос пользователя.

Особенно важно при проведении тестирования программ иметь заранее составленные сценарии тестирования и тестовые примеры, которые должны охватывать все варианты возможного поведения и реакции программы, как в режиме нормальной работы, так и в случае возникновения необычных ситуаций. Например, компилятор следует тестировать, подавая ему на вход не только правильные программы, но и программы, содержащие все возможные ошибки.

В настоящее время доступно множество специальных средств тестирования программ, обеспечивающих управление тестированием, высокую скорость тестирования и повторяемость тестов. Особенно важны такие средства автоматизированного тестирования для тестовых сценариев, предусматривающих большое количество взаимодействий и наличие пиковых нагрузок на программы (например, для сетевых приложений с центральным серверным звеном). Повторяемость тестов важна для повторных проверок после внесения исправлений. Некоторым недостатком имеющихся средств тестирования является их высокая стоимость и большие затраты времени на подготовку полноценных тестовых примеров.

Важной в тестировании является возможность проведения *регрессивного тестирования*. Регрессивные тесты, повторяемые после каждого исправления программы, позволяют убедиться, что функциональность программы, не связанная с внесенным исправлением, не затронута этим исправлением и не утрачена из-за него.

3.10. Профилировщики

Один из способов повысить эффективность работы программы – провести ее профилирование, то есть определить время, затрачиваемое на выполнение отдельных ее фрагментов. Профилировщик целесообразно использовать, чтобы выявить функции, которые требуют большого времени, затем определить, почему и как они вызываются, и поискать способы минимизации их использования. Часто бывает полезно задаться вопросом, а требуется ли вызывать функцию столько раз. Поскольку программы часто имеют несколько уровней, может оказаться, что наиболее трудоемкие функции вызываются неявно. В этом случае важно определить, какие из функций высшего

уровня являются ответственным за такие обращения. Часто профилировщик помогает выявить проблемы, которые могут быть решены:

- отказом от лишних вычислений, которые могут быть следствием невнимательности;
- корректировкой алгоритма, чтобы избежать вызова неэффективных функций;
- отказом от многократных повторных вычислений путем хранения результатов для последующего использования.

Современные профилировщики способны измерять время выполнения каждой функции программы, а иногда и каждой ее строки. Профилировщики позволяют разработчикам выявлять ошибки в программах, обнаруживать временные конфликты и множество других "узких мест" программ, влияющих на производительность системы, которые традиционные отладчики просто не замечают.

Особенно профилировщики важны при отладке программ, связанных с работой в реальном масштабе времени, ошибки в которых могут проявляться не сразу, а совсем в другом месте. Профилировщик позволяет увидеть, где в действительности находится ошибка, снижающая производительность программы, и устранить ее.

Профилировщик аналогичен программному логическому анализатору, способному выдавать огромные объемы информации. Профилировщик позволяет разработчику точно настраивать поведение системы в условиях реальной эксплуатации и визуализировать события для быстрого обнаружения проблемы. С помощью профилировщика можно разрешать временные конфликты, выявлять точки взаимной блокировки процессов, исправлять логические ошибки, выявлять скрытые ошибки в программах, собирать данные о взаимодействии процессов, проходящих в системе, фиксировать времена событий, определять участвующие в работе программные модули.

Профилировщик позволяет получать информацию о вызовах функций ядра операционной системы, аппаратных прерываниях, состояниях потоков ввода/вывода, сообщениях и деятельности планировщика. Благодаря возможностям фильтрации событий и вывода информации о них на экран разработчик может выделять те участки программ, которые вызывают снижение производительности, и видеть полную картину взаимодействия процессов. Наиболее известным и широко распространенным (хотя и не самым лучшим) профилировщиком является программа ***prof***, входящая в состав операционных систем UNIX.

3.11. Справочные системы

С самого своего появления системы программирования снабжались огромным количеством документации разного уровня: для системных программистов распространялись документы с инструкциями по установке и настройке компиляторов и библиотек, для обычных пользователей-программистов с системами программирования поставлялись документы с описаниями языков программирования, описаниями библиотечных функций, перечнями фиксируемых компиляторами ошибок, правил запуска отдельных компонентов и многие другие.

С распространением интегрированных систем документация также стала поставляться по-новому. В состав систем программирования стали включаться справочные системы, представляющие собой обширные базы данных с включенными в

них сведениями по всем интересующим пользователей вопросам. Полезной функцией современных систем программирования является возможность получения справочной информации, которая может выдаваться по трем направлениям:

- справки по семантике и синтаксису используемого языка программирования;
- справки по операционной системе и системе программирования;
- справки по библиотечным компонентам, входящим в систему программирования.

Базы данных справочных систем дополняются индексами, облегчающими поиск информации. Как и в обычных библиотеках, индексы строятся и по алфавиту заголовков, и по их тематической принадлежности.

Для создания справочных систем разрабатывается специальный системный инструментарий, который можно использовать при разработке собственных программ, обеспечивая и для их пользователей такой же сервис контекстно-зависимых справок, который предоставляется их разработчикам. Поэтому справочными системами в настоящее время снабжаются не только системы программирования и другие компоненты системного программного обеспечения, но также и системы прикладных программ.

В последнее время стал применяться метод удаленной работы с документацией: сами тексты документов не тиражируются и не передаются пользователям, но становятся доступными через Интернет. Такой подход позволяет компаниям поставщикам систем программирования своевременно вносить все необходимые исправления, поддерживая актуальность документации.

Однако простой демонстрацией текстов документов справочные системы современных систем программирования не ограничиваются. Интеграция всех компонентов систем программирования позволила обеспечить совместную работу текстовых редакторов, компиляторов и справочных систем. Справочную информацию теперь можно получать, не только обращаясь к базе данных документов и проводя поиск по индексам, но и выполняя быстрый контекстный поиск необходимой информации.

Работая в текстовом редакторе, пользователь может указать в тексте программы некоторый идентификатор и, нажав некоторую комбинацию клавиш на клавиатуре, сразу получить информацию об объектах, имеющих такое имя. Например, поставив курсор редактора на имя `fprintf`, пользователь сразу получает справку по всем функциям форматного ввода/вывода и способам задания форматов. Работа с контекстными справками не может заменить систематического изучения важнейших документов (например, описания языка), но значительно облегчает работу опытным пользователям, желающим быстро вспомнить знакомую им информацию.

4. Краткий обзор современных систем программирования

4.1. Компонентный подход и визуальное программирование

К настоящему времени разработано и внедрено в практику программирования большое число систем разработки и сопровождения программ, написанных на языках, поддерживающих принципы объектно-ориентированного программирования. При этом простым включением объектно-ориентированных языков в состав систем программирования дело обычно не ограничивается. Все более явственно просматривается тенденция строить сами системы программирования в объектно-ориентированном стиле, то есть включать в состав этих систем средства, позволяющие вести проектирование программ (и даже само "*техническое*" программирование) на основе принципов объектно-ориентированного подхода. Наиболее ярко эти тенденции проявляются в последовательном применении *компонентного и визуального программирования*.

Термин "*компонент*" также многозначен, как и многие другие термины, используемые в литературе по программированию. Он может соответствовать термину "*программный модуль*", обозначая в таком случае архитектурный компонент программной системы – некоторый абстрактный элемент структуры программы, выделенный для решения некоторых конкретных подзадач в рамках общего назначения системы и имеющий некоторый фиксированный интерфейс взаимодействия с другими программными модулями (подсистемами). Другой смысл вкладывается в этот термин, когда говорят о компонентах, как об "*элементах сборки*" программы – некоторых ее относительно небольших (иногда незавершенных) фрагментах или заготовках, которые можно извлекать из библиотек (вообще говоря, из любых библиотек, но чаще при этом имеются в виду разрабатываемые специально для подобного стиля программирования "*библиотеки компонентов*"). Наконец, в распределенных программных системах (серверах приложений, сетевых службах) под компонентом часто понимают определенную *функционально законченную и самодостаточную структурную единицу программы*, обладающую точно описанным интерфейсом и даже некоторую независимость от других подобных компонентов этого же программного комплекса. Например, в сервере приложений J2EE имеется специальный компонент, позволяющий создавать сервлеты, представляющие собой классы языка программирования Java, реализующие обработку запросов по протоколу взаимодействия HTTP и генерацию ответных сообщений в формате этого протокола, и взаимодействовать с созданными сервлетами. Этот компонент может присутствовать в конкретной программной системе или отсутствовать в ней (если взаимодействие через Интернет не является задачей системы), благодаря точному описанию интерфейса и независимости от других компонентов сервера приложений, он может даже использоваться в других системах, не обязательно написанных на языке Java.

Чаще всего, когда говорят о ***компонентном подходе*** к построению современных систем программного обеспечения, имеют в виду именно эту, последнюю трактовку термина "*компонент*". Компонентная разработка предлагает строить такие системы последовательно из отдельных элементов — "*компонентов*", каждый из которых, в свою очередь, может рассматриваться как отдельная программная система.

Компоненты отличаются от классов объектно-ориентированных языков. Класс определяет не только набор реализуемых интерфейсов, но и саму их реализацию. В описании компонента реализация интерфейсов обычно не зафиксирована. Класс описан на определенном языке программирования, компонент же не привязан ни к какому

языку (если его компонентная модель этого не требует, компонентная модель является для компонентов тем же, чем для классов является язык программирования). Наконец, обычно компонент является более крупной структурной единицей, чем класс, реализация компонента часто состоит из нескольких тесно связанных друг с другом классов. Понятие компонента является более узким, чем понятие программного модуля. Основное содержание понятия модуля — наличие четко описанного интерфейса между ним и его окружением. Использование компонента подразумевает возможность поставки или удаления компонента отдельно от всей остальной системы. Компоненты могут и разрабатываться отдельно, однако они должны следовать правилам определенной компонентной модели и реализовывать достаточно важные для пользователей функции.

Визуальным называется такой стиль программирования, который предусматривает создание приложений с помощью наглядных средств. Используя приемы визуального программирования, программист не создает тексты программ, а показывает, что должно получиться в результате. Например, многие системы программирования позволяют строить на экране монитора графические формы, состоящие из отдельных более или менее независимых элементов ("*компонентов*") – графических кнопок, надписей, диаграмм, окон для ввода и вывода информации и так далее. Тексты программ, управляющих такими формами, генерируются автоматически с помощью визуального прототипа соответствующего компонента на основе используемых в системе библиотек компонентов. Как и компонентное программирование, визуальное программирование основывается на объектно-ориентированном подходе, поскольку с каждым наглядным элементом связан какой-нибудь класс или чаще целый набор классов, описывающих интерфейсы и реализации методов, с помощью которых осуществляется взаимодействие с данным элементом. Визуальное программирование широко используется в системах создания приложений. Некоторые из таких систем кратко описаны далее.

4.2. Системы программирования компании Borland

Фирма Borland и ее основатель Филипп Канн (Philippe Kahn) оказались одними из пионеров создания современных представлений о системах программирования персональных ЭВМ. Начав с проектирования транслятора с языка программирования Паскаль, компания Borland за непродолжительное время создала несколько серий прекрасных систем программирования на языках Паскаль, Си, Си++, Пролог и других, включая язык ассемблера для персональной ЭВМ на базе процессоров iAPXi86.

4.2.1. Turbo Pascal

Наиболее известной серией этих систем, продолжающейся до сих пор, является самая первая из них, связанная с программированием на языке Паскаль в самой первой операционной системе персональных ЭВМ – DOS. В настоящее время компанией Borland для DOS выпускается несколько систем программирования: Turbo Pascal версии 7.0, Borland Pascal for DOS и Delphi. Все эти системы являются наследницами систем, выпускавшихся с 1983 года, и во многом совместимы с ними как по самому языку программирования, для которого они разработаны, так и по возможностям интегрированной среды разработки и отладки программ.

Начиная с версии 4.0 системы Turbo Pascal, появившейся примерно 15 лет назад, язык программирования, использованный в них, значительно отличается от того языка

Паскаль, классическое описания которого, дано его автором Никлаусом Виртом и другими в их многочисленных работах.

Компания Borland опередила в своей практической работе разработку теории модульного, структурного и объектно-ориентированного программирования, которую вел Никлаус Вирт, осуществляя переход от языка Паскаль к языку Модула-2.

Концепция стандартных модулей, предложенная в языке под названием Turbo Pascal, некоторыми своими чертами напоминает подход, описанный в языке Модула-2 с помощью модулей определений и модулей реализации. В Модуле-2 парадигмы модульного и структурного программирования нашли свое почти идеальное воплощение. В этом языке уже намечались те черты, которые впоследствии были явственно выделены в языках, ориентированных на работу с объектами. Однако развитие систем программирования компании Borland шло по другому пути.

Было признано более целесообразным не переходить к использованию другого языка программирования, пусть и напоминающего классический Паскаль, как Модула-2, но все же отличающегося от него в некоторых важных чертах, а сохранить и расширить сам Паскаль, дав ему новую жизнь в изменившихся условиях. Вместо модулей определений и реализации в языке Turbo Pascal появились модули, имеющие разделы интерфейса и реализации. Появились и элементы объектно-ориентированного программирования, в частности, привязка процедур и функций к описаниям сложных объектов. Однако последовательного внедрения принципов объектно-ориентированного программирования в системы программирования для DOS компанией Borland произведено не было, это было осуществлено только в другой серии систем программирования – в системах программирования для операционных систем Windows.

Указанные особенности языка программирования Turbo Pascal до некоторой степени повлияли и на саму систему программирования Turbo Pascal. Эта система в наибольшей степени автоматизирует сам процесс программирования (написания программ) и отладки программ, в ее состав входят

- Многооконный экранный редактор текстов, позволяющий
  - быстро отыскивать в архиве файловой системы необходимые тексты программ и их составных частей (например, модулей),
  - показывать в экранных окнах и редактировать тексты.
  - сохранять их в архиве для последующего использования.
- Транслятор с языка программирования Turbo Pascal с подсистемой фиксации и индикации синтаксических ошибок в текстах.
- Набор стандартных системных модулей для работы с основными внешними устройствами ЭВМ.
- Компоновщик модулей, позволяющий собирать из ранее оттранслированных модульных фрагментов программ и библиотечных модулей полноценные исполняемые программы.
- Отладчик программ, позволяющий отлаживать программы в пошаговом режиме и просматривать промежуточные значения внутренних переменных программ, состояния памяти ЭВМ. Отладчик помогает устанавливать в программе контрольные точки, на которых при выполнении программы может быть осуществлена остановка программы. После такой остановки программист имеет возможность исследовать значения переменных, а затем (если это необходимо) исправить текст программы и продолжить

выполнение программы в пошаговом или в обычном режиме до выхода на следующую контрольную точку или до конца работы программы.

Благодаря тому, что системы программирования Turbo Pascal работают под управлением операционной систему DOS, в состав этих систем удалось включить стандартные модули, работающие с ресурсами и аппаратными элементами ЭВМ (дисками и прерываниями – DOS, графическим экраном – Graph, текстовым экраном и звуком – Crt и др.) в режиме полного управления. Такие возможности иногда являются просто необходимыми, например, если компьютер включен в состав сложной системы управления и к нему подключены нестандартные внешние устройства. В других случаях большие возможности DOS по управлению компьютером превращаются в ненужные усложнения и представляют собой излишнюю нагрузку на программиста, заставляя вручную программировать использование различных аппаратных и программных ресурсов, совместно используемых различными модулями программ. В таких случаях более выгодно переходить к работе в других системах программирования, получивших торговую марку Delphi.

4.2.2. Delphi

Появление новой серии программных продуктов фирмы Borland, с маркой Delphi, фактически привело к завершению ранее выпускавшейся серии Turbo Pascal. Язык Паскаль в Delphi был еще раз существенно дополнен, точнее переработан. В дополнение к аспектам модульного программирования к нему были добавлены практически все признаки объектно-ориентированных языков. Чтобы не вводить более в заблуждение программистское сообщество, разработчики стали называть новый вариант языка Object Pascal, а после внедрения в системы Delphi новых сетевых технологий и очередной модернизации языка появилось наименование язык Delphi.

Как и язык Паскаль, языки Object Pascal и Delphi не ориентированы на какую-либо специальную прикладную область, а являются универсальными языками. Наиболее близким их аналогом, широко распространенным во всем мире, является объектно-ориентированный язык программирования Си++. Этот язык можно даже рассматривать не как аналог, а как образец, используемый разработчиками новых объектно-ориентированных языков на базе языка Паскаль. Язык Си++ появился раньше языка Object Pascal и раньше языка Delphi. Именно на примере Си++ были продемонстрированы принципы объектно-ориентированного программирования и его достоинства. Другим языкам долгое время приходилось лишь следовать указанным путем. До настоящего времени в языке Object Pascal отсутствуют или выглядят чрезмерно усложненными некоторые элементы, реализованные в Си++ легко и просто, хотя в нем есть и объекты, и классы, и наследование их свойств, и полиморфизм.

Изменение свойств основного языка программирования, являющего ядром системы программирования повлияло на свойства самой системы программирования. Если системы Turbo Pascal просто предоставляли удобный пользовательский интерфейс для создателей программ, то системы Delphi прямо ориентированы на “визуальное” программирование. Это стало возможным благодаря переходу от использования MS-DOS к работе в операционной системе нового поколения Windows и постепенной стабилизации и стандартизации возможностей, предоставляемых этой системой пользователям, в частности создателям систем программирования.

Система Delphi – это не просто интегрированная система программирования, а интегрированная среда разработки (*IDE – Integrated Development Environment*),

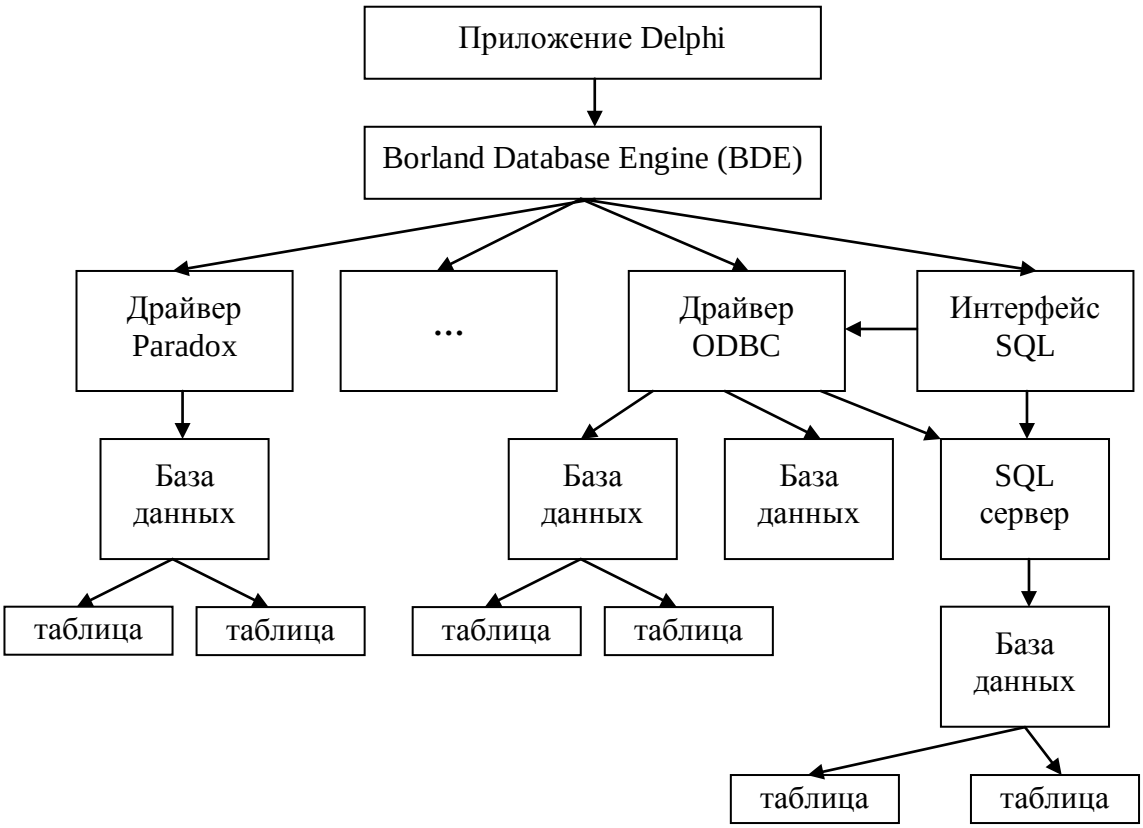
предоставляющая программистам формы с огромным количеством различных компонентов. Проектируя и программируя пользовательский диалог, программист просто размещает компоненты на формах, придавая этим компонентам необходимые атрибуты – размеры, цветовые оттенки, делая надписи наиболее удобными шрифтами. Результат проектирования диалога непосредственно виден на экране ЭВМ, никакой компиляции не требуется. Одновременно система автоматически создает на языке программирования Object Pascal или Delphi программу, которая представляет собой описание выбранных программистом форм и размещенных на них компонентов. В дальнейшем программист имеет возможность, редактируя автоматически созданные таблицы свойств компонентов, изменять значения отдельных свойств, полученные компонентами форм по умолчанию, и создавать обработчики некоторых событий, связанных с компонентами форм. Примером таких событий может быть событие, возникающее при нажатии на левую клавишу мыши в тот момент, когда указатель мыши находится на экране ЭВМ непосредственно в зоне, отведенной для изображения некоторого компонента формы, например, изображения какой-либо клавиши.

Таким образом, проектирование и программирование диалога с пользователем сводится к определению и установке значений свойств некоторых компонентов форм и программированию обработчиков событий. Номенклатура готовых компонентов постоянно расширяется, библиотеки компонентов (*VCL – Visual Component Library*), настолько обширны, что позволяют легко создавать самые сложные диалоги. Такая визуальная технология получила наименование технологии “быстрой разработки приложений” (*RAD – Rapid Application Development*).

Программирование не сводится к проектированию диалога, поэтому в системах Delphi имеется еще множество возможностей, позволяющих существенно облегчить процессы программирования и отладки систем программного обеспечения. К таким возможностям, прежде всего, можно отнести возможность прямого использования одного из самых популярных в мире языков программирования – языка Паскаль (хотя и в виде языков Object Pascal и Delphi). Благодаря этому системы Delphi могут использоваться для решения огромного количества задач, в которых обосновано применение универсального языка программирования, обеспечивающего контроль типов данных и развитые возможности для написания вычислительных задач (указатели, многомерные массивы, рекурсивные процедуры, встроенные функции).

Существенным дополнением к возможностям обычных систем программирования в системах Delphi является наличие средств подключения и работы с локальными и распределенными системами баз данных. В состав самых первых систем программирования Delphi уже был включен процессор баз данных компании Borland (*BDE – Borland Database Engine*). Процессор BDE является посредником между прикладными программами и базами данных. Для уменьшения зависимости прикладных программ от конкретной базы данных этот процессор предоставляет пользователям единый интерфейс, благодаря чему при смене базы данных приложение остается вполне работоспособным. В состав процессора BDE входят драйверы систем управления базами данных (СУБД) для некоторых, наиболее распространенных на персональных ЭВМ СУБД: Microsoft Access, FoxPro, Paradox, dBase и некоторых других. В состав BDE входит также драйвер ODBC (Open Database Connectivity), разработанный для включения в системы Delphi возможностей, предоставляемых для связи с базами данных системам фирмы Microsoft. Тем самым (хотя и с некоторой

потерей эффективности), системы фирмы Borland могут работать с любыми базами данных, которые подключаются с помощью ODBC:



Сам по себе программный продукт ODBC по своему назначению и своим функциям аналогичен процессору BDE, но разработан компанией, конкурирующей с компанией Borland на рынке систем программирования – компанией Microsoft. Этот продукт был подключен к системе BDE по той причине, что в состав многочисленных продуктов компании Microsoft, поддерживающих офисную автоматизацию, включена также и поддержка ODBC, поэтому для ODBC созданы драйверы почти всех СУБД, что обеспечила работу с этими системами и в системе Delphi/BDE.

Процессор BDE поддерживает стандартный язык запросов SQL, который позволяет выдавать запросы к серверам баз данных Oracle, Sybase, Microsoft SQL, Interbase и другим. Это обеспечивает возможность работать не только с локальными базами данных, то есть с базами, размещенными на тех же компьютерах, что и основная прикладная программа, но и с распределенными базами данных в системах с архитектурой “клиент/сервер”.

В рамках расширения возможностей и эффективности работы системы программирования Delphi в составе операционной системы Windows компания Borland начала использование разработанной в Microsoft технология ActiveX Data Objects (ADO). Этот пользовательский интерфейс включает как реляционные, так и не реляционные базы данных, электронную почту, поддержку системных, текстовых и графических файлов. Связь с данными осуществляется с помощью технологии OLE DB. Использование ADO позволяет решить проблемы локализации данных (в частности, проблемы многоязыковых данных и шрифтов), и полностью отказаться от специализированного процессора BDE, поскольку поддержка ADO включена в типовую поставку операционной системы Windows.

Сама компания Borland также продолжила развитие собственной системы программирования в части поддержки работы с базами данных. Ею были разработаны технологии InterBase Express (IBX) и dbExpress, которые полностью заменили процессор BDE. В настоящее время компания рекомендует пользоваться не процессором BDE, а более современной технологией dbExpress, которая использует для получения данных исключительно запросы SQL.

4.2.3. C++ Builder

По своим возможностям Си++ Builder практически полностью пересекается с системами Delphi: и здесь и там использован метод технического проектирования программы, называемый визуальным программированием. Отличие от систем Delphi в данном случае заключается в том, что базовым языком данной системы программирования является язык Си++.

В системе программирования Си++ Builder явно прослеживается тенденция построения многоязыковых систем программирования. В большой степени это связано с входящей в состав системы Си++ Builder библиотекой визуальных компонентов VCL. Первоначально эта библиотека была разработана для систем программирования на Паскале, то есть систем Delphi, а позднее была перенесена в Си++ Builder. Наличие этой библиотеки в разных системах программирования позволяет пользователю писать программу, состоящую из фрагментов, написанных на разных языках. При этом программист имеет возможность пользоваться одними и теми же абстракциями. В то же время системы Delphi и Си++ Builder – это разные системы, поэтому реально создавать многоязыковые программы с их помощью нелегко.

Библиотека VCL замечательна еще и тем, что она полностью построена на принципах объектно-ориентированного программирования и единой иерархии классов с общим базовым классом TObject, находящимся в основе этой иерархии. Все классы VCL являются потомками этого класса. Наличие общего корня библиотеки классов позволяет использовать полиморфизм для реализации общих алгоритмов и структур данных. По своей функциональности библиотека VCL в значительной степени пересекается с другими широко распространенными библиотеками Си++, в частности, со стандартной библиотекой Си++ и со стандартной библиотекой шаблонов.

4.3. Системы программирования компании Microsoft

К наиболее распространенным системам программирования для настольных ЭВМ относятся системы, выпускаемые компанией Microsoft. Весь комплекс программ, поставляемых компанией Microsoft, следует называть единой операционной средой, предназначенной для разработчиков системного программного обеспечения, прикладного программного обеспечения.

Системы, выпускаемые компанией Microsoft, выполнены в едино м стиле, их интерфейс хорошо продуман. Многооконный интерфейс позволяет одновременно видеть различную информацию о создаваемой, тестируемой или исполняемой программе. Все системы имеют развитый отладчик, который работает в терминах базового языка программирования (Basic/Си++/Язык ассемблера). В любой момент времени программист может проверить состояние того или иного объекта данных, а в процессе отладки можно даже менять некоторые значения переменных и сразу продолжать работу с точки остановки программы без дополнительной перекомпиляции.

4.3.1. Visual Basic

Наиболее известной и распространенной системой программирования компании Microsoft является система Visual Basic. Язык Basic, в том виде, каким он предстает в современных системах программирования, сильно отличается от своей первоначальной версии. В настоящее время это объектно-ориентированный язык, обладающий всеми возможностями других, более новых языков программирования, но оставшийся весьма простым для изучения, благодаря простым изобразительным средствам. Процесс создания диалоговых форм и расстановки на них элементов управления диалогом благодаря визуальному подходу стал несложным и понятным. Система программирования в процессе создания форм автоматически создает программу на языке Visual Basic. Собственно программы, как таковой, практически и не требуется. Требуется лишь написать процедуры реакции на события, возникающие при работе программы.

Заготовки процедур, реализующих такие реакции, вставляются в текст программы автоматически, пользователю остается лишь наполнить их реальным содержанием, что очень просто, учитывая существенное разделение первоначальной сложной задачи проектирование диалога на множество простых подзадач. Значительное облегчение пользователям приносят также библиотеки стандартных форм и элементов управления (“библиотеки компонентов”), а также развитая система оказания помощи пользователям, снабженная огромным количеством примеров. В процессе работы пользователь может свободно переключаться из режима просмотра и редактирования текста программ в режим графического редактирования форм. Пользователь может добавлять новые формы, расставлять на них новые элементы управления, менять их размеры и свойства. Отладчик, встроенный в систему программирования, работает в терминах языка Visual Basic, поэтому отладка программ не представляет особой сложности.

В целом, систему Visual Basic можно определить, как инструментальную среду для разработки самых различных программных продуктов. Создаваемые в этой интегрированной инструментальной среде программы обладают свойством автономности и в состоянии после завершения разработки функционировать в отрыве от самой среды. Следует только помнить о необходимости сопровождать распространение программы, написанной в системе Visual, библиотеками, отслеживая совместимость версий стандартных библиотек фирмы Microsoft с версией созданной программы. Отсутствие нужной библиотеки, а иногда и небольшого системного файла в системном каталоге неминуемо заблокирует работу программы.

Особенно удобно работать в системе Visual Basic с приложениями Microsoft Office. Каждое отдельное офисное приложение (текстовый процессор Word, электронные таблицы Excel, система управления базой данных Access, система электронной почты Outlook и другие) представлено в системе библиотекой встроенных иерархических классов. С помощью методов этих классов программист непосредственно из программы на языке Basic может активизировать приложения, создавать документы, редактировать их, передавать из одного приложения в другое и делать многое другое. Например, можно создать программу, которая (с помощью созданных обычным для Visual Basic способом диалоговых форм) будет запрашивать у пользователя некоторые данные (например, личные данные клиента) и формировать документ установленного образца. Полученный документ можно затем редактировать в диалоговом режиме с помощью текстового процессора Word. Из программы можно

также автоматически передавать нужные данные в систему электронных таблиц Excel, а также автоматически отправлять их по электронной почте, используя адреса, хранящиеся в базе данных Access.

4.3.2. VBA

Для создания интегрированных приложений компанией Microsoft предлагается система, называемая Visual Basic for Applications или VBA. Чтобы ею воспользоваться, на ЭВМ требуется устанавливать не дополнительные системные компоненты, а только продукт Microsoft Office. В отличие от системы Visual Basic, система VBA не предназначена для создания автономных программ. Программные продукты, созданные в инструментальной среде VBA для конкретного приложения, могут быть запущены только из этого приложения и функционируют только вместе с ним.

В системе Visual Basic язык Basic выступает как компилирующий язык, создающий исполняемые файлы, в то время, как в системе VBA используется интерпретатор, который интерпретирует программу, хранящуюся в текстовом виде на языке Basic или в виде текста на промежуточном языке.

Система VBA представляет собой единую комплексную среду для поддержки разработки сложных прикладных программ и автоматизированных документов. С ее помощью сложные прикладные программы могут разрабатывать не только профессиональные программисты, но и квалифицированные пользователи приложений. Сервис, который система предлагает, включая удобный интерактивный отладчик, вполне для этого достаточен.

4.3.3. Visual C++

Некоторым аналогом системы VBA является система программирования Visual C++, базовым языком в которой является язык Си++. В системе Visual C++ имеется полный набор библиотек, позволяющих выполнять все виды работ, которые можно выполнять и в системах Visual Basic и VBA. Возможность использовать язык Си++ превращает эту систему программирования в инструмент, позволяющий создавать не только обычные офисные приложения, но и решать другие задачи.

В систему программирования встроен удобный интерактивный отладчик, работающий в терминах языка Си++ или языка ассемблера и позволяющий одновременно видеть на экране тексты различных фрагментов программ, значения переменных и регистров центрального процессора ЭВМ, стек вызовов процедур и другую необходимую при отладке информацию. Отладчик позволяет менять значения переменных, что иногда помогает программисту проверить гипотезу о причинах неправильного поведения программы, а впоследствии и исправить программу.

Как и для других программных продуктов компании Microsoft, при работе в системе Visual C++ доступна вся справочная информация, как о самой системе, так и о языке Си++, библиотечных функциях и операционной системе Windows. Справочник снабжен большим количеством примеров, которые часто позволяют повысить эффективность, как процесса программирования, так и процесса работы уже подготовленной программы.

Система Visual C++, как и системы Visual Basic и VBA, в настоящее время уже считается компанией Microsoft устаревающей. В последние комплекты поставок программного обеспечения все эти системы уже не входят, однако, то широкое распространение, которое они нашли, доказывает их высокие потребительские качества.

4.3.4. Концепция .NET и C#

С развитием глобальной сети Internet возникла необходимость писать программы, переносимость которых обеспечивается не только на уровне текстов программ, но и на более глубоком уровне. Это привело к созданию концепции Java, в которой переносимость достигается трансляцией текста программы в промежуточный язык, называемый байт-кодом, который затем интерпретируется виртуальной машиной Java (*Java Virtual Machine, JVM*). Обработанная по этой технологии программа может быть исполнена на любой платформе, имеющей виртуальную машину JVM. Технология Java позволяет иметь всего одну исполняемую версию программы, в то время, как использование обычных языков программирования требует создания исполняемых программ для каждого возможного варианта системного окружения.

Язык Java решил многие проблемы переносимости программ, однако, выбранная технология работы с этим языком поставила его в изолированное положение по отношению к другим языкам программирования. Компания Microsoft, разрабатывая новую технологию .NET (*dot NET*), стала поддерживать *многоязыковое программирование*, то есть такой способ взаимодействия программ, при котором программы, написанные на разных языках, могут работать совместно. В принципе можно было продолжить работу над совершенствованием технологии Java, но компания выбрала другой путь и предложила новый язык программирования Си#, как и язык Java основанный на языке Си++. Язык Си# строится на объектной модели языка Си++, а синтаксис и многие служебные слова во многом заимствованы из языка Си.

В последнее время компания Microsoft активно продвигает новое поколение систем программирования, объединяемых общим наименованием .NET Framework (управляемая среда для разработки и исполнения приложений). Эта среда состоит из общеязыковой исполняющей среды (CLR) и библиотеки классов. С самого начала она обеспечивала межъязыковую совместимость программ, написанных на трех языках программирования – Visual Basic .NET, Visual C# и Visual C++, а также сценариев, написанных на языке JScript. Технология .NET решает задачу создания единой универсальной платформы (базы) программирования, равно годящейся для разработки любых программ – обычных приложений, приложений для работы с базами данных, сетевых служб, приложений для мобильных и переносных устройств. Независимо от языка программирования в системе .NET Framework используется общая система типов (*common type system – CTS*), что обеспечивает совместимость типов между всеми языковыми компонентами. Как и в технологии Java, при обработке текстов программ трансляция с любого языка программирования сначала осуществляется в единый язык внутреннего представления (*Microsoft Intermediate Language – MSIL* или *IL*, впоследствии *Common Intermediate Language – CIL*). Внутреннее представление выбрано таким, чтобы низкоуровневый промежуточный язык охватывал все возможности исполняющей среды CLR. Любому элементарному типу данных во всех языках в среде CLR и в языке промежуточного представления соответствует некоторый базовый тип. Применение во всех языках общей системы типов, которые можно преобразовывать друг в друга, позволяет компонентам обмениваться данными, избегая потери времени на преобразование типов. Правильность решений, заложенных в язык промежуточного представления, доказывается тем, что трансляцию в этот язык, а значит и включение в общую исполняющую среду CLR, выполнили еще некоторые компании, создавшие трансляторы с языков Фортран и Кобол. Технологию .NET с языком промежуточного представления CIL поддерживает и система

программирования Delphi, начиная с версии 8. Существуют проекты переноса технологии в операционные системы UNIX и Linux. Распространение технологии .NET на другие платформы несколько затруднено из-за проблем, связанных с воспроизведением пользовательского интерфейса: экраны настольного компьютера, карманного компьютера и мобильного телефона сильно различаются.

Кроме общей системы типов CTS и общего языка внутреннего представления программ, технология .NET характеризуется следующими спецификациями, составляющими общеязыковую инфраструктуру (Common Language Infrastructure - CLI):

- *Extensible Metadata - расширяемые метаданные.* В технологии .NET подразумевается, что команды CIL помещаются в единицу распространения – сборку (assembly) – и сопровождаются метаданными, которые делают сборку полностью самодостаточным объектом. В метаданные помещаются имя и версия сборки, сведения о локализации, данные о типах, включенных в сборку, список внешних файлов (сборок), от которых зависит данная сборка и т. п.
- *Framework Class Library* – библиотека классов, которые должна использовать любая программа в рамках технологии. Библиотека VCL Delphi и C++ Builder в чем-то подобна .NET Framework. Разница между ними, прежде всего, состоит в том, что библиотеку .NET Framework можно использовать при создании программ на любом языке программирования, поддерживающем технологию .NET. Более того, эту библиотеку можно расширять новыми классами, которые затем могут использоваться в программах на других языках программирования.
- *Platform Invocation Service (P/Invoke)* - служба согласования платформ. Программы, исполняемые в .NET, предельно изолированы друг от друга и от средств операционной системы. Однако вне этих средств .NET Framework не может реально работать. P/Invoke реализует взаимодействие .NET Framework и операционной системы.
- *Extended Portable Executable (PE) File Format* - стандартный формат исполняемых файлов, используемый для хранения объектов технологии. Он загружается обычным загрузчиком точно так же, как и любой другой исполняемый файл. Однако в его заголовке имеется бит, указывающий на то, что файл относится к технологии .NET. Обнаружив бит, загрузчик обращается к исполняющей среде CLR, которая и производит обработку файла.

Включение в технологию .NET именно исполняющей среды CLR, а не виртуальной машины, работу которой надо интерпретировать специальными программами (аналогично JVM), существенным образом отличает эту технологию .NET от технологии Java. В отличие от технологии, применяемой при работе с языком Java, основная работа идет не с интерпретатором промежуточного языка, а с транслятором, преобразующим программу на промежуточном языке в машинные команды перед ее выполнением. Эта трансляция выполняется с помощью JIT-компилятора (*Just-in-time*), вызываемого автоматически в тот момент, когда управление передается в ту часть программы, которая представлена на промежуточном языке. Компилятор преобразует CIL по мере надобности и вставляет заглушки на место

вызова методов объектов. При первом обращении к заглушке, управление передается JIT-компилятору, который заменяет ее реальной программой. Такого рода обращения к компилятору производятся непосредственно из фрагментов программы, состоящих из обычных машинных команд, ранее сгенерированных компилятором, что повышает скорость исполнения программ. Таким же образом производится обработка программ, написанных на любых других языках программирования, входящих в состав системы программирования.

В ходе компиляции программы CIL она дополнительно подвергается верификации. Верификация проверяет саму программу и метаданные в поисках выходов из надежного окружения. Надежность типов объектов есть надежность их изоляции от других объектов и надежность их защиты от ошибочного или злонамеренного разрушения. Во время верификации программа проверяется на доступ к разрешенной памяти и вызов только правильно определенных методов. Например, не допускается обращение к полям, которые выходят за отведенные им границы. Дополнительно верификация проверяет правильность генерации машинных команд.

4.4. Системы программирования ОС UNIX и Linux

С самого начала разработки системы UNIX она рассматривалась в качестве переносимой (мобильной) операционной системы. Высокая мобильность часто ограничивала использование наиболее современных периферийных устройств. В сложившихся условиях система программирования ОС UNIX (по крайней мере, на первом этапе своего развития) стала ориентироваться на работу с командной строкой.

До сих пор этот режим работы в системе UNIX (и в системах, основанных на ней или функционально и организационно близких) остается актуальным. Он доказал свою эффективность, хотя в настоящее время для систем типа UNIX разработаны современные графические пользовательские интерфейсы.

Работа в режиме командной строки, в том числе для выполнения обычных действий, с которых начинается непосредственная подготовка программ в системах программирования – начального заведения и редактирования текстов, выполняется в системах UNIX с помощью специальных средств – интерпретаторов языка управления заданиями, которые в UNIX называются командными интерпретаторами. Командные интерпретаторы являются посредниками между пользователями и системными программами. Для систем UNIX были разработаны многие варианты командных интерпретаторов, среди которых наибольшее распространение получили 4 варианта: *C shell* (*csh*), *Bourne shell* (*sh*), *Korn shell* (*ksh*) и *Bourne again shell* (*bash*). Командные оболочки, за которыми работают командные интерпретаторы, позволяют не только выполнять отдельные команды операционной системы, но и формировать командные файлы, содержащие заранее сформированные последовательности команд.

Кроме командных интерпретаторов, система UNIX содержит весь комплекс программ, обеспечивающих жизненный цикл программ: редактор, компиляторы с библиотеками классов, процедур и функций, необходимых для компоновки программ, подготовленных системой программирования, стандартный ассемблер и компоновщик. В составе системы UNUX также имеются

- Редактор текстов ***vi*** (***vi****sual editor* – визуальный редактор), представляющий собой диалоговый редактор, позволяющий в интерактивном режиме вводить тексты или приказы на редактирование. Редактор ***vi*** позволяет также осуществлять поиск по тексту и контекстную замену.

- Развитый, самодокументированный, расширяемый экранный редактор реального времени *EMACS* (***E**ditor **MAC**ro**S***), настраиваемый на разные типы терминалов и потребности пользователей редактор. Расширяемость редактора основана на использовании встроенного в редактор интерпретатора языка Лисп (диалекта Common Lisp). Как и все редакторы, редактор EMACS позволяет проводить удаление и вставку текста, а также осуществлять контекстный поиск и контекстную замену в текстах. Существенное отличие редактора EMACS от других редакторов – в его мобильности. Этот редактор работает не только в UNIX-системах, но также в системах VMS, MS-DOS, системах семейства Microsoft Windows. Редактор EMACS может запускаться не только из командной строки. Он работает в многооконном режиме с интерфейсом X Window. Редактор EMACS имеет основные режимы ввода и редактирования текстов для языков программирования Лисп, Scheme (вариант Лиспа), Awk, Си, Си++, Фортран, Icon, Java, Objective-C, Паскаль, Perl (Practical Extraction Report Language), Pik ę CORBA IDL, и Tcl. Есть также основной режим для *Make*-файлов. Наличие этих режимов позволяет не только осуществлять форматный ввод текстов программ, наиболее удобный для конкретного языка программирования, но и запускать сами компиляторы.
- Пользовательский интерфейс X Window System, разработанный в Массачусетском технологическом институте в 1984 году, который облегчает работу в любой оконной системе UNIX. В основу этого интерфейса положены различия между функциями клиента и сервера в процессе рисования изображения на экране компьютера. Рисование на экране выполняет только сервер, а клиентская часть программы передает серверной части все необходимые для рисования данные. Такой подход позволяет не только решать проблемы безопасности, но и позволяет легко создавать приложения, работающие с несколькими экранами сразу. С точки зрения системы программирования интерфейс X Window System важен том, что на его основе созданы многочисленные системы рабочих столов (*desktop*), являющиеся основами интерактивных систем программирования. Среди наиболее распространенных рабочих столов – системы *CDE* (*Common Desktop Environment*) и *KDE* (*K Desktop Environment*).
- Компиляторы с различных языков программирования, от языка ассемблера до объектно-ориентированных языков Си++ и Java.
- Разделяемые (динамические) библиотеки.
- Компоновщик ***ld***, представляющий собой классический редактор связей, позволяющий объединять оттранслированные модули и компоненты библиотек в единую программу, готовую к исполнению. Обычно компоновщик в системе программирования ОС UNIX вызывается автоматически, заданием нужных параметров при обращении к компилятору, но может вызываться и явными командами операционной системы.
- Система управления сборкой и компиляцией программ (*Make*), позволяющая отслеживать изменения файлов, составляющих программный комплекс, задавать взаимозависимости файлов, оптимизировать процесс подготовки программных комплексов на основе взаимозависимостей и правил компиляции и компоновки.

- Система управления версиями исходных текстов (***S**ource **C**ode **C**ontrol **S**ystem, SCCS*), позволяющая отслеживать изменения, осуществляемые в текстах программ, работать с последними и более ранними версиями программ, восстанавливать предыдущие версии файлов и блокировать одновременное внесение исправлений в тексты со стороны нескольких пользователей. Система SCCS поставляется почти со всеми версиями UNIX и этим имеет преимущество перед другими системами управления версиями.
- Профилировщик ***prof***, позволяющий определять, как программа распределяет время при исполнении, например, определять, на вызов каких функций затрачивается особенно много времени. Результатом работы профилировщика является упорядоченная таблица, в которой собрана информация о совокупном времени работы каждой процедуры, числе вызовов этой процедуры и среднем времени работы процедуры в расчете на 1 вызов.
- Программа ***lint*** синтаксического контроля программ, написанных на языке программирования Си.
- Символический отладчик ***dbx***, позволяющий выполнять и отлаживать программы в пошаговом режиме, выполнять редактирование текстов программ непосредственно во время их отладки, получать доступ к значениям переменных и выполнять трассировку программ. Информация о ходе выполнения программы и состоянии переменных выдается в терминах исходного языка программирования, что значительно упрощает процесс отладки программ. Отладчик ***dbx*** – это стандартный отладчик ОС UNIX. Он поставляется с большинством версий этой операционной системы, но в настоящее время он уступает по своим качествам отладчикам других распространенных систем программирования (например, отладчикам компаний Borland и Microsoft), а также другим отладчикам системы UNIX (например, ObjectWorks\C++), поставляемым на коммерческой основе.
- Программа ***Lex*** – генератор лексических анализаторов.
- Программа построения синтаксических анализаторов ***Yacc***.

В системе UNIX имеется огромное количество полезных системных программ, а многочисленные версии этой системы могут содержать свои собственные варианты программ и дополнительные программы, прямо связанные с особенностями конкретных реализаций. Преимуществом всех этих систем и вариантов является то, что в любой системе всегда имеется в наличии некоторый стандартный набор системных программ, обеспечивающий единообразие технологических приемов во всех системах.

4.5. Проект GNU

Автором общего проекта мобильной (переносимой) операционной системы GNU (рекурсивный акроним – *GNU's Not Unix*) и входящего в нее многоязыкового компилятора GCC является Ричард Столмен, сотрудник Лаборатории искусственного интеллекта Массачусетского Технологического Института, инициировавший работу в 1983 году. Сегодня количество пользователей системы GNU, распространяемой с открытыми исходными текстами программ и электронной документацией, оценивается в десять миллионов человек.

Кроме технических целей Ричард Столмен, недовольный неограниченной ничем коммерциализацией системных программ и самого процесса системного

программирования, ставил перед собой цель создать бесплатно распространяемую систему программирования, которая могла бы оказаться серьезным конкурентом дорогостоящим коммерческим системам.

Система GNU способна исполнять программы UNIX, но не совпадает с ней полностью. В эту систему авторами были внесены усовершенствования, которые они считали удобными, основываясь на собственном опыте работы с другими операционными системами. В частности, были введены длинные имена файлов, поддержка версий файла, отказоустойчивая файловая система, поддержка дисплея, независимая от терминала.

Ядро системы имеет наименование GNU Hurd (коллекция серверов или “стадо гну”, “herd of gnus”). Это ядро основано на использовании многопотоковых серверов, обменивающихся сообщениями и работающих поверх микроядра Mach, разработанного в университете Карнеги-Меллон (Carnegie Mellon University) и позднее в университете штата Юта (University of Utah). Работа над ядром GNU Hurd еще продолжается, однако, сейчас доступно другое ядро. После появления в 1991 г. системы Linux его ядро в течение 1992 г. было объединено с незавершенной системой GNU в полноценную операционную систему. Эта версия называется GNU/Linux, поскольку она скомбинирована из двух этих систем.

Разработчики проекта GNU постоянно стремились добиться реализации в своей системе всех тех возможностей, которые предоставляются системой UNIX, одновременно реализуя некоторые дополнительные возможности и добиваясь максимально возможной переносимости своих программ. Система программирования GNU практически полностью повторяет систему программирования UNIX. Она содержит весь комплекс программ, обеспечивающих жизненный цикл программ, имеющихся в системе UNIX: редактор, командный интерпретатор *bash*. В системе имеются многоязыковый компилятор GCC с библиотекой классов, процедур и функций, необходимой для правильного функционирования компилятора и программ, обработанных системой программирования, компоновщик и ассемблер GAS (по скорости превышающий возможности стандартного ассемблера UNIX почти вдвое). В составе системы GNU также имеются

- развитый, самодокументированный, расширяемый экранный редактор реального времени EMACS, настраиваемый на разные типы терминалов и потребности пользователей редактор,
- программа ***Flex*** – аналог стандартного генератора лексических анализаторов ***Lex***, позволяющий получать более эффективные по сравнению с ***Lex*** анализаторы,
- программа построения синтаксических анализаторов ***Yacc*** и ее свободно распространяемый аналог ***Bison***,
- отладчик ***GDB*** и программа пакетной подготовки программ ***MAKE***,
- системы, поддерживающие работу с версиями программ в больших программных проектах, *RCS* (*Revision Control System*) и *CVS* (*Concurrent Version System*),
- программа криптографического кодирования GNU *Privacy Guard,*
- почти полностью совместимый с языком Postscript графический язык Ghostscript,
- расширенный вариант стандартной архивной утилиты ***tar***,
- программу сжатия файлов ***gzip***,

- более быстрые по сравнению со стандартными варианты утилит ***grep*** и ***diff***,
- интерактивная программа для рисования математических выражений и данных ***gnuplot***,
- электронные таблицы,
- игровые программы (например, для игры в шахматы) и многое другое, включая документацию.

В настоящее время продолжается работа над созданием модели сетевой объектной среды GNOME (*GNU Network Object Model Environment*). В других системах для аналогичного окружения используется термин "*desktop*".

Многие программы, работающие в системах UNIX, технически вполне готовы к работе в системе GNU, но не используются там или используются с некоторыми ограничениями из-за проблем, связанных с лицензированием. К таким программам относятся система пользовательского оконного интерфейса X Window, библиотека классов *Motif*, библиотека компонентов пользовательского интерфейса *Qt* (в системе GNU ее аналогом являются библиотека *KDE* и библиотека Harmony, предназначенная для работы с программами *KDE*, не используя лицензированную библиотеку *Qt*). Некоторые известные своим широким применением в системах UNIX лицензированные программы имеют в системе GNU свои свободно распространяемые аналоги. Например, библиотека *Motif* имеет аналог в виде библиотеки *LessTif*.

Можно наблюдать и обратное влияние на систему UNIX со стороны проекта GNU. В первую очередь это влияние проявляется в том, что практически все версии UNIX включили в свой состав переносимый многоязыковый компилятор GCC. В начальный период, когда в GNU включался только язык программирования Си, аббревиатура ***GCC*** расшифровывалась, как ***G***NU ***C C***ompiler. В настоящее время это же сокращение трактуется, как ***G***NU ***C***ompiler ***C***ollection. Проект этого компилятора возник на основе использования языка ***RTL*** (***R****egister* ***T****ransfer* ***L****anguage*) в качестве языка внутреннего представления программ. Идея, лежавшая в основе разработке языка RTL, заключалась в интерпретации пар последовательно сгенерированных транслятором машинных команд и их замены (где это оказывается возможным) на эквивалентные одиночные команды. Машина при этом описывалась в терминах передачи элементов данных между регистрами и памятью.

Основной задачей, которую ставили перед собой авторы компилятора GCC, была задача создания хорошего, быстрого компилятора для 32-х разрядных вычислительных машин, адресующих оперативную память с точностью до отдельного байта и имеющих несколько регистров общего назначения. Многоязыковость и многоплатформенность компилятора GCC позволили на практике свести проблему построения $m \times n$ компиляторов с m языков для n вычислительных машин к значительно менее сложной проблеме построения $m+n$ компиляторов. Этот компилятор известен также тем, что его подключение к системе программирования к очередной вычислительной машине существенно проще, чем в случае других, тоже многоязыковых, а иногда и многоплатформенных компиляторов. Достигается это примененным методом описания той вычислительной машины, для которой должна осуществляться трансляция.

Компилятор GCC получает основную информацию об объектной вычислительной машине из ее описаний, выполненных в виде алгебраических формул каждой из машинных команд. В тех случаях, когда такую формулу записать слишком

сложно, в свободно распространяемые тексты программ компилятора можно добавить новый параметр, описывающий дополнительный режим трансляции.

Транслятор GCC обладает достаточной гибкостью, чтобы формировать различные последовательности команд при трансляции одних и тех же программ, работающих на одних и тех же ЭВМ, в одном операционном окружении, но с разными прикладными системами, например, с разными пакетами прикладных программ. Использование компилятора GCC не требует от пользователей полного отказа от работы с другими компиляторами, а также с библиотеками, созданными другими компиляторами. Этот компилятор имеет четкий, хорошо описанный интерфейс и достаточное количество средств управления режимами работы. Например, при генерации вызовов процедур с параметрами, транслятор GCC может формировать различные последовательности команд записи значений фактических параметров в стек (в прямом или в обратном порядке), что позволяет точно согласовывать порядок и способы передачи параметров процедурам и функциям, а также методы обратной передачи значения функций после завершения их выполнения. Этим достигается возможность комплектования программ, оттранслированных с помощью GCC, с программами, полученными при использовании других компиляторов и ассемблеров, а значит построения гибких, мощных и удобных систем программирования.

4.6. Системы программирования компании IBM

4.6.1. Комплексная система программирования Rational Software

Примером комплексной системы программирования служит система Rational Software Corporation, принадлежащая в настоящее время компании IBM. Это наиболее полная система, в основе которой современная методология проектирования, поддержанная совокупностью технических средств. В состав этой системы входят

- Совокупность приемов и решений RUP (*Rational Unified Process*), распространяемая не только в виде пособий и книг, но и в виде гипертекста с помощью страниц Интернета;
- Программные средства Rational Suite, состоящие из
  - средств управления требованиями (Rational Requisite Pro);
  - средств визуального проектирования (Rational Rose), основанные на формальном языке моделирования UML (Unified Modeling Language), ставшем всеобщим стандартом описания сложных систем (например, систем рабочих потоков – workflow).
- Генератор программ на Си++ и Java, работающий на основе выбранного каркаса приложения (Rational Apex);
- Средства автоматизации тестирования Rational Team Test (RTT), Rational Robot (часть RTT, используемая для автоматизации прогонов тестов), Rational Test Factory (для автоматизации тестирования интерактивных задач) и Rational Pure Coverage (для контроля полноты покрытия тестами).
- Средства создания документации (SoDA);
- Профилировщик
- Единая база проекта

В этой системе программирования (и проектирования) детально проработана итеративная модель жизненного цикла программного продукта. Архитектура строящейся системы описывается в терминах языка UML в виде набора графических

моделей. Работа по реализации системы определяется целями проектируемого процесса, которые формулируются в виде сценариев взаимодействия строящейся программной системы с другими системами или пользователями. Во время каждой итерации система автоматически контролирует приближение реализации к описанным вариантам использования.

При разработке программного продукта выделяются 4 основные фазы, которые могут повторяться циклически необходимое число раз. Внутри каждой из фаз также возможно циклическое повторение отдельных работ. К этим фазам относятся:

- Задание отправной точки проекта (Inception), которая определяется при формулировании целей, назначении руководящих органов и исполнителей проекта, утверждении бюджета и выборе основных методов и инструментов реализации проекта. По оценкам эта фаза занимает около 10% времени и требует около 5% бюджета всего проекта.
- Фаза проектирования (Elaboration), которая состоит в разработке структуры программного продукта, решающего поставленные задачи. Фаза занимает около 30% времени и требует около 20% бюджета.
- Построение системы (Construction), то есть окончательное уточнение требований и собственно разработка системы в рамках ранее определенной структуры. Получающаяся система должна удовлетворять всем выделенным вариантам ее использования. Разработка системы занимает около 50% времени и 65% бюджета.
- Внедрение (Transition), заключающееся в том, что система делается доступной пользователям. На этой же фазе проводится тестирование системы. Фаза занимает около 10% времени и требует около 10% бюджета.

В ходе работы над проектом могут создаваться различные таблицы информации, базы данных, текстовые документы, тексты программ, объектные модули, а также различные модели поведения системы, описываемые в виде диаграмм UML.

4.6.2. Интегрированная среда разработки Eclipse

Интегрированная среда Eclipse является примером системы, построенной на базе обычного диалогового редактора, и превратившейся в интегрированную самодостаточную среду разработки программного обеспечения, служит приобретающая все больше пользователей среда разработки Eclipse. Эта система, сопровождаемая компанией IBM, работает с различными операционными системами (Windows, UNIX/Linux) и различными языками программирования (Java, Си, Си++, Кобол, UML). Свойства расширяемости и настраиваемости позволяют ее пользователям самим создавать подобия систем программирования, подключая отдельно распространяемые модули или создавая такие модули своими силами.

Среда разработки Eclipse (то есть основное окно Eclipse в целом) называется рабочим столом. Внутри основного окна расположены различные панели, называемые видами, которые отображают консольный вывод, текущий проект в виде дерева объектов и т. д. Многочисленные виды объединяются в группы и могут быть выбраны при помощи закладок. Кроме видов рабочий стол содержит одну специальную панель – редактор, в котором можно редактировать документы различных типов, например, исходный текст программы. При выполнении различных задач, таких как программирование, отладка или синхронизация изменений текстов программ с

централизованным репозиторием, можно устанавливать расположение окон, удобное для конкретной задачи. Это (зависящее от задачи) расположение окон называется перспективой.

Перспектива может иметь много видов, но только один редактор, являющийся центральным объектом рабочего стола. Если в редакторе открыт файл, остальные виды отображают различные аспекты этого файла. Если это файл с текстом программы, один вид покажет физическое месторасположения в файле и иерархии пакета, а другой вид покажет методы класса и атрибуты. При открытии нескольких файлов эти виды будут меняться при переходе на различные файлы в редакторе.

В среде Eclipse достаточно просто открывать новые перспективы и переключаться между ними, но в процессе работы в этом обычно нет необходимости, поскольку перспективы меняются автоматически в нужное время. Перспективы являются простым и естественным способом работы, поскольку удаляют из интерфейса все инструментальные кнопки, меню и виды, которые не относятся к текущей задаче. Это значительно облегчает поиск необходимых инструментов.

Среда Eclipse обладает всеми современными возможностями по проверке синтаксической правильности вводимых текстов программ. При вводе ведется немедленный синтаксический анализ и выдаются подсказки по продолжению ввода (например, при вводе имени класса выдается список методов класса, из которого программист может выбрать нужный).

Важной возможностью, имеющейся в среде Eclipse, является ее способность осуществлять рефакторинг текстов программ, то есть изменение структуры текста без изменения его функциональности:

- переименование полей, переменных, классов, интерфейсов,
- изменение логической организации программ на уровне классов, например, перемещение методов или полей из класса во вложенный или объемлющий класс,
- изменение состава классов, например, преобразование локальных переменных в поля класса, выделение части метода и организация нового метода на ее основе, генерация установочных методов для полей.

Интегрированная среда Eclipse предоставляет также возможность управлять версиями создаваемых программ. Для этого она может взаимодействовать с системой CVS, используя две отдельные перспективы. Первая перспектива служит для выбора репозиториев и модулей для подсоединения и анализа содержащихся в них файлов. Вторая перспектива нужна для объединения изменений в репозитории. Такой подход поощряет дисциплину и структурированность рабочего процесса.

Очень важным фактором развития системы Eclipse является ее свободное распространение с полным набором исходных текстов. Это привлекает к ее развитию большое сообщество разработчиков, проектирующих для этой системы подключаемые модули, как свободно распространяемые, так и коммерческие.

4.6.3. Системы программирования ЭВМ zSeries

Компания IBM выпускает программное обеспечение, в том числе и системы программирования, не только для персональных ЭВМ, но и для больших ЭВМ, предназначенных для одновременной работы многих самых разнообразных пользователей. В настоящее время аппаратная платформа IBM носит наименование

zSeries, она является развитием выпускающейся на протяжении нескольких десятилетий линии ЭВМ S/360, S/370 и S/390. На платформе zSeries могут работать различные операционные системы, наиболее распространенными среди которых являются z/VM, z/OS и Linux (z/OS UNIX). Наиболее характерной для архитектуры zSeries является система программирования, работающая под управлением z/OS, так как система программирования Linux в точности соответствует всем остальным системам, построенным на базе операционных систем типа UNIX, а операционная система виртуальных машин z/VM обычно работает в качестве промежуточной системной платформы, над которой в свою очередь надстраиваются другие операционные системы, в частности, и Linux, и z/OS (возможно даже одновременно).

В состав системы программирования z/OS входят все необходимые средства, с помощью которых можно создавать, модифицировать, хранить и распространять новое прикладное программное обеспечение на различных языках программирования, включая язык ассемблера и языки высокого уровня, такие как Си, Си++, Кобол, PL/1, Фортран, Ада, Java и др. (всего – около 20 языков программирования). Помимо базовых средств разработки, включающих текстовый редактор, набор компиляторов, редакторы связей и средства загрузки программ, в z/OS реализована универсальная языковая среда Language Environment, содержатся многочисленные библиотеки программ и классов, представлен менеджер сопровождения разработки программного обеспечения (ISPF/SCLM). Кроме того, выпускается целый ряд продуктов, таких как IBM Visual Age, IBM Application Development Tool, которые поставляются вне z/OS и служат для автоматизации и повышения эффективности процесса разработки приложений.

Поскольку от программных продуктов, создаваемых и работающих под управлением системы z/OS, требуется совместимость с продуктами, разработанными для ранее выпускавшегося аппаратного и системного программного обеспечения, обладавших ограниченными возможностями (например, ранее использовался только 24-разрядная система аппаратной адресации и короткие имена внешних объектов), в состав новой системы программирования включаются сразу по несколько компонентов, выполняющих сходные функции, но работающие в разных режимах совместимости.

Так, например, в систему входят сразу два редактора связей: стандартный Linkage Editor и усовершенствованный Program Management Binder. Стандартный редактор связей служит для построения объектных модулей “старого” формата, поддерживающих только 24- и 31-разрядные режимы адресации с ограничением общего объема кода в 16 мегабайт. Усовершенствованный компоновщик Binder обеспечивает возможность связывания объектных модулей в модули нового формата (64-разрядный режим), называемые программными объектами. Объем кода при этом может достигать одного гигабайта.

Базовый компонент z/OS, называемый языковым окружением LE (Language Environment), поддерживает единую универсальную среду выполнения для приложений, созданных на языках программирования высокого уровня Си, Си++, Кобол, PL/1 и Фортран. Библиотеки языковой среды включают наиболее существенные и часто используемые компоненты времени выполнения, такие как формирование сообщений, обработка событий, управление памятью, поддержка функций даты и времени и т. п. Эти компоненты доступны всем приложениям, независимо от используемого языка программирования. Кроме того, языковое окружение упрощает взаимодействие между приложениями, написанными на разных языках или для разных операционных сред, за счет специальных интерфейсных средств. Средства языковой

среды LE через соответствующие макровызовы доступны и тем приложениям, которые написаны на языке ассемблера. Языковая среда LE состоит из следующих элементов:

- базовые программы, обеспечивающие универсальную обработку сообщений, запуск и завершение программ, динамическое распределение памяти, обработку событий (в том числе ошибок) времени выполнения, взаимодействие между программами, написанными на разных языках;
- общие библиотеки, содержащие набор модулей для поддержки математических функций и функций даты и времени, реализуемых на основе стандартного интерфейса вызовов функций LE;
- специфические библиотеки, содержащие модули, применяемые только для одного из поддерживаемых языков высокого уровня.

Создаваемые с использованием универсальных модулей языковой среды приложения могут выполняться в различных операционных средах, включая как внутрисистемные (например, UNIX shell), так и среды промежуточного слоя (DB2, CICS). Все компоненты, входящие в состав системных библиотек LE, делятся на две группы: резидентные (статически загружаемые) и динамические. Резидентные программы при редактировании связей включаются в объектный модуль приложения. К ним относятся, например, программы запуска и завершения приложения. Динамические программы не включаются в объектный модуль, они добавляются к программе уже при ее выполнении при непосредственном их вызове.

Огромное внимание в описываемой системе программирования уделяется подготовке программ для использования в пакетном режиме – традиционном способе разработки программ, применяемом программистами для ЭВМ, выпускавшимися компанией IBM, а также многими ее последователями в течение десятилетий. Ключевым элементом данного способа является использование стандартных процедур языка управления заданиями (*Job Control Language, JCL*), хранящихся в системной библиотеке и предназначенных для компиляции, редактирования связей и исполнения различных программ, написанных на том или ином языке высокого уровня. По-существу, язык JCL (точнее первые его версии, существенные развитые к настоящему времени) еще 1960-х годах стал основой для проектирования множества различных командных языков множества операционных систем, в том числе систем, используемых на персональных ЭВМ.

Примером может служить, “каталогизированная” процедура ASMACLG (представляющая собой, по-существу, полный аналог командных файлов других систем программирования), предназначенная для компиляции, редактирования связей и выполнения ассемблерной программы, представленной в виде исходного модуля. На первом шаге (*C*) вызывается ассемблер ASMA90, на втором шаге (*L*) – редактор связей HEWL, а на третьем (*G*) – запускается созданный объектный модуль. По меркам z/OS эта процедура совсем проста, она не содержит символических параметров, хотя использование таких параметров есть обычная практика при программировании на языке JCL:

```
//ASMACLG   PROC
//C         EXEC  PGM=ASMA90,PARM=(OBJECT,NODECK)
//SYSLIB    DD    DSN=SYS1.MACLIB,DISP=SHR
//SYSUT1    DD    DSN=&&SYSUT1,SPACE=(4096,(120,120),,,ROUND),
//                UNIT=VIO,DCB=BUFNO=1
//SYSPRINT  DD    SYSOUT=*
```

```
//SYSPUNCH  DD    SYSOUT=B
//SYSLIN    DD    DSN=&&OBJ,SPACE=(3040,(40,40),,,ROUND),
//                UNIT=VIO,DISP=(MOD,PASS),
//                DCB=(BLKSIZE=3040,LRECL=80,RECFM=FBS,BUFNO=1)
//L         EXEC  PGM=HEWL,PARM='MAP,LET,LIST,NCAL',COND=(8,LT,C)
//SYSLIN    DD    DSN=&&OBJ,DISP=(OLD,DELETE)
//          DD    DDNAME=SYSIN
//SYSMOD    DD    DISP=(,PASS),UNIT=SYSDA,SPACE=(CYL,(1,1,1)),
//                DSN=&&GOSET(GO)
//SYSUT1    DD    DSN=&&SYSUT1,SPACE=(1024,(120,120),,,ROUND),
//                UNIT=VIO,DCB=BUFNO=1
//SYSPRINT  DD    SYSOUT=*
//G         EXEC  PGM=*.L.SYSMOD,COND=((8,LT,C),(8,LT,L))
//          PEND
```

Имея такую процедуру, легко организовать ассемблирование любой пользовательской программы. Полная обработка программы, написанной на языке ассемблера, включающая также и выполнение этой программы, будет заключаться в задании только одной команды, в которой закодирован вызов заранее подготовленной каталогизированной процедуры:

```
EXEC ASMACLG,PARM.C=LIST,PARM.L=NOMAP
```

Во избежание большого ручного труда при составлении подобных каталогизированных процедур (командных файлов) в системе программирования z/OS предусмотрен специальный диалоговый режим работы с компонентами этой системы, прежде всего, с компиляторами, редакторами связей и отладчиками.

5. Разработка распределенных программ

Особую важность приобретают системы программирования, когда возникает необходимость использования их для создания распределенных программ. Распределенная система – это набор независимых компьютеров, представляющихся их пользователям единой объединенной системой. Тем самым, во-первых, **от пользователей скрыты различия** между компьютерами и способы связи между ними (компьютеры распределенной системы автономны), во-вторых, пользователи и **приложения единообразно работают** в распределенных системах, независимо от того, где и когда происходит их взаимодействие. Наиболее важными свойствами таких систем являются прозрачность, открытость, масштабируемость и безопасность.

Прозрачностью программной системы называется целый комплекс ее свойств, благодаря которым обеспечивается сокрытие (упрятывание) особенностей реализации системы, например, разницы в способах представления данных и доступа к ресурсам, местоположения составных частей и параллельного использования системы несколькими пользователями одновременно. *Открытость системы* есть использование синтаксических и семантических правил, основанных на стандартах. Для распределенных систем – это, прежде всего, использование формализованных протоколов взаимодействия в информационных сетях, в том числе, в глобальных сетях, корпоративных или общедоступных, например, в сети Интернет. Важными свойствами, связанными с открытостью программных систем являются их *переносимость* (способность приложений работать в составе разных распределенных систем) и *способность к взаимодействию*.

Технология разработки распределенного программного обеспечения тоже может обладать прозрачностью настолько, насколько она позволяет разработчику забыть о том, что создаваемая система распределена, и насколько легко в ходе разработки можно отделить аспекты построения системы, связанные с ее распределенностью, от решения задач прикладной области, в рамках которых системе предстоит работать.

Распределенные системы обычно строятся с использованием служб, предоставляемых другими системами, и в то же время сами часто являются составными элементами или поставщиками служб для других систем, поэтому при разработке распределенных часто систем используется компонентный подход.

Достаточный уровень масштабируемости больших систем, которые на практике могут поддерживать одновременную работу нескольких тысяч пользователей, может быть достигнут только с помощью децентрализации основных служб системы и управляющих ею алгоритмов, которая все шире внедряется в практику построения распределенных систем, в частности, современных сетевых служб.

5.1.Системы клиент-сервер

В распределенных системах важнейшими являются два понятия, для обозначения которых используются термины “***клиент***” и “***сервер***”. *Клиентом называется программная система (программный компонент), посылающий запрос серверу на выполнение какой-либо услуги, сервером называется программная система (программный компонент), выполняющий задание, полученное по запросу от клиента.* Первые распределенные системы состояли из клиентской и серверной частей, которые взаимодействовали друг с другом, посылая запросы и получая ответы на них. В настоящее время программные компоненты могут выступать то в роли клиента, то в роли сервера.

Взаимодействие между клиентскими и серверными частями распределенной программной системы может быть *синхронным* и *асинхронным*. *Синхронным* называется такое взаимодействие, при котором клиент, отослав запрос, блокируется и может продолжать работу только после получения ответа от сервера. По этой причине такой вид взаимодействия называют иногда *блокирующим*. В рамках *асинхронного* или *неблокирующего* взаимодействия клиент после отправки запроса серверу может продолжать работу, даже если ответ на запрос еще не пришел. Примером такого асинхронного взаимодействия является электронная почта.

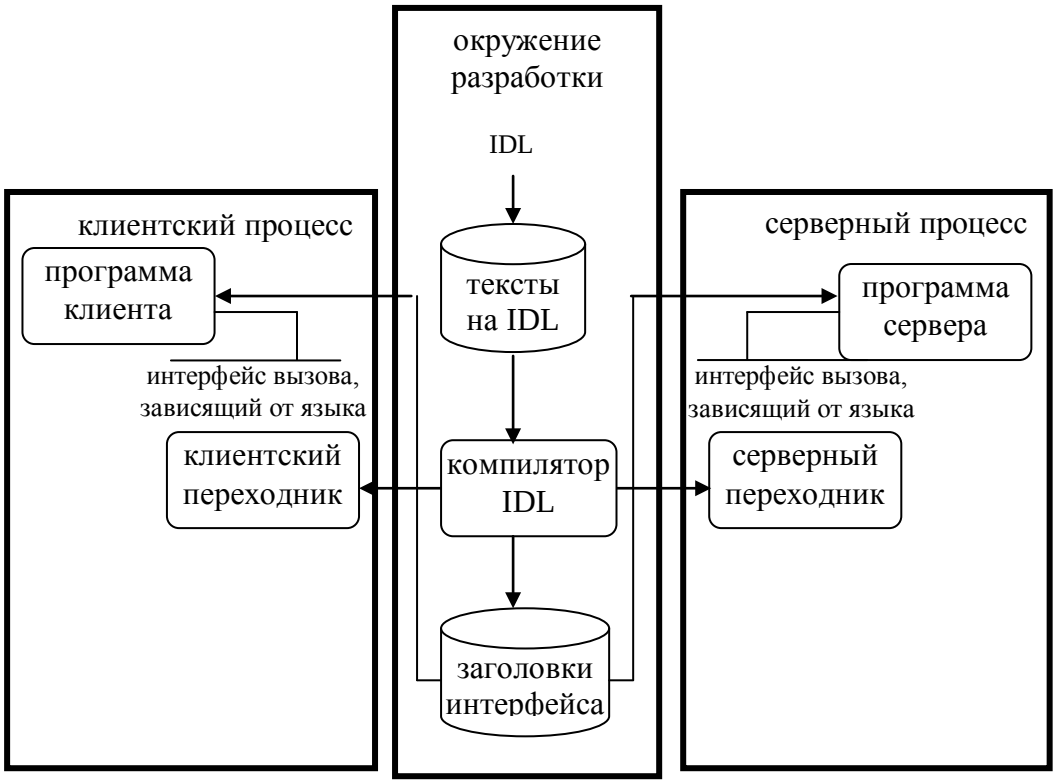
Наиболее распространенным и универсальным способом реализации синхронного взаимодействия в распределенных системах является ***удаленный вызов процедуры***. Основное преимущество модели удаленного вызова процедуры состоит в том, что сам клиент не знает, что вызов был удаленный. Однако и сервер не знает о том, что обрабатывал удаленный вызов.

Удаленный вызов процедур определяется как способ организации взаимодействия между компонентами, так и методику разработки этих компонентов. На первом шаге разработки определяется интерфейс процедур, которые будут использоваться для удаленного вызова. Это делается при помощи *языка определения интерфейсов* (*Interface Definition Language, IDL*), в качестве которого может выступать специализированный язык или обычный язык программирования, с ограничениями, определяющимися возможностью передачи вызовов на удаленную машину. Определение процедуры для удаленных вызовов компилируется компилятором IDL в описание этой процедуры на языках программирования, на которых будут разрабатываться клиент и сервер (например, заголовочные файлы на языке Си или Си++), и два дополнительных компонента — ***клиентский*** и ***серверный переходники***. *Клиентский переходник* представляет собой компонент, размещаемый на той же машине, где находится компонент-клиент. Удаленный вызов процедуры клиентом реализуется как обычный, локальный вызов определенной функции в клиентском переходнике. При обработке этого вызова клиентский переходник выполняет следующие действия:

1. Определяется физическое местонахождение в системе сервера, для которого предназначен данный вызов. Это шаг называется *привязкой* к серверу. Его результатом является адрес машины, на которую нужно передать вызов.
2. Вызов процедуры и ее аргументы упаковываются в сообщение в некотором стандартном для данной конкретной системы формате, понятном серверному переходнику. Этот шаг называется *маршалингом*.
3. Полученное сообщение преобразуется в поток байтов (*сериализация* сообщения) и отсылается с помощью какого-либо протокола (транспортного или более высокого уровня) на машину, на которой помещен серверный компонент.
4. После получения от сервера ответа, он распаковывается из сетевого сообщения и возвращается клиенту в качестве результата работы процедуры.

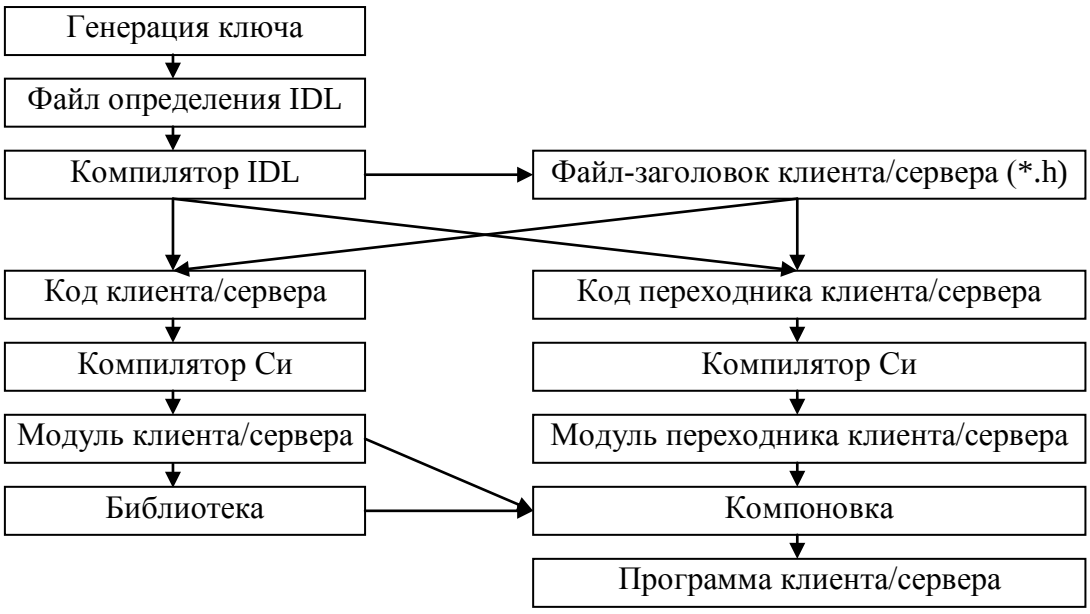
В результате для клиента удаленный вызов процедуры выглядит как обращение к обычной функции. *Серверный переходник* располагается на той же машине, где находится компонент-сервер. Он выполняет операции, обратные к действиям клиентскому переходнику — принимает сообщение, содержащее фактические параметры вызова, распаковывает эти параметры при помощи *десериализации* и

демаршалинга, вызывает локально соответствующую функцию серверного компонента, получает ее результат, упаковывает его и посылает по сети на клиентскую машину.



Эта же техника может быть использована и для реализации взаимодействия компонентов, работающих в рамках различных процессов на одной машине.

Системами типа "*клиент-сервер*" называются простейшие распределенные программные системы, построенные только в двух уровнях – уровне клиента и уровне сервера. Примером системы, позволявшей создавать прикладные системы типа клиент/сервер на вызовах удаленных процедур, является система DCE (*Distributed Computing Environment*), предложенная группой Open Software Foundation (Open Group). Процесс подготовки программ на языке Си к работе выглядит в системе DCE так:



В состав DCE входит много компонентов языки, библиотеки, службы. Созданные клиенты и серверы могут оказаться самыми разными. В основе их всех – язык IDL, в котором функции описываются образом, похожим на прототипы функций в языке Си, а в файлах имеются определения типов и констант для маршалинга параметров и демаршалинга результатов.

Первым шагом в подготовке программ клиента и сервера всегда является запуск системной программы, которая создает прототип файла на языке IDL, содержащий уникальный ключ (идентификатор интерфейса), гарантированно не содержавшийся ни в одном интерфейсе, созданном ранее, определения типов, констант, типов параметров и результатов функций. Этот прототип редактируется, в него вносятся имена удаленных процедур и их параметров, затем выполняется компиляция с IDL. Эта компиляция формирует три файла:

- файл-заголовок (например, "`interface.h`")
- файл клиентского переходника
- файл серверного переходника

В файл-заголовок из исходного описания на IDL передаются уникальный идентификатор интерфейса, определения типов, констант и описания функций (прототипы). Клиентский переходник содержит те процедуры, которые будет непосредственно вызывать клиентская программа. Эти процедуры непосредственно отвечают за подбор параметров и их упаковку в исходящие сообщения с последующими обращениями к системе для их отправки. Переходник также занимается распаковкой ответов, приходящих от сервера, и передачей значений, содержащихся в этих ответах, клиенту. Серверный переходник организован также, но по отношению к серверу.

Следующий шаг – написание реальных программ клиента и сервера. После их компиляции и компоновки (с библиотечными добавками) формируются обе исполняемые программы – клиентская и серверная.

В целом, DCE – это попытка *стандартизовать* понятие удаленного вызова процедуры. Впоследствии, когда широко распространились принципы и приемы объектно-ориентированного программирования, система DCE также была расширена и дополнена объектно-ориентированными языками.

5.2. Технологии COM/DCOM

Технология COM (*Component Object Model*) и ее вариант для распределенных систем DCOM (*Distributed COM*) была разработана компанией Microsoft. DCOM является расширением технологии COM и включает в себя среду распределенных вычислений DCE и механизм удаленного вызова процедур.

Общий подход при использовании той или технологии таков: клиентская программа использует объекты своего программного сервера так, как если бы эти объекты являлись частью клиентской программы. Основную роль, как и во всех системах клиент/сервер, играет интерфейс объектов, формируемый при помощи некоторого языка IDL, в данном случае – объектно-ориентированного. Клиент знает об используемых объектах только их интерфейсы. Сервер предназначен для реализации объектов, он представляет собой программу, содержащую, кроме всего прочего, еще один или несколько объектов, построенных в соответствии с моделью COM.

Реализации объектов создаются на основе нескольких классов, каждый из которых представляет различные варианты поведения объекта.

Переход к объектно-ориентированному взаимодействию привел к существенному пересмотру модели удаленного вызова процедуры и появлению модели ***удаленного обращения к методу***. Эта модель основана на объектно-ориентированном расширении понятия вызова процедуры и понятии о распределенном объекте. Методы вспомогательных объектов, включаемых в состав клиентского и серверного переходников, которые строятся для реализации удаленного обращения к методу, имеют интерфейсы в точности соответствующие интерфейсам реальных удаленных объектов. Их реализации позволяют скрытым от пользователей образом организовать маршалинг и сериализацию параметров и возвращаемых значений методов. Как и при использовании "классического" удаленного вызова процедуры, при удаленном обращении к методу не только клиент не знает о том, что взаимодействует с удаленным сервером, но и сервер не знает, что обращение к нему осуществлено со стороны.

Система автоматически строит взаимодействия клиента и сервера, независимо от того, как клиент и сервер распределены по компьютерам: они могут исполняться

- на одном компьютере в рамках единого процесса, при этом взаимодействие между клиентом и сервером происходит при помощи интерфейса объекта в едином адресном пространстве с использованием динамических библиотек.
- на одной машине в рамках разных процессов, при этом между клиентом и сервером возникают два промежуточных звена, а схема взаимодействия становится похожей на сокращенную схему удаленного обращения к методу (с переходниками, маршалингом и демаршалингом, но без обращения к сети).
- на различных (вообще говоря, несовместимых друг с другом аппаратно и имеющих разные операционные системы) компьютерах в рамках информационной сети (технология DCOM), использующих модель удаленного обращения к методу полностью, включая сериализацию и обработку сообщений.

Система COM не является системой программирования в классическом значении этого термина. Это, скорее, система библиотек компонентов и правил их использования. Однако система COM содержит все элементы, необходимые для построения распределенной системы, в частности, компонентную модель, библиотеки классов, которые могут быть импортированы для анализа структуры серверов COM, универсальный протокол обмена между клиентами и серверами и другие.

Технологию COM могут поддерживать самые различные языки программирования. В настоящий момент наиболее широко используются Visual Basic, Си++ и Delphi. Однако разработчик системы (компания Microsoft) объявила эту технологию устаревшей и активно продвигает другой подход – технологию .NET. Основной недостаток технологии COM, который и привел к отказу от нее, это серьезные ограничения в организации взаимодействия между разными платформами, которые постоянно возникают в глобальных сетях. Определенными недостатками и ограничениями обладает и выбранный для систем COM язык Microsoft IDL, в котором недостаточно развиты средства объявления типов данных, из которых строится программа. В целом, технологию COM/DCOM обычно используют для построения небольших распределенных систем, имеющих не очень большое число узлов.

5.3. Брокеры объектов CORBA

Более адекватно соответствующим принципам построения распределенных систем, чем системы "клиент-сервер", надо признать системы не двухуровневые, а имеющие, по крайней мере, еще один "промежуточный" уровень, позволяющий разделить решаемые задачи на "клиентские" и "серверные" части. В двухуровневых системах клиентские части чаще всего связаны с отображением данных в виде, адекватно соответствующем конкретной прикладной области, назначение серверных частей – выполнять основные прикладные программы и программы системной поддержки. Наличие промежуточного уровня (для обозначения которого часто используется англоязычный термин *middleware*) существенно расширяет возможности распределенных систем. Системы "клиент-сервер" в качестве одной из самых своих серьезных проблем имеют ограниченность возможностей сервера по связи со многими клиентами одновременно. Двухъярусные архитектуры не справились с требованиями, предложенными разработчиками локальных информационных сетей. Их появление заставило разработчиков предпринять меры по интеграции серверов, а для размещения программ, ответственных за эту интеграцию, наилучшим образом подходил промежуточный уровень трехуровневых систем. Однако простой интеграцией преимущества новых систем не ограничиваются. Трехуровневые системы обладают свойством масштабируемости в значительно большей степени. Если системы "клиент-сервер" привели к стандартизации интерфейсов прикладного слоя, то их развитие позволило стандартизовать интерфейсы слоя управления ресурсами, а это привело к возможности интеграции в рамках одной системы самых разнородных информационных ресурсов. Возникли даже попытки стандартизовать глобальные свойства и интерфейсы между разными системными платформами на основе объектно-ориентированного подхода. Одним из примеров такой стандартизации является стандарт брокера объектов CORBA (*Common Object Request Broker Architecture*). Брокером объектов называют распределенные системы программного обеспечения, основанные на использовании понятия распределенного объекта и модели удаленного обращения к методам этого объекта, являющейся объектно-ориентированным расширением модели удаленного вызова процедуры.

Стандарт CORBA – это спецификация для создания и управления объектно-ориентированными приложениями, распределенными в вычислительной сети. В настоящее время выработано несколько версий стандарта CORBA, первая из которых была разработана в начале 90-х годов консорциумом Open Management Group, включавшим более 800 компаний. Все эти стандарты описывают только интерфейсы и не содержат никакой реализации.

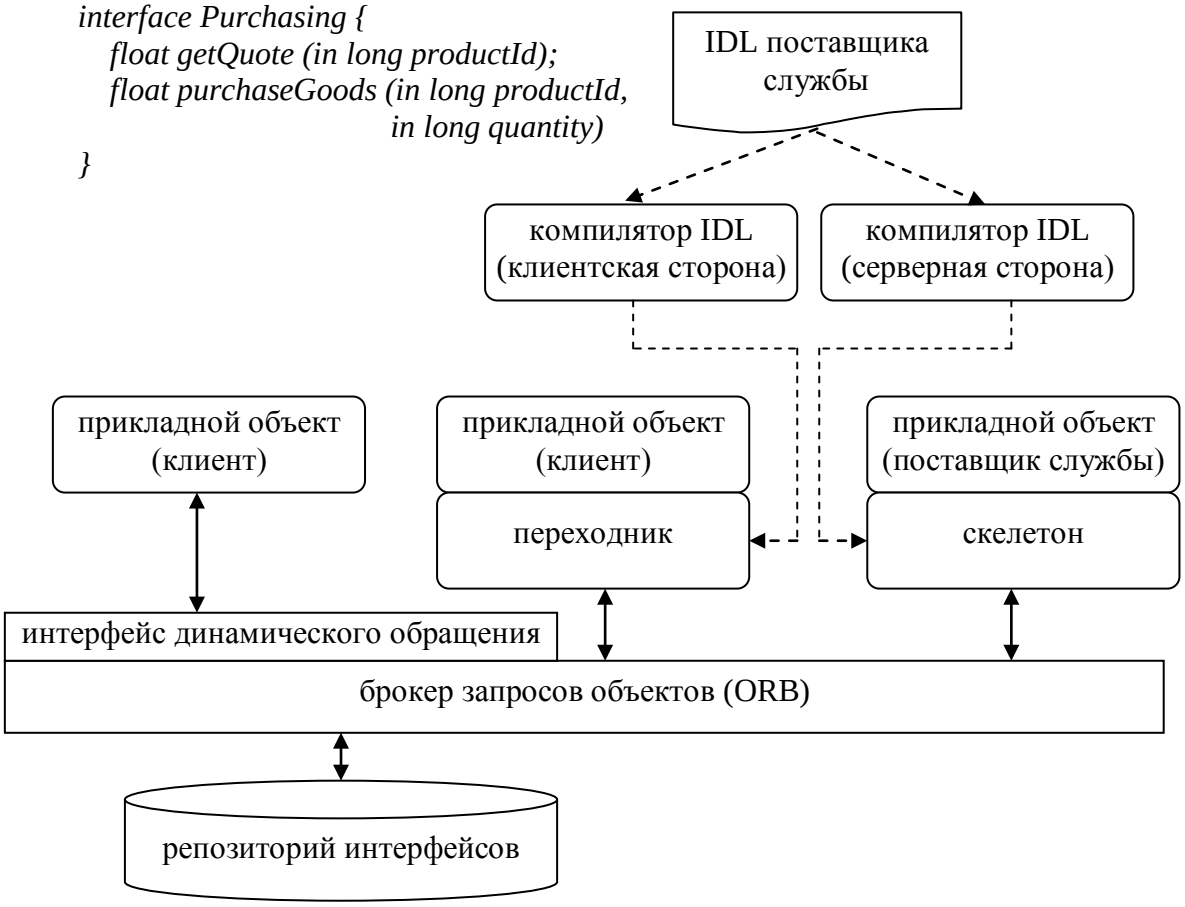
Всякая система, подчиняющаяся спецификации CORBA, состоит из трех основных частей:

- *брокер запросов объектов*, содержащий базовые функции взаимодействия объектов.
- *службы CORBA*. Набор служб, известный под этим собирательным именем, доступен с помощью стандартизованного прикладного программного интерфейса. Службы предоставляют функциональность, обычно необходимую большинству объектов, например, сохранность, управление жизненным циклом и безопасность.

- *средства CORBA*. Набор средств и инструментов, известный под этим собирательным именем, предоставляет службы верхнего уровня, необходимые приложениям, а не индивидуальным объектам. Сюда входят управление документами, интернационализация, поддержка мобильных агентов. Средства CORBA могут также включать службы, специфичные для отраслевых рынков, например, образования, здравоохранения или транспортных перевозок.

Чтобы к объекту можно было обратиться через брокер объектов, этот объект должен сначала объявить свой интерфейс, из чего клиенты узнают о методах, которые он предоставляет. Интерфейсы описываются на языке описания интерфейсов IDL (стандарт CORBA предлагает свой собственный вариант такого языка). Спецификации IDL транслируются в скелетоны на стороне сервера и в переходники на стороне клиента.

```
interface Purchasing {
   float getQuote (in long productId);
   float purchaseGoods (in long productId,
                        in long quantity)
}
```



В дополнение к описанию методов, в отличие от систем на базе удаленного вызова процедур, язык IDL спецификации CORBA поддерживает множество объектно-ориентированных концепций, например, наследование и полиморфизм. Спецификации, написанные на IDL CORBA, могут быть переданы компилятору с этого языка, который формирует заместителя объекта и скелетон. Заместитель объекта – это переходник, ответственный за сокрытие распределенности. Программа заместителя содержит в себе описание методов, предоставляемых реализацией объекта. Для получения готового клиентского приложения, она должна быть загружена вместе с программой клиента. С другой стороны скелетон защищает от проблем распределенности сервер, поэтому

сервер может разрабатываться так, как если бы вызовы к нему поступали из локального окружения. Как заместитель, так и скелетон могут быть написаны на любом из тех языков, которые поддерживаются компилятором с языка IDL CORBA, на которые текст IDL может быть оттранслирован. Например, спецификация CORBA 3 поддерживает трансляцию с IDL на Си, Си++, Java, Smalltalk, Аду, Кобол, Лисп, PL/1, Python и IDLScript.

Из-за различных способов употребления и различных способов реализации ссылок на объекты в языке Си++, эти ссылки при трансляции отображаются в два типа Си++. Константы IDL отображаются непосредственно в определения констант Си++. Базовые типы данных IDL имеют естественное отображение в типы данных Си++. Структурированные типы данных ***class***, ***struct***, ***union***, *sequence* отображаются в структуры или классы Си++. Массивы отображаются в определения массивов Си++. Операции отображаются в виртуальные методы Си++, имеющие то же имя, что и операции.

Современные версии спецификации CORBA допускают и обратные отображения: например, в стандарте CORBA 3 прямо предусмотрено проведение отображения записи интерфейсов на языке Java в записи интерфейсов на IDL. Обратное отображение позволяет программистам на языке Java создавать объекты, доступные из других приложений, написанных (возможно) на других языках программирования. Имеются в виду объекты, имеющие объектную ссылку CORBA и доступ к межброкерному протоколу IIOP (*Internet Inter-ORB Protocol*). Конечно, такая возможность подразумевает введение некоторых ограничений на использование языка Java (в частности, нельзя использовать параметры ***inout*** и ***out***, все выходы в IDL присутствуют в возвращаемом значении). Обработка программы на языке Java обратным компилятором позволяет получить эквивалентный интерфейс, написанный на IDL, имея который, можно построить (на языке Java или другом языке программирования) программу клиента CORBA, имеющую доступ к нужному объекту.

Чтобы иметь техническую возможность взаимодействовать с некоторым сервером, все, что требуется знать программисту, это IDL-интерфейс этого сервера. Конечно, разработчик должен быть осведомлен о семантике интерфейсов методов и других ограничениях (например, о специфическом порядке, в котором следует вызывать методы для достижения той или иной цели). Эти аспекты не формализованы, их предполагается описывать другими средствами, например, в виде комментариев, вставляемых в IDL-спецификации, или в составе сопроводительной документации.

Описанный механизм спецификации CORBA, призванный обеспечивать способность к взаимодействию, требует, чтобы клиент был статически привязан к интерфейсу: компилятор IDL статически генерирует переходник, специфический для конкретного сервисного интерфейса. Однако модель удаленного обращения к методу допускает динамическое обнаружение новых объектов и построение обращений к ним в процессе работы, даже если для данного клиента не был создан никакой переходник. Эта возможность базируется на двух компонентах: *репозитории интерфейсов* и *интерфейсе динамического обращения*. *Репозиторий интерфейсов* хранит определения всех объектов, известных брокеру ORB. Приложения могут использовать репозиторий для поиска, редактирования или удаления IDL-интерфейсов. Один брокер может иметь несколько репозиториев, и несколько брокеров могут иметь доступ к одному репозиторию. Единственное требование, поставленное в спецификации CORBA, заключается в том, что каждый брокер должен иметь хотя бы один репозиторий.

Возможность динамического построения обращений к методам на основе динамически обнаруженных интерфейсов решает только часть проблемы динамического обращения к службе. Он предполагает, что клиенты уже идентифицировали нужную им службу. Для того чтобы это стало возможным, в спецификации CORBA ссылки на сервисные объекты выдаются только *службой именования* и *справочной службой*. Отношения между этими двумя службами напоминают отношения между алфавитными и тематическими каталогами или телефонными справочниками. Служба именования позволяет извлекать ссылки на объекты, отталкиваясь от их имени, а справочная служба дает возможность клиентам искать службы, основываясь на их свойствах. Службы в свою очередь вносят сведения о своих свойствах в справочник, при этом разные службы имеют разные свойства, описывающие их нефункциональные характеристики. Используя справочную службу, клиенты могут искать не только объекты, реализующие тот или иной интерфейс, но также объекты, свойства которых имеют заданные значения (например, книги по Си).

Спецификация CORBA позволяет пользователям систем, построенных на ее основе, организовывать свои программы в виде служб, предоставляющих услуги другим программам, то есть таким же службам или более традиционно построенным программам пользователей. Однако обычно (по аналогии с библиотеками стандартных программ) вместе с базовой системой (самим брокером CORBA) могут распространяться программы служб, спецификации которых также стандартизованы. Некоторые из этих служб являются обязательными и распространяются всегда, другие службы, несмотря на стандартность интерфейса, имеют более ограниченное применение и распространяются по отдельным соглашениям с пользователями.

5.4. Серверы приложений и сетевые службы

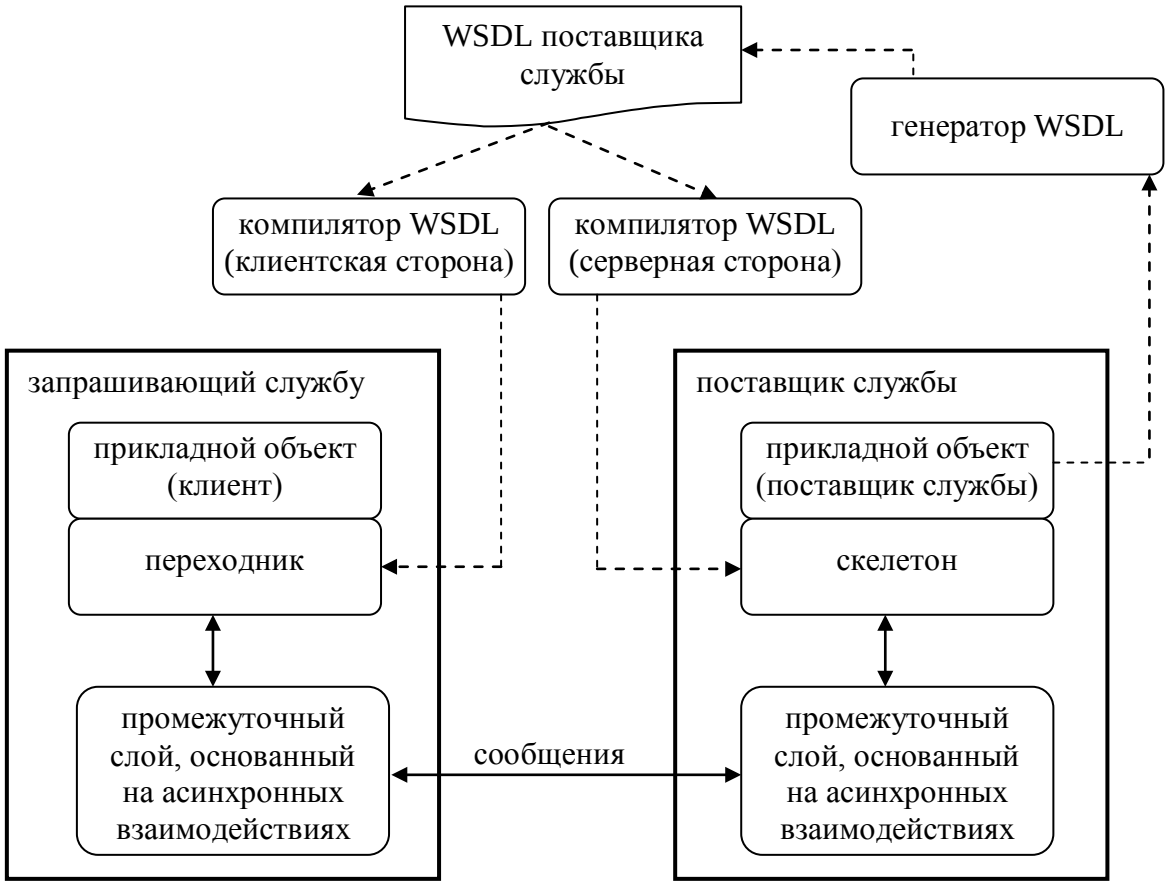
Как и описанные ранее технологии построения распределенных систем COM и DCOM, стандарт CORBA не ставит целью зафиксировать какое-либо представление о том, какими должны быть системы программирования для распределенных систем. В то же время эти стандарты направлены на решение задач, являющихся одновременно и задачами систем программирования – обеспечение поддержки программных продуктов на протяжении их жизненного цикла. Поддержка, которую подобные стандарты и системы оказывают прикладным программам, очень важна, причем по мере развития представлений о распределенных системах она становится все более необходимой. Еще более такая поддержка необходима в тех случаях, когда строящаяся распределенная система предназначается для интеграции программных компонентов, взаимодействующих друг с другом посредством глобальных сетей и, прежде всего, глобальных сетей общего доступа, например, посредством сети Интернет.

В последнее время системная поддержка распределенных программ, взаимодействующих в Интернете, приняла форму, которая называется сервером приложений, а само взаимодействие в этой сети стало осуществляться посредством использования сетевых служб. Наиболее широко распространены серверы приложений J2EE фирмы Sun Microsystems, .NET фирмы Microsoft, WebSphere компании IBM, WebLogic фирмы BEA Systems, OAS фирмы Oracle Corporation и многие другие, функционально близкие друг другу.

Все ранее описанные виды системных платформ базируются чаще всего на синхронных методах обращений, когда клиентское приложение обращается к методу, предлагаемому (возможно, динамически определяемым) поставщиком службы. Только когда поставщик службы заканчивает выполнение своей работы, он выдает ответ

клиенту. Однако в последнее время все большее распространение приобретают подходы, поддерживающие более динамичные асинхронные формы взаимодействия, а также системы распределенного программного обеспечения, взаимодействующие на основе обмена сообщениями. При асинхронном взаимодействии клиент, запрашивающий услугу у сервера, после выдачи запроса продолжает свою работу в той мере, в которой это возможно, не тратя время на ожидание результата. При этом в силе остается использование принципов объектно-ориентированного программирования, на которых основываются и современные серверы приложений и сетевые службы.

Серверы приложений представляют собой крупные библиотеки компонентов, содержащих средства поддержки, как на этапе программирования (проектирования интерфейсов), так и на этапе выполнения. Сетевые службы используются серверами приложений в качестве окон во внешний мир, именно через них наиболее удобно осуществлять взаимодействие в глобальной сети. В то же время сетевые службы сами по себе пригодны для использования и в более локальных системах, с их помощью могут даже взаимодействовать отдельные независимые компоненты серверов приложений. Для описания услуг, предоставляемых сетевыми службами, как и в других распределенных системах, используется специальный язык описания интерфейсов WSDL (*Web Service Definition Language*), компилятор с которого включается в состав системы программирования.



Современные системы программирования для сетевых служб содержат одновременно и специальные средства генерации описания интерфейсов по исходным текстам на более традиционных языках программирования, например, по текстам на языке Java. В остальном схема формирования сетевой службы напоминает аналогичные

схемы формирования серверной и клиентской частей в системах, основанных на моделях удаленного вызова процедур и удаленного обращения к методам.

По-существу, при взаимодействии сетевых служб и происходит обращение одной службы (выступающей в данном случае в роли клиента) к удаленной процедуре, реализованной внутри другой службы, являющейся в этот момент сервером. Однако сетевые службы выглядят гораздо более симметричными, чем клиенты и серверы в более традиционных распределенных системах. Одни и те же службы могут попеременно выступать в обеих ролях – быть и клиентами и серверами. Более того, возможно и такое взаимодействие сетевых служб, когда они одновременно являются клиентами одних служб и серверами запросов других служб.

Сетевые службы имеют и другое важнейшее отличие от традиционных средств взаимодействия: для них удалось стандартизовать не только интерфейсы, как для процедур и методов классов, но и протоколы взаимодействия. В обычных системах последовательность вызова процедур или обращений к методам классов формально никак не регламентирована. Только из неформальных описаний семантики процедур и методов, а также из примеров, обычно включаемых разработчиками в документацию, при программировании приложений удается добиться правильной последовательности вызовов. Для сетевых служб протоколы их функционирования в информационной сети описываются на специально для этого разработанном языке описания протоколов. Таких языков в настоящее время существует несколько, наиболее перспективным в настоящее время можно считать язык выполнения бизнес-процессов BPEL.

Однако полезность серверов приложений не ограничивается только их способностью проводить взаимодействие в глобальной сети посредством хорошо стандартизованных сетевых служб. Например, в сервере приложений J2EE для поддержки взаимодействия и презентации предназначены сервлеты, а также язык тегов JSP и его интерпретатор, прикладной интерфейс для работы с языком XML (JAXP – прикладной программный интерфейс для синтаксического анализа текстов на языке XML), система электронной почты, служба аутентификации и авторизации. Поддержка интеграции приложений обеспечивается специальными компонентами EJB, интерфейсом именования и каталогов JNDI, службой сообщений и транзакционным интерфейсом. Поддержка доступа к ресурсам осуществляется компонентами обеспечения связи с базами данных JDBC и подключения архитектур J2CA.

Серверы приложений позволяют работать с самыми разнообразными клиентскими программами, что важно при работе именно в разнородной глобальной сети. Такими клиентскими программами могут быть:

- *сетевые навигаторы*, включая те, которые работают с простыми страницами HTML, и те, которые умеют загружать и выполнять аплеты.
- *приложения*, такие же, как и в традиционных системах (возможность подключать к сетевым службам другие прикладные системы позволяет интегрировать множество разнородных приложений в единые слаженно работающие комплексы, при этом в сами приложения никаких изменений вносить не требуется).
- *устройства*, например, мобильные телефоны.
- *программы электронной почты*.
- *клиенты сетевых служб*, то есть приложения, взаимодействующие с сервером приложений через стандартные протоколы сетевых служб.

6. Средства автоматического грамматического разбора

6.1. Построение лексических анализаторов по регулярным выражениям

Разработка теории формальных грамматик привела к разработке практических систем, осуществляющих автоматический разбор текстов, записанных на формальных языках, по формализованным правилам.

Наиболее успешными работы по автоматизации грамматического разбора были в области лексического анализа, так для описания лексики языков программирования оказалось достаточным использовать наиболее теоретически простые языки – регулярные. Всякий регулярный язык может быть одним из трех способов:

- с помощью регулярной (праволинейной или леволинейной) грамматики,
- с помощью конечного автомата,
- с помощью регулярного множества (так же, как и с помощью обозначающих их регулярных выражений).

Это не значит, что регулярные языки можно выражать только этими тремя способами, но именно эти три способа их задания полностью эквивалентны друг другу. Существуют алгоритмы, которые позволяют любой регулярный язык, заданный каким-либо одним из этих трех способов, описать любым другим способом. Это значит, что *на основе любой регулярной грамматики можно построить регулярное выражение, определяющее тот же язык*.

Регулярные множества для алфавита V *определяются рекурсивно*:

1. Пустое множество $\varnothing$ есть регулярное множество.
2. Множество из одного пустого элемента $\{\varepsilon\}$ есть регулярное множество.
3. Множество из одного элемента алфавита $\{a, \ \forall\ a \in V\}$ есть регулярное множество.
4. Если множества P и Q – произвольные регулярные множества, то их объединение $P \cup Q$, их конкатенация PQ, итерация P^* и усеченная итерация P^+ ($P^+ = PP^*$) есть регулярные множества.
5. Других регулярных множеств не существует.

Регулярные множества обозначаются с помощью регулярных выражений, которые рекурсивно вводятся следующим образом:

1. Пустое множество $\varnothing$ обозначается как 0.
2. Множество из одного пустого элемента $\{\varepsilon\}$ обозначается как ε.
3. Множество из одного элемента алфавита $\{a, \ \forall\ a \in V\}$ обозначается как a.
4. Если p и q – регулярные выражения, обозначающие регулярные множества P и Q, то $p|q$, pq, p^* и p^+ есть регулярные выражения, обозначающие регулярные множества $P \cup Q$, PQ, P^* и P^+.
5. Если p – регулярное выражение, обозначающее регулярное множество P, то (p) есть регулярное выражение, обозначающее это же регулярное множество P.

При записи регулярных выражений можно использовать скобки. При их отсутствии операции выполняются слева направо с учетом приоритетов, наивысший из которых имеет левоассоциативная операция итерации, средний – операция конкатенации, низший – операция объединения множеств. Усеченная итерация имеет те же приоритет и ассоциативность, что и итерация.

Регулярные выражения часто используются для записи грамматических правил лексического разбора языков программирования. Например, идентификаторы некоторого языка программирования, представляющие собой букву, за которой следует нуль или несколько букв или цифр, можно определить с помощью регулярного выражения таким образом:

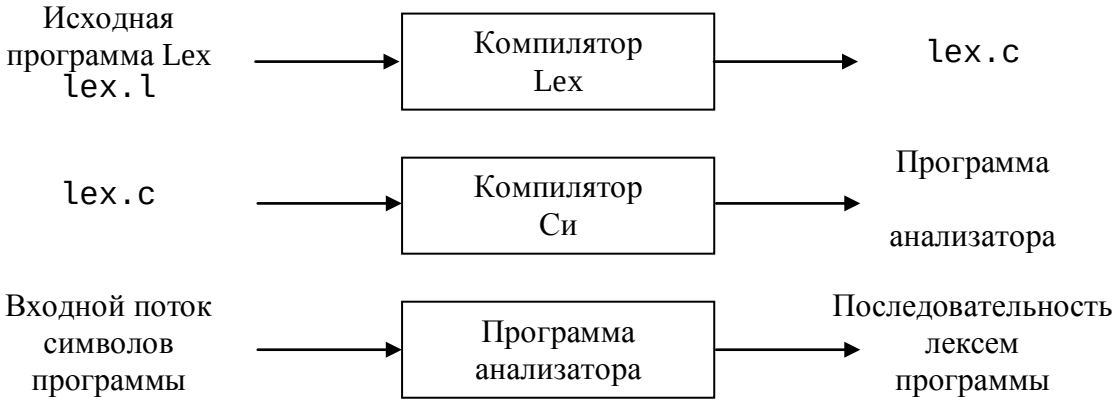
```
letter(letter|digit)*
```

Правила построения лексических единиц такого языка можно записать так:

```
letter      →    a | b | ... | z | A | B | ... | Z
digit       →    0 | 1 | 2 | ... | 9
Id          →    letter (letter | digit)*
Num         →    digit+
```

Одним из лучших инструментов для автоматического построения лексических анализаторов является программа ***Lex***, осуществляющая построение программы анализатора на основе обработки спецификаций, использующих регулярные выражения. Обработка ведется на основе концепции конечного автомата, диаграмма состояний которого описывается регулярными выражениями. Первоначально эта программа была создана как компонент операционной системы UNIX, однако сейчас имеется множество ее вариантов, работающих в различных операционных системах, в том числе, в системах MS-DOS и Microsoft Windows.

Схема использования программы ***Lex*** представляет собой трехшаговый алгоритм:



На первом шаге подготавливается спецификация лексического анализатора, то есть на языке ***Lex*** записываются регулярные выражения, описывающие лексемы анализируемого языка (файл `lex.l`). Эта программа обрабатывается компилятором ***Lex***, в результате чего получается текст на языке программирования Си (в настоящее время существуют версии программы ***Lex***, создающие выходные тексты на других языках, например, Си++, Паскаль, Java). Эта программа содержит табличное представление диаграммы переходов, построенной по регулярным выражениям из файла `lex.l`. В нее также включается стандартная программа, использующая созданную таблицу переходов для распознавания лексем. Действия, которые связаны с регулярными выражениями в файле `lex.l`, представляют собой фрагменты программы на языке Си, копируемые из файла `lex.l` в файл `lex.c`. Эти действия

обычно выполняются всяким лексическим анализатором в его работе, они могут состоять, например, в создании записей в информационной таблице компилятора.

На втором шаге программа `lex.c` компилируется с помощью компилятора Си, в результате чего создается окончательная программа анализатора исходного языка. На третьем шаге эта программа в своей работе осуществляет ввод и лексический анализ текстов исходного языка.

Программа ***Lex*** основана на анализе и преобразовании регулярных выражений, с помощью которых записываются лексические правила. В программе ***Lex*** введены несколько расширенные правила записи регулярных выражений, позволяющие оптимизировать их. Для удобства записи регулярных выражений часто вводятся классы символов. Запись *[abc]*, где *a*, *b* и *c* – символы алфавита, означает регулярное выражение *a|b|c*. Сокращенный класс символов типа *[a-z]* означает регулярное выражение *a|b|...|z*. С использованием классов символов можно описать идентификаторы как строки, заданные регулярным выражением *[A-Za-z][A-Za-z0-9]*. Символ **‘-’** используется для обозначения диапазона в классе символов, символ **‘.’** означает, что в классе символов на этом месте может стоять любой символ, кроме символа новой строки. Чтобы в класс символов поместить непосредственно сам символ **‘.’** (или **‘-’**), перед ними следует ставить символ обратной косой черты, например, класс символов `[\.\-]` состоит из двух символов **‘.’** и **‘-’**.

В записи регулярных выражений программы ***Lex*** имеются также следующие способы указания повторений некоторых последовательностей символов:

`P*` - итерация (повторение нуль или более раз)
`P+` - усеченная итерация (повторение один или более раз)
`P?` – необязательное вхождение (нуль или один раз)

Двойные кавычки используются для цитирования, то есть непосредственного указания последовательности символов:

`“a.+*”` обозначает строку из четырех символов (без самих кавычек)

Приведем примеры записи регулярных выражений, правильных с точки зрения программы ***Lex***:

```
if                                      /* идентификатор if    */
[a-z][a-z0-9]*                          /* идентификатор       */
[0-9]*                                  /* число               */
([0-9]+“.”[0-9]*)|([0-9]*“.”[0-9]+)     /* вещественное число  */
(“—-”[a-z]*“\n”)|(“ ”|“\n”|“\t”)+       /* вид комментария     */
.                                       /* произвольный символ */
```

Программа `lex.l` состоит из трех разделов: объявлений, правил трансляции и вспомогательных процедур. В файле `lex.l` эти разделы следуют друг за другом именно в таком порядке, отделяясь друг от друга строками, состоящими из парных символов процента ‘%%’.

Раздел объявлений состоит из описаний переменных, именованных констант (идентификаторов, используемых для представления констант) и регулярных определений, которые используются в качестве компонентов регулярных выражений в правилах трансляции:

```
%{  /* Определение именованных констант,
       обозначающих коды лексем, например,
       ID, NUMBER, DELIMITER                  */
%}

/*     Регулярные определения                 */

delim   [ \t\n\b\v\r]
spaces  {delim}+
letter  [A-Za-z]
digit   [0-9]
id      {letter}({letter}|{digit})*
number  {digit}+

%%
```

Обычно описания переменных и констант записываются внутри скобок *'%{'* и *'%}'*. Все, что находится внутри таких скобок, непосредственно переписывается в создаваемую программу лексического анализатора, не рассматривается как часть регулярных определений или правил трансляции. Так же обрабатываются вспомогательные процедуры, входящие в третий раздел программы `lex.l`.

Правила трансляции программы `lex.l` имеют вид

p_i {действие$_i$}

где каждое p_i есть регулярное выражение, а каждое *действие$_i$* есть фрагмент программы, описывающий выполняемые лексическим анализатором действия, если лексема соответствует регулярному выражению (шаблону) p_i. Действия записываются на том же языке программирования, на котором должен быть сгенерирован анализатор (например, на том же языке Си):

```
{spaces}    { /* Действия не определены, возврата нет */ }
{id}        { yylval = found_id (); return ID; }
{number}    { yylval = found_num (); return NUMBER; }
"<"         { yylval = LT; return RELATION; }
"<="        { yylval = LE; return RELATION; }
.           { /* Такое правило обычно ставится последним.
                 Оно позволяет фиксировать ошибку,
                 связанную с появлением символа, который
                 не может начинать никакую лексему    */ }
%%
int found_id (void) {
/* На первый символ идентификатора указывает переменная yytext.
   Длина идентификатора содержится в переменной yyleng.
   Процедура заносит лексему в таблицу анализатора          */
}
int found_num (void) {
/* На первый символ числа указывает переменная yytext.
   Длина числа содержится в переменной yyleng.
   Процедура заносит лексему в таблицу анализатора          */
}
```

Третий раздел программы содержит различные вспомогательные процедуры, которые могут быть скомпилированы отдельно, но загружены должны быть вместе с лексическим анализатором.

Лексический анализатор, создаваемый программой ***Lex***, работает с синтаксическим анализатором по схеме однопроходного компилятора. Главной

программой является синтаксический анализатор, который запрашивает лексемы у лексического анализатора, начиная чтение оставшейся части исходного текста посимвольно и продолжая его до тех пор, пока не будет найдена *самая длинная последовательность*, соответствующая одному из регулярных выражений p_i (*это первое правило разрешения неоднозначностей при обработке регулярных выражений*). Затем для найденного шаблона выполняется соответствующее $действие_i$. Чаще всего это действие возвращает управление синтаксическому анализатору (то есть заканчивается оператором вида `return Token`). Но так бывает не всегда. В этих случаях лексический анализатор продолжает поиск лексем в тексте, пока действие, соответствующее одной из них, не возвратит управление синтаксическому анализатору. Такой поиск лексем позволяет осуществить обработку последовательностей пробелов и комментариев.

Лексический анализатор возвращает синтаксическому только код обнаруженной лексемы. В программе **Lex** предусмотрено, что дополнительные атрибуты лексемы могут содержаться в глобальной переменной `yylval`, текстовое представление лексемы находится в переменной `yytext`, а длина лексемы в символах – в переменной `yyleng`.

Если какой-либо лексеме соответствуют сразу несколько шаблонов, то выбирается тот из них, который *размещается в перечне шаблонов первым*, поэтому частные шаблоны следует ставить в этом перечне раньше общих (*это второе правило разрешения неоднозначностей при обработке регулярных выражений – порядок следования регулярных выражений имеет определяющее значение*). Например, шаблон `{if}`, который может быть записан для распознавания лексемы начала условного оператора, должен предшествовать шаблону `{id}`, которому соответствует всякий идентификатор, в том числе, идентификатор `if`.

Строка `"<="` будет соответствовать шаблону "<=", несмотря на то, что шаблон "<" находится в перечне раньше. В данном случае длина соответствия шаблону "<" меньше, чем длина соответствия шаблону "<=". Программа **Lex** позволяет проводить и еще более сложный анализ лексем. В частности, в ней имеются возможности использования *прогностических операторов*, позволяющих распознавать лексемы, в зависимости от той последовательности символов, которые располагаются за ними.

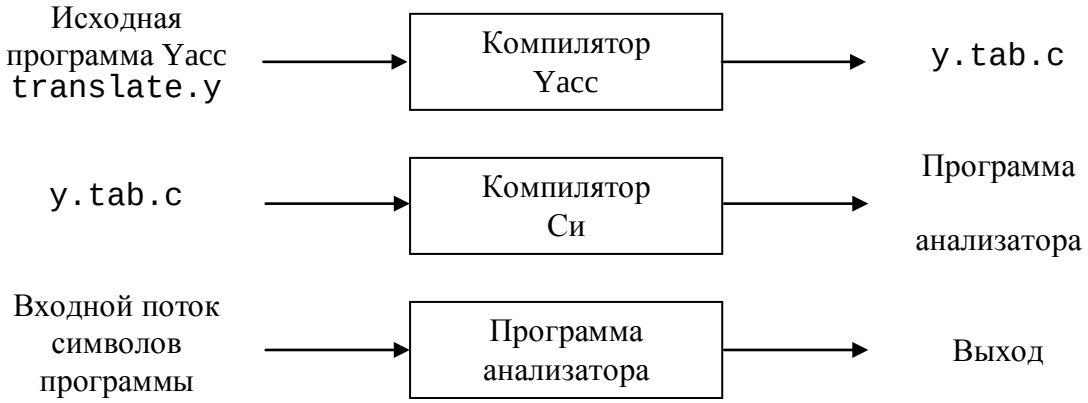
Свободно распространяемая версия программы **Lex** носит название **Flex** (*fast lexical analyzer generator*). Эта программа генерирует лексические анализаторы, по скорости работы не уступающие анализаторам, запрограммированным вручную, поскольку в ней представление детерминированного конечного автомата транслируется непосредственно в программу (на основе операторов перебора – `case`), и не возникает дополнительных затрат времени на интерпретацию таблицы переходов.

6.2. Автоматизация построения синтаксических анализаторов

Практически все современные языки программирования, позволяющие создавать практически ценные программные комплексы, являются контекстно-зависимыми. Однако расширение возможностей грамматического описания языков введением описаний действий, то есть замена обычных грамматик на грамматики с действиями, позволяет снизить требования к используемым грамматикам, а значит, к синтаксическим анализаторам и их генераторам. Грамматики с действиями нашли себе применение в программе автоматической генерации синтаксических анализаторов – **Yacc** (***Y****et* ***a****nother* ***c****ompiler-****c****ompiler* – еще один компилятор компиляторов).

По сравнению с генератором лексических анализаторов, работающим с регулярными выражениями, работа генератора синтаксических анализаторов осложнена возможными неоднозначностями в грамматиках. Генерация синтаксического анализатора для произвольной контекстно-свободной грамматики оказывается слишком сложным процессом. Не удается создать такой генератор даже для *LR(1)*-грамматик. Только специальный вид *LR(1)*-грамматик, называемый грамматиками *LALR(1)*, то есть "*Look Ahead*" *LR(1)*-грамматиками, позволяет разработать метод разрешения конфликтов на основании правого контекста длиной *1*. Класс *LALR(1)*-грамматик достаточно широк. Любой контекстно-свободный язык может быть задан такой грамматикой. Единственным их недостатком по сравнению с *LR(1)*-грамматиками является то, что *LALR(1)*-грамматики ограничивают возможности распознавателей по обнаружению ошибок во входных цепочках символов, точнее заставляют для обнаружения ошибок делать больше шагов вывода. Этот недостаток не может умалить важного достоинства *LALR(1)*-грамматик – возможности автоматического построения *практически осмысленных* генераторов синтаксических анализаторов, к которым принадлежит и ***Yacc***.

Программа ***Yacc*** работает примерно по тому же трехшаговому алгоритму, что и программа ***Lex***: сначала создается текст искомого анализатора на языке Си (Си++, Java, Pascal, …), затем он компилируется нужным компилятором, после чего может передаваться на исполнение, во время которого будет анализировать синтаксис подаваемых ему на вход текстов:



Исходная ***Yacc***-программа имеет три части: объявления, правила трансляции и подпрограммы поддержки. В исходном описании указанные разделы отделяются друг от друга строками из двух символов '%%'. По своей структуре эти части напоминают разделы, обрабатываемые генератором лексических анализаторов, однако в разделе объявлений вместо определений регулярных выражений записываются определения лексем, например,

```
%token DIGIT
%token ID
```

Лексемы, определенные в этом разделе, могут использоваться в правилах трансляции. Правила трансляции представляют собой обычные грамматические правила грамматик с действиями, записываемые по специальным правилам:

```
<Левая часть> : <Alt 1> {Семантическое действие 1}
              | <Alt 2> {Семантическое действие 2}
            . . .
              | <Alt n> {Семантическое действие n}
              ;
```

Такой вид правил полностью соответствует нормальным формам Бэкуса-Наура с учетом специфики представления информации в текстах программ. В правилах ***Yacc*** терминальные символы представляются одиночными символами в одинарных кавычках (*'c'*). Нетерминальными символами считаются строки из букв и цифр, не взятые в кавычки и не объявленные ранее как лексемы. Альтернативы в правилах разделяются символами вертикальная черта '|'. После последней альтернативы каждого правила ставится точка с запятой. Стартовым символом считается левая часть первого правила, но его можно определить с помощью директивы начала:

```
%start expr
```

Семантические правила записываются непосредственно на том языке программирования, на котором будет формироваться синтаксический анализатор, но в них допускается использование специальных символов:

- *$$* представляет значение атрибута, связанного с нетерминалом в левой части,
- *$i* представляет значение, связанное с *i*-м грамматическим символом (терминалом или нетерминалом) справа.

Семантическое действие, входящее в правило, обычно сводится к вычислению *$$* по *$i*:

```
expr  :  expr '+' term   { $$ = $1 + $3; } | term ;
```

По умолчанию выполняется простейшее семантическое действие:

```
{ $$ = $1; }
```

Подпрограммы поддержки пишутся на том языке программирования, на котором будет сформирован синтаксический анализатор. Среди этих подпрограмм обязательно должна существовать функция с именем *yylex()*, которая выполняет лексический анализ текстов исходного языка. Все остальные подпрограммы (например, процедуры обработки ошибочных ситуаций) добавляются только в случае их реальной необходимости. Лексический анализатор *yylex()* возвращает пары лексема-значение. Если функция возвращает лексему типа *DIGIT*, эта лексема должна быть предварительно объявлена в первом разделе спецификации. Значение, соответствующее обнаруженной лексеме, возвращается через предопределенную переменную *yylval*.

Генераторы синтаксических анализаторов должны иметь средства, позволяющие им разрешать неоднозначности, часто имеющиеся в контекстно-свободных грамматиках, в том числе в *LALR(1)*-грамматиках. Например, для правила грамматики

```
E → E+E|E-E|E*E|E/E|(E)|-E|number
```

в программе для ***Yacc*** следует писать такие правила:

```
%token number
%left ‘+’ ‘-’
%left ‘*’ ‘/’
%right UMINUS
%%
expr : expr ‘+’ expr { $$ = $1 + $3; } | expr ‘-’ expr { $$ = $1 - $3; }
     | expr ‘*’ expr { $$ = $1 * $3; } | expr ‘/’ expr { $$ = $1 / $3; }
     |      ‘-’ expr %prec UMINUS { $$ = - $2; }
     |      ‘(’ expr ‘)’          { $$ =   $2; }         | NUMBER;
```

Из-за неоднозначности грамматики при генерации анализатора могут возникать конфликты, о количестве которых генератор выдаст сообщение. Конфликты разрешаются применением по умолчанию двух следующих правил:

- Порядок правил имеет значение и, в случае конфликта, нужным правилом будет считаться правило, стоящее в списке правил перед конфликтующим.
- Из конфликтующих правил выбирается то, которое задает максимально длинный контекст. Это правило корректно разрешает конфликт, возникающий из-за кочующего ***else***.

Однако генератор ***Yacc*** позволяет пользователям самим влиять на способ разрешения конфликтов, задавая приоритеты и ассоциативность терминальных символов. В приведенном выше примере задано, что приоритет сложения равен приоритету вычитания, но меньше приоритета умножения и деления. Все эти операторы объявлены левоассоциативными. Можно также задавать правоассоциативность (*%right*) терминалов и их неассоциативность (*%nonassoc*), которая означает невозможность комбинирования операторов (например, операторов отношения *‘>’*). Уровни приоритетов лексем определяются порядком их перечисления в разделе объявлений, поэтому приоритет унарного минуса из приведенного примера оказывается выше приоритета умножения и деления.

В правило можно вставлять явное задание приоритета правила с помощью конструкции *%prec* *<терминал>* (как в предпоследней альтернативе на приведенном выше примере). Используемый терминал может быть искусственно введенным. Такой терминал никогда не возвращается лексическим анализатором и объявляется единственно для того, чтобы определить приоритет правила. Именно поэтому унарный минус, определяемый предпоследней альтернативой в приведенном примере, получает наивысший приоритет, по сравнению с другими операторами.

В отличие от генератора лексических анализаторов ***Lex***, который способен построить распознаватель по любому правильному регулярному выражению, генератор ***Yacc*** не всегда способен решить поставленную перед ним задачу. Это связано с тем, что заданная контекстно-свободная грамматика может оказаться не соответствующей ограничениям *LALR(1)*-грамматик. В этом случае генератор выдает сообщение об ошибке, то есть о наличии неразрешимого конфликта в *LALR(1)*-грамматике. Пользователю необходимо или преобразовать грамматику, или ввести дополнительные правила, облегчающие построения анализатора.

Современные генераторы синтаксических анализаторов ***Yacc*** (а также другие генераторы, выполняющие аналогичные построения, например, ***Bison***) представляют собой мощные программные инструменты. С их помощью построены реальные анализаторы для различных компиляторов и прикладных систем.

Литература

Основная литература

- Материалы к курсу “Системы программирования” и практикуму на ЭВМ (http://sp.cmc.msu.ru/win/courses/prak2/index.html, http://cmcmsu.no-ip.info/2course/).
- И. А. Волкова, Т. В. Руденко. “Формальные грамматики и языки. Элементы теории трансляции”, М.: МГУ, 1 9 9(Шифр в библиотеке МГУ: 5 В6 6 В-676; http://sp.cmc.msu.ru/courses/prak2/lang_grams.pdf).
- А. В. Гордеев, А. Ю. Молчанов. “Системное программное обеспечение”, СПб.: “Питер”, 2002.
- А. Ю. Молчанов. “Системное программное обеспечение. Учебник для вузов”, СПб.: Питер, 2003.
- Alfred V. Aho, Jeffrey D. Ullman. “The Theory of Parsing, Translation and Compiling”. Prentice-Hall, Inc., Englewood Cliffs, N. J., 1973 (А. Ахо, Дж. Ульман. “Теория синтаксического анализа, перевода и компиляции”, в 2-х т., М.: “Мир”, 1978).
- Andrew W. Appel, Maia Ginsburg. “Modern compiler implementation in C”, Cambridge University Press, 1998.
- P. J. Brown (ed.). “Software Portability”, Cambridge University Press, 1977 (“Мобильность программного обеспечения”, под ред. П. Брауна, М., Мир, 1980).
- Joseph M. Fox. “Software and its development”. Prentice-Hall, Inc., Englewood Cliffs, 1982 (Дж. Фокс, “Программное обеспечение и его разработка”, М., Мир, 1985).
- P. Genuys (ed.). “Programming Languages”, Academic Press, 1968 (“Языки программирования”, под ред. Ф. Женюи, М., “Мир”, 1972).
- David Gries. “Compiler construction for digital computers”, John Wiley and sons, Inc., 1971 (Д. Грис. “Конструирование компиляторов для цифровых вычислительных машин”, М., “Мир”, 1975).
- P. M. Lewis, D. J. Rosenkranz, R. E. Stearns. “Compiler Design Theory”, Addison-Wesley, Readin g , MA, 1 9 7 6(Ф. Льюис, Д. Розенкр анц, Р. Стир нз. “Теоретические основы проектирования компиляторов”, М.: “Мир”, 1979).
- Robert Morgan. “Building an optimizing compiler”, Butterworth-Heinemann, 1998.
- Steven S. Muchnick. “Advanced compiler design and implementation”, Morgan Kaufmann Publishers, Inc., 1997.

Дополнительная литература из библиотеки МГУ

- А. М. Вендров. “CASE-технологии. Современные методы и средства проектирования информационных систем”, http://www.citforum.ru/database/case/index.shtml.
- Л. Е. Карпов. "Архитектура распределенных систем программного обеспечения", М., МАКС Пресс, 2007. Шифр в библиотеке МГУ: 5ВГ66, К-265.
- Н. Н. Мансуров, О. Л. Майлингова. “Методы формальной спецификации программ: языки MSC и SDL”, М.: Изд-во “Диалог-МГУ”, 1998 (Шифр в библиотеке МГУ: 5ВГ66 М-238).
- И. О. Одинцов. “Профессиональное программирование. Системный подход”, СПб.: “БХВ-Петербург”, 2002 (Шифр в библиотеке МГУ: 5ВГ66 О-425).
- Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. “Compilers. Principles, techniques and tools”. Addison-Wesley Publishing Company, Inc., 1985 (Альфред Ахо, Рави Сети,

Джеффри Ульман. “Компиляторы. Принципы, технологии, инструменты”, М.: Изд. дом “Вильямс”, 2001. Шифр в библиотеке МГУ: 5ВГ66 А-955).
- Grady Booch. “Object-Oriented Analysis and Design with Applications”, Benjamin/Cummings, 1990, Addison-Wesley Publishing Company, Inc., 1994 (Г. Буч “Объектно-ориентированный анализ и проектирование с примерами приложений на С++”, 2-е издание, М., СПб.: “Бином” – “Невский диалект”, 1998. Шифр в библиотеке МГУ: 5ВГ66 Б-947; http://redlib.narod.ru/cidocs/buch.zip).
- Anton Eliëns. “Principles of Object-Oriented Software Development”. Second edition, Addison-Wesley, 2000 (А. Элиенс. “Принципы объектно-ориентированной разработки программ”, 2-е издание. М.: Издательский дом “Вильямс”, 2002. Шифр в библиотеке МГУ: 5ВГ66 Э-460).
- M. Fowler, K. Scott. “UML Distilled: Applying the Standard Object Modeling Language”. Addison-Wesley, 1999 (М. Фаулер, К. Скотт. “UML в кратком изложении. Применение стандартного языка объектного моделирования”, М.: “Мир, 1999. Шифр в библиотеке МГУ: 5ВГ66 Ф-282).
- Herb Schildt. “Teach Yourself C++”. Third edition, Osborne/McGraw-Hill, 1998 (Г. Шилдт. “Самоучитель С++”. 3-е изд. СПб: “БХВ-Петербург”, 2002. Шифр в библиотеке МГУ: 5ВГ66 Ш-576).
- Bjarne Stroustrup. “The C++ programming language”. Special edition. Addison-Wesley Longman, 2000 (Б. Страуструп. “Язык программирования С++”. Специальное издание, М.: “Бином”, 2005. Шифр в библиотеке МГУ: 5ВГ66 С-835).

Вспомогательная литература

Стандарты

- P. Naur (Editor), “Revised Report on the Algorithmic Language ALGOL 60”, Comm. ACM, 1963, v. 6, No. 1. pp. 1-17; Computer Journal, 1963, v. 5, No. 9, pp. 349-367; Num. Math., 1962, v. 4, No. 5, pp. 420-453 (“Алгоритмический язык АЛГОЛ-60. Пересмотренное сообщение”, М.: “Мир”, 1965).
- European Computer Manufacturers Association and American National Standards Institute, PL/1, Basis/1-12. ECMA/TC10, ANSI.X3J1, July 1974.
- “Modified Report on the Algorithmic Language ALGOL-60”, Computer Journal, v. 19, No. 4, Nov. 1976, pp. 364 – 379 (“Алгоритмический язык АЛГОЛ 60. Модифицированное сообщение”, под ред. А. П. Ершова, М.: “Мир”, 1982).
- American National Standard Programming Language FORTRAN, X3, 9-1978.
- “IEEE Standard for Microprocessor Assembly Language”. IEEE Std 694-1985.
- “IEEE Standard for Binary Floating-Point Arithmetic”. IEEE Std 754-1985.
- “IEEE Standard for Radix-Independent Floating-Point Arithmetic”. IEEE Std 854-1987.
- Standard for the C Programming Language. ISO/IEC 9899, 1990.
- M. C. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber. Capability Maturity Model for Software, Version 1.1, SEI Technical Report CMU/SEI-93-TR-024, Software Engineering Institute, Pittsburgh, Feb. 1993
  http://www.sei.cmu.edu/pub/documents/93.reports/pdf/tr24.93.pdf
- M. C. Paulk, C. V. Weber, S. M. Garcia, M. B. Chrissis, and M. Bush. Key Practices of the Capability Maturity Model, Version 1.1, SEI Technical Report CMU/SEI-93-TR-025, Software Engineering Institute, Pittsburgh, Feb. 1993.
  http://www.sei.cmu.edu/pub/documents/93.reports/pdf/tr25.93.pdf

- Alexander Stepanov, Meng Lee. “The Standard Template Library”. Doc No: X3J16/94-0030R1.
- ISO/IEC 12207:1995, Information Technology — Software life cycle processes, 1995. Amendments 2002, 2004.
- IEEE 1074-1997 IEEE Standard for Developing Software Life Cycle Processes, 1997.
- Standard for the C++ Programming Language. ISO/IEC 14882, 1998.
- IEEE/EIA 12207.0-1996 Industry Implementation of International Standard ISO/IEC 12207:1995, New York, Mar. 1998.
- IEEE/EIA 12207.1-1997 Industry Implementation of International Standard ISO/IEC 12207:1995 Software Life Cycle Processes – Life Cycle Data, New York, Apr. 1998.
- IEEE/EIA 12207.2-1997 Industry Implementation of Int'l Standard ISO/IEC 12207:1995 Software Life Cycle Processes — Implementation Considerations, New York, Apr. 1998.
- ГОСТ Р-1999. ИТ. Процессы жизненного цикла программных средств.
- Capability Maturity Model Integration (CMMI), Version 1.1. CMMI for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMMI-SE/SW/IPPD/SS, V1.1). Continuous Representation. SEI Technical Report CMU/SEI-2002-TR-011, Software Engineering Institute, Pittsburgh, March 2002. http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr011.pdf
- Capability Maturity Model Integration (CMMI), Version 1.1. CMMI for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMMI-SE/SW/IPPD/SS, V1.1). Staged Representation. SEI Technical Report CMU/SEI-2002-TR-012, Software Engineering Institute, Pittsburgh, March 2002. http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr012.pdf
- ISO/IEC 15288:2002, Systems engineering — System life cycle processes, 2002.
- ISO/IEC 15504-1-9, Information technology — Process assessment, Parts 1-9. 15504-1,3,4:2004, 15504-2:2003/Cor 1:2004, TR 15504-5:2004.

Методы и технологии программирования

- Leland L. Beck. “System Software, An Introduction to Systems Programming”. Addison-Wesley Publishing Company, Inc., 1985 (Бек Л. “Введение в системное программирование”, М.: “Мир”, 1988).
- Edsger W. Dijkstra. “A discipline of programming”, Academic Press, 1976 (Э. Дейкстра. “Дисциплина программирования”, М.: “Мир”, 1978).
- Sidney Fernbach (ed.). “Supercomputers”, Elsevier Science Publishers B. V., 1986 (“СуперЭВМ. Аппаратная и программная реализация”, под ред. Фернбаха, М.: “Радио и связь”, 1991).
- Barbara Liskov and John Guttag. “Abstraction and Specification in Program Development”, The MIT Press, McGraw-Hill Book Company, 1986 (Лисков Б., Гатэг Дж. “Использование абстракций и спецификаций при разработке программ”. М. “Мир”, 1989).
- Niklaus Wirth. “Systematic Programming. An Introduction”, Prentice-Hall, Inc., Englewood Cliffs, 1973 (Н. Вирт. “Систематическое программирование. Введение”, М.: “Мир”, 1977).
- Niklaus Wirth. “Algorithms + Data structures = Programs”. Prentice-Hall, Inc., Englewood Cliffs, 1976 (Н. Вирт. “Алгоритмы + структуры данных = программы”, М.: “Мир”, 1985).
- Niklaus Wirth. “Algorithms and data structures”, Prentice-Hall, Inc., Englewood Cliffs, 1986 (Н. Вирт. “Алгоритмы и структуры данных”, М.: “Мир”, 1989).

- Кулямин. “Технология программирования. Компонентный подход”, http://www.ispras.ru/~kuliamin/sdt-course.html.
- Philippe Kruchten, “The Rational Unified Process: An Introduction”, Second Edition, Addison Wesley Professional, 2000 (Филипп Крачтен, “Введение в Rational Unified Process”, “Вильямс”, 2002).
- http://www.omg.org/technology/documents/formal/uml.htm.

Системы программирования

- А. Ананьев, А. Федоров. “Самоучитель Visual Basic 6.0”, СПб.: “БХВ-Петербург”, 2003.
- А. Я. Архангельский. “Язык Pascal и основы программирования в Delphi. Учебное пособие”, М.: ООО “Бином-Пресс”, 2004.
- С. И. Бобровский. “Delphi 7. Учебный курс”, СПб.: “Питер”, 2003.
- В. И. Король. “Visual Basic 6.0, Visual Basic for Applications 6.0. Язык программирования”. Справочник с примерами, М.: КУДИЦ-ОБРАЗ, 2000.
- С. Н. Лукин. “Турбо-Паскаль 7.0. Самоучитель для начинающих”, 2-е изд., М., “Диалог-МИФИ”, 2004.
- А. В. Матросов, Ф. А. Новиков, Г. Е. Усаров, И. А. Харитонова. “Microsoft Office XP: Разработка приложений”, СПб.: БХВ-Петербург, 2003.
- В. Фаронов. “Delphi. Программирование на языке высокого уровня”, СПб.: “Питер”, 2003.
- David Gallardo, “Migrating to Eclipse: A developer's guide to evaluating Eclipse vs. JBuilder”. http://www-128.ibm.com/developerworks/library/os-ecjbuild (Дэвид Галлардо. “Миграция на Eclipse: Руководство разработчика по сравнительной оценке Eclipse и JBuilder”).
- David S. Platt. “Introducing Microsoft .NET”, Microsoft Press, 2001 (Дэвид С. Платт. “Знакомство с Microsoft .NET”, М.: Издательско-торговый дом “Русская Редакция”, 2001).
- “Developing Windows-Based Applications with Microsoft Visual Basic .NET and Microsoft Visual C# .NET”. Training Kit MCAD/MCSD, Microsoft Corporation, 2002 (“Разработка Windows-приложений на Microsoft Visual Basic .NET и Microsoft Visual C# .NET”. Учебный курс MCAD/MCSD, М.: “Русская редакция”, 2003).
- Ying Bay. “Applications Interface Programming Using Multiple Languages. A Windows Programmer's Guide”. Prentice Hall PTR, 2003 (И. Бей. “Взаимодействие разноязыковых программ. Руководство программиста”. Издательский дом “Вильямс”, М., СПб., К., 2005).

Макропроцессоры

- P. J. Brown. “Macro processors and Techniques for portable software”. John Wiley and sons, 1974 (П. Браун. “Макропроцессоры и мобильность программного обеспечения”, М.: “Мир”, 1977).
- M. Campbell-Kelly. “An Introduction to Macros”, MacDonald London, American Elsevier Inc., N. Y., 1973 (М. Кэмпбел-Келли. “Введение в макросы”, М., “Советское радио”, 1978).
- W. M. Waite. “Implementing Software for Non-Numeric Applications”, Prentice Hall, Englewood Cliffs, N. J., 1973.

Библиотеки

- Московский Государственный Университет имени М. В. Ломоносова, Научно-исследовательский Вычислительный Центр, “Библиотека численного анализа”, http://www.srcc.msu.su/num_anal/lib_na/libnal.htm.
- Ю. В. Тихомиров. “Самоучитель MFC”, изд. “БХВ – Санкт-Петербург”, 2000.
- M. D. McIlroy; “Mass Produced Software Components” in Naur, Randell, Buxton (eds.). Software Engineering: Concepts and Techniques, Proceedings of the NATO Conferences, New York, 88.98; 1976.
- The Numerical Algorithms Group. http://www.nag.co.uk/.
- The Portland Group™. http://www.pgroup.com/.
- Bjarne Stroustrup. “Parameterized Types for C++”. Proc. USENIX C++ Conference, Denver, October, 1988.
- D. R Musser, A Saini. “C++ Programming with the Standard Template Library. STL Tututorial and Reference Guide” New York: Addison-Wesley Publishing. 1996.
- D. R. Musser, A. A. Stepanov. “Algorithm-oriented Generic Libraries”. Software Practice and Experience, Vol. 24 (7), July 1994.

Системы управления конфигурацией

- Concurrent Versions System (CVS), http://ximbiot.com/cvs, http://www.cvs.ru.
- Rational ClearCase, Software Configuration Management, IBM Corporation, USA, (http://www.rational.com).
- Software Configuration Management System, Peforce Software, Inc. (http://www.perforce.com).
- “Subversion”. Version Control System. http://subversion.tigris.org/.

Тестирование программ

- Boris Beizer. “Black-Box Testing. Techniques for Functional Testing of Software and Systems”. John Wiley & Sons, Inc., 1995 (Борис Бейзер. “Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем”, Питер, 2004).
- C. Kaner, J. Falk, H. Quoc Nguen. “Testing Computer Software”. Thomson Publishing Company, 1993 (С. Канер, Дж. Фолк, Нгуен Е. К. “Тестирование программного обеспечения”, М.: “DiaSoft”, 2001).
- John D. McGregor, David A. Sykes. “A Practical Guide to Testing Object-Oriented Software”, Addison-Wesley, 2001 (Дж. Макгрегор, Д. Сайкс. “Тестирование объектно-ориентированного программного обеспечения. Практическое пособие”, М.: “DiaSoft”, 2002).
- Glenford J. Myers. “Software reliability. Principles and practices”. John Wiley and sons, 1976 (Г. Майерс. “Надежность программного обеспечения”, М., Мир, 1980).
- Glenford J. Myers. “The Art of Software Testing”. John Wiley and sons, 1979 (Г. Майерс. “Искусство тестирования программ”, М.: “Финансы и статистика”, 1982).
- Simon Wallace. “The ePMbook”. http://www.epmbook.com, 2004.

Автоматизация построения анализаторов программ

- M. E. Lesk. "Lex – a lexical analyzer generator". Tech. Rep. Computing Science Technical Report 39, Bell Laboratories, Murray Hill, NJ, 1975.

- R. W. Gray. "γ-GLA – a generator for lexical analyzers that programmers can use". In USENIX Conference Proceedings. USENIX Association, Berkeley, CA, pp. 147-160, 1988.
- V. Paxson. "Flex – Fast lexical analyzer generator". Lawrence Berkeley Laboratory, Berkeley, CA, ftp://ftp.ee.lbl.gov/flex-2.5.3.tar.gz, 1995.
- S. C. Johnson. "Yacc – yet another compiler compiler". Tech. Rep. CSTR-32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.
- Charles Donnelly, Richard Stallman. "Bison. The Yacc-compatible Parser Generator", version 2.1, (http://www.gnu.org/software/bison/manual/pdf/bison.pdf), Free Software Foundation, Inc., 2005 (Чарльз Доннелли, Ричард Столлмен. "Генератор синтаксических анализаторов, совместимый с YACC", http://linux.yaroslavl.ru/docs/altlinux/doc-gnu/bison/bison_toc.html).

Проект GNU

- http://gcc.gnu.org; http://gcc.gnu.org/wiki; http://www.gnu.org.
- http://www.gnu.org/software/emacs/, http://www.linux.org.ru/books/GNU/emacs/.
- Richard M. Stallman. “Using and porting the GNU compiler collection. Version 2.95”, Free Software Foundation, Inc., 1988 – 1999.

Распределенные системы

- Andrew S. Tanenbaum, Maarten van Steen. "Distributed Systems. Principles and paradigms". Prentice Hall, Inc., 2002 (Э. Таненбаум, М. ван Стеен. "Распределенные системы. Принципы и парадигмы". СПб.: Питер, 2003)
- Gustavo Alonso, Fabio Casati, Harumi Kuno, Vijay Machiraju. "Web Services. Concepts, Architectures and Applications". Springer-Verlag, 2004.
- "OSF DCE 1.2.2 Application Development Guide – Core Components", The Open Group, 1997.
- B. Viveney. "DCE and Object Programming". In W. Rosenberry (ed.) "DCE Today", pp. 251 – 264. Upper Saddle River, NJ, Prentice Hall Inc., 1998.
- Л. А. Калиниченко, М. Р. Когаловский, "Стандарты OMG: Язык определения интерфейсов IDL в архитектуре CORBA", Системы Управления Базами Данных, № 2, стр. 115-129, 1996
- Robert Orfali, Dan Harkey, Jeri Edwards. "Instant CORBA". Wiley Computer Publishing, John Wiley & Sons, Inc., 1997 (Р. Орфали, Д. Харки, Д. Эдвардс, "Основы CORBA", М., МАЛИП, 1999).
- А. А. Цимбал. "Технология CORBA для профессионалов". СПб.: Питер, 2001.
- Jon Siegel. "Quick CORBA™ 3". Wiley Computer Publishing, John Wiley & Sons, Inc., 2001 (Джон Сигел, "CORBA 3", М., МАЛИП, 2002).
- W. Richard Stevens. "UNIX Network Programming. Networking APIs", Prentice Hall PTR, 2nd edition, 1998 (У. Стивенс "Разработка сетевых приложений", СПб.: Питер, 2004).
- А. Касаткин. "Средства middleware и их классификация". PCWeek, № 19 (193), 1999.
- И. Ш. Хабибуллин. "Создание распределенных приложений на Java 2". СПб.: БХВ-Петербург, 2002.
- Eric Newcomer. "Understanding Web Services: XML, WSDL, SOAP and UDDI", Addison-Wesley, 2002 (Эрик Ньюкомер. "Веб-сервисы. Для профессионалов", СПб.: Питер, 2003).

- А. А. Цимбал, М. Л. Аншина. "Технологии создания распределенных систем. Для профессионалов". СПб.: Питер, 2003.
- http://www.corba.org
- http://www-128.ibm.com/developerworks/webservices/standards/
- http://www-128.ibm.com/developerworks/webservices/library/specification/ws-tx/
- http://www-128.ibm.com/developerworks/library/specification/ws-bpel/
- http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
- http://www.sei.cmu.edu/str/descriptions