ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Конспект книги
Пратт Т., Зелковиц М. «Языки программирования: разработка и реализация»

для студентов 3 курса ФПМИ

Новосибирск
2010

1. История языков программирования

С тех пор как в 50е гг. появились первые языки программирования высокого уровня, методы их разработки и реализации постоянно развиваются. Первые версии FORTRAN и LISP были разработаны в 50е гг., история языков Ada, C, Pascal, Prolog начинается в 70х гг., в 80е возникли такие языки, как C++, ML, Perl, Postscript, а 90х появились Java, C#.

Языки численных расчетов. Первый этап развития компьютерных технологий относится к периоду, который начался до Второй мировой войны и продлился до начала 40х. Во время Второй мировой войны основной задачей компьютеров было определение баллистических траекторий путем решения дифференциальных уравнений движения. В начале 50х начали появляться первые языки, использовавшие символьные системы обозначений. Грейс Хупер (Grace Hooper) возглавил в компании Univac группу, которая разрабатывала язык A0, а Джон Бэкус (John Backus) создал язык Speedcoding для IBM 701. Оба эти языка предназначались для преобразования простых арифметических выражений в исполняемый машинный код.

Настоящий прорыв произошел в 1957 г., когда Бэкус руководил группой по созданию языка FORTRAN, или транслятора формул (FORmula TRANslator). На ранней стадии разработки FORTRAN был ориентирован на численные вычисления, но конечной целью был вполне законченный язык программирования, включающий в себя управляющие структуры, условные операторы и операторы ввода-вывода. Поскольку немногие верили, что получится язык, способный конкурировать с языком ассемблера, в котором машинные команды кодировались вручную, основной задачей было создать эффективный исполняемый код, поэтому многие операторы разрабатывались с учетом специфики ЭВМ IBM 704. Концепции языка FORTRAN типа механизма трехветвевого перехода вытекала напрямую из аппаратной архитектуры IBM 407, а операторы типа READ INPUT TAPE сегодня выглядят весьма причудливо. Все это выглядело не очень изящно, но в то время еще не задумывались об элегантном программировании, зато разработанный язык позволял писать программы, которые выполнялись достаточно быстро на ЭВМ упомянутого типа.

FORTRAN оказался весьма удачным языком и оставался доминирующим вплоть до 70х гг. Следующая версия FORTRAN вышла в 1958 г. и стала называться FORTRAN II, а спустя несколько лет появился FORTRAN IV. Поскольку каждый производитель ЭВМ реализовывал для своих компьютеров собственную версию языка, то, понятное дело, царил хаос. В 1966 г. FORTRAN IV стал стандартом под названием FORTRAN 66 и с тех пор несколько раз подвергался пересмотру, в результате чего возникли стандарты FORTRAN 77 и FORTRAN 90. Наличие большого количества программ, написанных на ранних версиях языка, явилось причиной того, что создаваемые трансляторы должны были удовлетворять требованиям обратной совместимости, что препятствовало внедрению в язык новых идей и концепций программирования (нужно заметить, что до сих пор наиболее используемыми

остаются привычные программистам старшего поколения конструкции стандарта 77, хотя большинство современных компиляторов уже полностью поддерживают стандарт 95 года).

Поскольку FORTRAN оказался столь успешным языком, в Европе возникли опасения, что IBM будет доминировать в компьютерной отрасли. Немецкое общество прикладной математики (German society of applied mathematics — GAMM) создало комитет по разработке универсального языка. В то же время Association for Computing Machinery (ACM) организовала похожий комитет в США. Несмотря на то что у европейцев было некоторое беспокойство по поводу господства американцев, оба этих комитета слились в один. Под руководством Питера Наура (Peter Naur) этот комитет разработал IAL (International Algorithmic Language). Предлагавшееся название ALGOL (ALGOrithmic Language) было вначале отвергнуто. Но поскольку оно стало общеупотребительным, официальное имя IAL пришлось впоследствии изменить на ALGOL 58. Новая версия появилась в 1960 г., и ALGOL 60 (с небольшими изменениями, сделанными в 1962 г.) с 60х и до начала 70х гг. прошлого века был стандартом академического языка программирования.

Если FORTRAN разрабатывался для эффективного использования на IBM 704, перед разработчиками языка ALGOL стояли совершенно другие цели, а именно:

1. Сделать систему обозначений в ALGOL как можно ближе к стандартной математической.

2. ALGOL должен быть приспособлен для описания алгоритмов.

3. Программы на ALGOL должны были компилироваться в машинный язык.

4. ALGOL не должен быть привязан к конкретной машинной архитектуре.

Все эти цели оказались в 1958 г. недостижимы. Чтобы обеспечить независимость от архитектуры, в этот язык не были включены возможности ввода-вывода; для таких операций следовало создавать специальные процедуры. Хотя это и обеспечивало независимость языка от архитектуры компьютера, но в то же время приводило к тому, что каждая реализация неизбежно была несовместима с любой другой. Чтобы сохранить аналогию со строгой математикой, подпрограмма рассматривалась как макроподстановка, что породило концепцию *передачи параметров по имени*; которую очень сложно эффективно реализовать.

Хотя ALGOL и имел некоторый успех в Европе, он так и не получил коммерческого успеха в Америке, тем не менее его влияние на другие языки было достаточно велико. Бэкус был редактором отчета, в котором определялся язык ALGOL. Он использовал синтаксическую систему обозначений, сопоставимую с концепцией контекстно-свободного языка, разработанной Хомским (Chomsky). Так произошло внедрение теории формальных грамматик в область языков программирования. Поскольку Бэкус и Наур внесли огромный вклад в концепцию разработки ALGOL, эта система обозначений но-

сит их имя – нормальная форма Бэкуса-Наура (НФБ) (Backus–Naur Form — BNF).

Можно привести еще один пример влияния ALGOL. Компания по продаже компьютеров Burroughs, которая после слияния с компанией Sperry Univac образовала компанию Unisys, обнаружила работы польского математика Лукашевича (Lukasiewicz), который разработал методику, позволяющую записывать арифметические выражения без скобок, используя алгоритм вычислений с использованием стека. Эта методика оказала значительное влияние на теорию разработки компиляторов. Используя технологию, основанную на этой методике, компания Burroughs разработала компьютер B5500 со стековой архитектурой и вскоре реализовала компилятор языка ALGOL, значительно превышавший по скорости существовавшие в то время компиляторы языка FORTRAN.

С этого момента ситуация изменяется. В 60-е гг. была разработана концепция пользовательских типов данных, которая не была реализована ни в языке FORTRAN, ни в языке ALGOL. Язык Simula_67, разработанный норвежцами Найгардом (Nygaard) и Далом (Dahl), ввел концепцию классов в ALGOL. В 80-е гг. это натолкнуло Страуструпа (Stroustrup) на идею создать C++ как расширение C с добавлением понятия классов. В середине 60-х Вирт (Wirth) разработал расширение языка ALGOL – ALGOL-W, пользовавшийся меньшим успехом. Тем не менее в 70-х гг. он же разработал Pascal, который стал языком научного программирования тех лет. Другой комитет, ориентируясь на успех ALGOL 60, разработал язык ALGOL 68, который, однако, оказался слишком сложным для понимания и эффективной реализации.

С началом серийного выпуска в 1963 г. новых компьютеров модели 360 фирма IBM в своей лаборатории Hursley, находящейся в Англии, разработала новый язык — NPL (New Programming Language). В связи с некоторым недовольством сотрудников Национальной физической лаборатории (English National Physical Laboratory) язык был переименован в MPPL (Multy_Purpose Programming Language). В дальнейшем это название было сокращено до PL/I. Язык PL/I объединил в себе вычислительные возможности языка FORTRAN и возможности бизнес-программирования (обработки деловой информации), имевшиеся в языке COBOL. В 70-80е гг. PL/I пользовался некоторой популярностью (хотя он и был не очень удобен из-за слишком большого количества языковых конструкций, PL/I обладал большим количеством возможностей), в настоящее же время он практически забыт, поскольку вытеснен такими языками, как C++ и Ada.

BASIC был разработан, чтобы удовлетворить потребность в численных расчетах людей, не имеющих отношения к науке, однако в дальнейшем его возможности были расширены и вышли далеко за рамки первоначальных целей.

Языки обработки деловой информации. Сфера обработки деловой информации стала следующей после численных расчетов областью, которая привлекла внимание разработчиков языков. В 1955 г. группа сотрудников Univac под руководством Грейса Хупера (Grace Hooper) разработала язык

FLOWMATIC. Целью было создание приложений для обработки деловой информации с использованием некоторого англоподобного текста. В 1959 г. Министерство обороны США профинансировало совещание по разработке языка CBL (Common Business Language), который должен был стать бизнес-ориентированным языком, максимально использующим английский язык в качестве системы обозначений для своих программ. В связи с несогласованной деятельностью различных компаний для быстрой разработки этого языка был сформирован специальный комитет Short Range Committee. Хотя члены этого комитета думали, что они будут разрабатывать некий промежуточный вариант языка, оказалось, что разработанная ими спецификация, опубликованная в 1960 г., определила новый язык — COBOL (COmmon Business Oriented Language). COBOL пересматривался в 1961 и 1962 гг. и был стандартизован в 1968 г. В 1974 и 1984 гг. он снова подвергся пересмотру.

Языки искусственного интеллекта. Интерес к языкам искусственного интеллекта возник в 50-е гг., когда компанией Rand Corporation был разработан язык IPL (Information Processing Language). Версия IPL_V стала довольно широко известна, но ее использование ограничивалось тем, что IPL_V не был языком высокого уровня. Огромным шагом вперед стала разработка Джоном Мак_Карти (JohnMcCarthy), сотрудником Массачусетсского технологического института (MIT), языка LISP (LISt Processing) для компьютеров IBM 704. Версия LISP 1.5 стала стандартом для его реализации на многие годы. Развитие LISP продолжается до настоящего времени.

LISP разрабатывался как функциональный язык обработки списков. Естественной областью приложения LISP явилась разработка стратегии ведения игры, поскольку обычная программа, написанная на языке LISP, могла создавать дерево возможных направлений движения (как связанный список) и, продвигаясь по этому дереву, искать оптимальную стратегию. Другой естественной областью применения этого языка стал автоматический машинный перевод текста, где одна цепочка символов может заменяться на другую. В этой области первой разработкой был язык COMIT, созданный Ингве (Yngve), сотрудником MIT. Каждый оператор программы, написанной на этом языке, был очень похож на контекстно-независимое правило и представлял собой набор замен, которые можно было осуществить, если в исходных данных обнаруживалась конкретная цепочка символов. Поскольку Ингве запатентовал свой код, группа разработчиков из AT&T Bell Telephone Laboratories решила создать свой собственный язык — так появился SNOBOL.

Если LISP создавался как язык обработки списков для универсальных приложений, Prolog стал специализированным языком с основанными на понятиях математической логики структурами управления и стратегией реализации.

Системные языки. В целях повышения эффективности выполняемых программ использование языка ассемблера в системной области продолжалось достаточно долго даже после того, как для приложений в других областях стали использоваться языки высокого уровня. Все изменилось с появле-

нием языка С. С развитием в 70-е гг. конкурентоспособной среды UNIX, написанной в основном на языке С, была доказана эффективность использования языков высокого уровня и в системной области.

2. Эволюция архитектуры программного обеспечения

На развитие языков программирования огромное влияние оказывает то оборудование, на котором должны выполняться написанные на них программы, и операционные среды, установленные на этом оборудовании.

Эра универсальных ЭВМ. С момента появления первых компьютеров в 40-е и вплоть до 80-х гг. в области вычислений доминировали большие универсальные ЭВМ. Дорогостоящий компьютер занимал тогда огромное помещение и обслуживался большим количеством специалистов.

Пакетные среды. Самая ранняя и простая операционная среда полностью состояла из внешних файлов с данными. Программа брала несколько входных файлов с данными, обрабатывала их и создавала несколько выходных файлов с данными. Такая операционная среда называется *средой пакетной обработки* (batch_processing), поскольку входные данные группируются в *пакеты* внутри файлов и в виде пакетов обрабатываются программой. 80-колонная перфокарта, или карта Холлерита (Hollerith card), названная так по имени Германа Холлерита (Herman Hollerith), придумавшего ее для переписи населения США, была неотъемлемой частью компьютеров 60-80 гг.

Такие языки, как FORTRAN, COBOL и Pascal, изначально разрабатывались под пакетную среду выполнения, хотя сейчас могут использоваться в интерактивной и встроенной операционной средах.

Интерактивные среды. В начале 70-х гг., ближе к концу эпохи универсальных ЭВМ, появилось интерактивное программирование. Чтобы не использовать при создании программы пачку перфокарт, к компьютеру были подсоединены электронно-лучевые мониторы. Появились компьютеры с возможностью *разделения времени*. В таких системах каждому пользователю выделялись небольшие кванты процессорного времени. Поскольку пользователи зачастую тратят большую часть компьютерного времени на обдумывание задач, а не на непосредственное взаимодействие с компьютером, то те немногие, кто в данный момент реально используют компьютер, имеют возможность пользоваться большим количеством временем, чем отведенная им квота в два отрезка времени в секунду.

Влияние на языки программирования. В языках, разработанных для пакетной среды, файлы обычно являются основой для большинства структур ввода-вывода. Хотя файлы также можно использовать и для интерактивного ввода-вывода на терминал, в этих языках не было необходимости в реализации специальной возможности интерактивного ввода-вывода. Например, файлы обычно хранятся в виде записей фиксированной длины, однако при интерактивном вводе данных программа должна считывать каждый символ по мере его введения с клавиатуры. Также обычно структура ввода-вывода не

обеспечивает доступа к специальным устройствам ввода-вывода, используемым во встроенных системах.

Ошибки, вызывающие остановку выполнения программы в средах пакетной обработки, допустимы, но они дорого обходятся, поскольку после исправления ошибки программу нужно запускать заново. Также в этих средах программист не имеет возможности немедленно исправить вручную обнаруженную ошибку. Таким образом, при разработке этих языков возможностям обнаружения и обработки ошибок и исключительных ситуаций непосредственно в программе придавалось большое значение. Программа должна была обрабатывать наиболее вероятные ошибки и продолжать свое выполнение без каких-либо остановок.

Третьей отличительной характеристикой среды пакетной обработки является отсутствие временных ограничений на выполнение программы. Обычно в языке, разработанном для такой среды, не заложены средства для отслеживания скорости выполнения или прямого ее контроля.

Операции интерактивного ввода-вывода существенно отличаются от стандартных операций с файлами, которые заложены в большинстве языков, разработанных для пакетных сред обработки. И это затрудняет адаптацию подобных языков программирования к интерактивным средам.

Например, язык С включает в себя функцию доступа к строкам текстового файла, а также функцию, которая выводит на терминал все символы, вводимые пользователем с клавиатуры, в то время как в языке Pascal прямой ввод текста с терминала выглядит очень громоздко. Этим объясняется популярность использования языка С (и разработанного на его основе С++) для написания интерактивных программ.

Также в интерактивных средах используется совершенно другой подход к обработке ошибок. Если пользователь вводит неправильные входные данные с клавиатуры, программа может вывести сообщение об ошибке и предложить произвести исправления. Средства языка для обработки ошибок внутри программы (например, возможность пропустить ошибку и выполнять программу дальше) становятся менее существенными. Хотя (в отличие от сред пакетной обработки) завершение работы программы в связи с возникшей ошибкой, как правило, не применяется.

Часто в интерактивных программах приходится использовать некоторые временные ограничения. Например, в видеоиграх, если пользователь в течение определенного времени не реагирует на выведенную сцену, активизируется некоторая реакция со стороны программы. Если интерактивная программа работает настолько медленно, что не успевает отвечать на вводимую команду в течение разумного времени, она считается непригодной для использования.

Эра персональных компьютеров

Если заглянуть в прошлое, то эра универсальных ЭВМ с разделением времени продлилась очень недолго – с начала 70-х до середины 80-х. На смену им пришли персональные компьютеры. Аппаратные технологии шли вперед семимильными шагами. Микрокомпьютеры, в которых целый процессор

размещался на одной квадратной пластинке из пластика или кремния размером пару сантиметров, с каждым годом становились все быстрее и дешевле. Стандартные универсальные ЭВМ уменьшились в размерах и из комнаты, заполненной стойками и накопителями на магнитных лентах, превратились в декоративную офисную машину.

В 1978 г. компания Apple выпустила компьютер Apple II, первый по-настоящему коммерческий персональный компьютер. Он представлял собой небольшую настольную машину, на которой запускался BASIC. Эта машина оказала огромное влияние на рынок образовательных услуг, однако деловой мир скептически отнесся к минимизированному компьютеру Apple.

Состояние дел изменилось в 1981 г. Фирма IBM выпустила свой персональный компьютер, а фирма Lotus разработала свое приложение Lotus 1–2–3, основанное на программе обработки электронных таблиц VisiCalc. Эта программа стала первой из *прикладных программ_«приманок»* (для расширения круга потенциальных заказчиков), которыми промышленность была вынуждена пользоваться. Именно с этого момента персональные компьютеры стали пользоваться неожиданным прежде успехом.

Началом современной эры персональных компьютеров можно считать январь 1984 г., когда в США проходили матчи на кубок по американскому футболу. Именно во время трансляции матчей на этот кубок по телевидению и была показана реклама компьютера Macintosh фирмы Apple. Он характеризовался оконным графическим пользовательским интерфейсом с мышью для ввода данных. Хотя фирма Xerox разработала эту технологию в своем исследовательском центре PARC (PaloAlto Research Center) раньше, компьютер Macintosh стал первым коммерческим применением данной технологии. Позже внешний вид интерфейса Macintosh был заимствован компанией Microsoft для своей операционной системы Windows и стал основным для персональных компьютеров.

Среда встроенных систем. Встроенные компьютеры являются боковой ветвью развития персональных компьютеров. *Встроенной компьютерной системой* называется система, которая управляет частью более крупной системы, такой как промышленный завод, станок, автомобиль или даже тостер. Поскольку компьютерная система стала составной частью более крупных систем, то ее неисправность обычно означает сбой в работе всей системы в целом. В отличие от обычных персональных компьютеров, где неудобства от сбоя программы обычно связаны лишь с необходимостью ее перезапуска, сбой встроенного приложения может быть связан с угрозой для человеческой жизни. Сбой компьютера в атомной промышленности может вызвать перегрев атомного реактора, а сбой компьютера в больнице может привести к прекращению мониторинга пациентов. Поэтому к надежности и точности приложений, используемых в таких областях, предъявляются повышенные требования. В этой области широко используются языки Ada, C, C++, поскольку они отвечают специфическим требованиям сред встроенных систем.

Влияние на языки программирования. С появлением персональных компьютеров вновь изменилась роль языка. Во многих прикладных областях

производительность перестала быть основным требованием. Компьютер, снабженный таким удобным пользовательским интерфейсом, как многооконный интерфейс Windows, полностью управляется одним пользователем. Благодаря достаточно низким ценам на компьютеры отпала необходимость в разделении времени. Задачей первостепенной важности стала разработка языков с хорошей интерактивной графикой.

В настоящее время основным типом пользовательского интерфейса являются интерфейсы с «оконной» конфигурацией. Пользователи персональных компьютеров, как правило, хорошо знакомы с многооконным интерфейсом: такие понятия, как окно, значок, полоса прокрутки, меню и другие многочисленные элементы управления графического интерфейса стали привычными для абсолютного большинства пользователей. Однако программирование таких пакетов программ может оказаться сложным делом. Поэтому поставщики операционных систем с многооконными пользовательскими интерфейсами создают специальные библиотеки пакетов для работы с окнами. Для облегчения разработки приложений разработчики первым делом должны ознакомиться с этими библиотеками.

Естественной моделью для данной среды является объектно-ориентированное программирование. Использование языков Java и C++ с их иерархией классов облегчает взаимодействие с пакетами, разработанными сторонними производителями. Язык же C# фактически специально разрабатывался под удобное программирование интерактивных оболочек.

Программы, написанные для встроенных систем, обычно работают без основной операционной системы и обычной среды взаимодействия с файлами и устройствами ввода-вывода. Наоборот, такие программы должны обращаться к нестандартным устройствам ввода-вывода через специальные процедуры, учитывающие все особенности конкретного устройства. Поэтому в языках, используемых для встроенных систем, файлам и связанным с ними операциям ввода-вывода уделяется меньше внимания. Доступ к специальным устройствам, как правило, осуществляется с использованием тех средств языка программирования, которые обеспечивают доступ к конкретным аппаратным регистрам, областям памяти, обработчикам прерываний или к подпрограммам, написанным на ассемблере или другом языке низкого уровня.

Во встроенных системах особенно важна обработка ошибок. Обычно каждую программу составляют так, чтобы она могла самостоятельно обработать любую ошибку и принять меры для восстановления и продолжения своей работы. Как правило, завершение работы программы при возникновении ошибки не считается допустимым выходом из положения, кроме случаев катастрофического сбоя системы. Кроме того, в таких системах обычно нет пользователя, который мог бы в интерактивном режиме устранить ошибку.

Встроенные системы, как правило, работают в *режиме реального времени*, то есть большая система, в которую интегрирована компьютерная система, требует от нее ответов на запросы и выдачу выходного сигнала в течение строго определенных интервалов времени. Например, управляющий самолетом компьютер должен мгновенно реагировать на изменение высоты и ско-

рости полета. Для функционирования этих программ в режиме реального времени язык должен поддерживать такие возможности, как отслеживание интервалов времени, реакция на задержку сверх установленного времени (которая может означать сбой некоторой части системы), запуск и завершение определенных действий в назначенное время.

Наконец, встроенная компьютерная система обычно является *распределенной системой*, состоящей из нескольких компьютеров. Программа, работающая в таких распределенных системах, обычно состоит из ряда одновременно выполняемых задач, каждая из которых управляет одной из частей системы или наблюдает за ней. Главная программа, если таковая существует, занимается только запуском этих задач. Будучи однажды запущенными, эти задачи обычно выполняются одновременно и независимо друг от друга, поскольку необходимость в их остановке возникает, только если по какой-либо причине происходит сбой или остановка всей системы.

Сетевая эра

Распределенная обработка данных. По мере того как в 80-е гг. компьютеры становились более быстродействующими, меньше в размерах и дешевле, они начали проникать в деловой мир. Для использования в крупных организациях были разработаны локальные вычислительные сети (ЛВС) с архитектурой *клиент–сервер*, использующие линии телекоммуникаций для связи между компьютерами. Программа-*сервер* должна обеспечивать доступ к информации, а множественные *клиентские* программы могут запрашивать сервер для получения этой информации.

Интернет. В середине 90-х гг. наблюдалось преобразование распределенных ЛВС в международную глобальную сеть Интернет. В 1970 г. DARPA (Defense Advanced Research Projects Agency) начало разработку проекта по соединению универсальных вычислительных машин в большую, надежную и защищенную сеть. Целью ее создания было обеспечение избыточности на случай войны, так чтобы военные могли иметь доступ к компьютерам из любой точки страны. К счастью, сеть ARPANET не пришлось использовать для этих целей, и с середины 80-х гг. военная сеть ARPANET преобразовалась в ориентированную на исследователей сеть Интернет. Впоследствии в сеть добавлялись новые компьютеры, и в настоящее время во всем мире каждый пользователь может подключить свой компьютер к сети Интернет. Миллионы машин соединены в динамически изменяемый комплекс сетевых серверов.

На заре функционирования глобальной сети Интернет доступ в нее требовал наличия двух типов компьютеров. Пользователь должен был находиться за клиентским персональным компьютером. Для получения информации он должен был подсоединиться к соответствующему серверу. Для этого использовались протоколы telnet и протокол передачи файлов FTP (file transfer protocol). Протокол telnet позволял на компьютере пользователя эмулировать удаленный терминал сервера, что позволяло пользователю непосредственно общаться с удаленным сервером, в то время как протокол FTP просто позволял клиентской машине посылать файлы на сервер или получать их с серве-

ра. В обоих случаях пользователь должен был знать, на какой именно машине находится необходимая ему информация.

В то же самое время был разработан третий протокол — простой протокол передачи сообщений (SMTP — Simple Mail Transfer Protocol). Протокол SMTP – основа сегодняшней электронной почты. Каждый пользователь имеет локальное регистрационное имя на клиентской машине, а каждая машина имеет уникальное собственное имя. Послать сообщение можно, используя программу, поддерживающую протокол SMTP, и зная имя пользователя и имя машины, на которой он зарегистрирован. Здесь важно отметить, что, как правило, нет необходимости знать точный адрес компьютера, на котором зарегистрирован данный пользователь. Нет никакой необходимости знать точный адрес машины в Интернете.

В конце 80-х основной целью стало упрощение поиска информации в Интернете. Прорыв в этом направлении осуществился в Европейском институте ядерных исследований (CERN), находящемся в Женеве (Швейцария). Бернерс-Ли (Berners-Lee) разработал концепцию *гиперссылок* в рамках языка HTML (HyperText Markup Language) как способа навигации в Интернете. После создания в 1993 г. web-браузера Mosaic и добавления к Интернет-технологиям протокола передачи гипертекстов HTTP (HyperText Transfer Protocol) наконец-то произошло открытие Интернета для широких слоев населения. К концу XX столетия изменилась целая структура поиска информации и получения знаний, так как наличие доступа к Интернету (имеющегося у значительной части населения) позволяет отыскивать необходимые сведения в любом уголке Всемирной паутины.

Влияние на языки программирования. Появление Всемирной паутины (WWW – World Wide Web) вновь изменило роль языков программирования. Вычисления снова стали централизованными, но существенно иным образом, нежели в раннюю эру универсальных компьютеров. По всему миру создаются крупные серверы информационных архивов. Для получения информации пользователи подключаются к этим серверам через Интернет, а для ее обработки (например, для создания отчета) используют локальные клиентские машины. Вместо того чтобы распространять миллионы копий нового программного обеспечения, поставщик может просто выложить продукт на сайт WWW, а пользователь может загрузить его себе на машину для локального использования. Чтобы пользователь мог загрузить программный продукт, а поставщик программного обеспечения имел возможность получить плату за использование этого продукта, необходим язык программирования, позволяющий вести диалог между клиентским компьютером и сервером. Развитие электронной торговли напрямую зависит от наличия языков с такими возможностями.

Изначально web_страницы были статическими: можно было просмотреть текст, рисунки или графики. Для доступа к другой странице пользователь мог щелкнуть на ее адресе URL (Uniform Resource Locator). Однако для развития электронной коммерции информация должна передаваться в обоих направлениях между клиентской машиной и сервером, поэтому web-

страницы должны были стать более активными. Подобные возможности обеспечиваются такими языками программирования, как Perl и Java.

Использование WWW поставило перед языками такие проблемы, которые не были очевидны в предыдущие две эры. Одна из них — безопасность. Посетитель web-сайта должен быть уверен в том, что его владелец не имеет злого умысла и не испортит клиентский компьютер, удалив с него информацию. Эта проблема, характерная для систем с разделением времени, отсутствует для персональных компьютеров, к которым, в принципе, имеет доступ только сам пользователь. Следовательно, доступ со стороны web-сервера к локальным файлам пользователя должен быть ограничен.

Еще одна важная проблема — производительность. Хотя персональные компьютеры стали очень быстродействующими, линии связи, соединяющие пользователя с Интернетом, как правило, имеют ограниченную скорость передачи. Вдобавок, хотя сами машины достаточно быстры, при подключении к серверу достаточно большого количества пользователей он может оказаться перегружен. Чтобы избежать возникновения таких ситуаций, можно обрабатывать информацию на клиентской машине, а не на сервере. Чтобы разгрузить сервер за счет клиентской машины, он должен переслать клиенту небольшую исполняемую программу. Проблема состоит в том, что сервер не знает, каким компьютером является клиентская машина, поэтому не ясно, какого вида должна быть исполняемая программа. Язык Java был специально разработан для решения этой проблемы.

3. Основные факторы, влияющие на развитие языков программирования

Возможности компьютеров. Компьютеры эволюционировали от огромных, медленных и дорогих ламповых машин 50-х гг. до современных суперкомпьютеров и микрокомпьютеров. В то же время между аппаратной частью компьютера и языком программирования появилось промежуточное звено, представляющее собой программное обеспечение операционных систем. Эти факторы оказали влияние как на структуру языков, так и на стоимость использования тех или иных языковых возможностей.

Области применения. В 50-е гг. компьютеры использовались лишь в военной отрасли, науке, деловом мире и промышленности, где высокая стоимость была обоснованной. В настоящее же время их применение распространилось на область компьютерных игр, программ для персональных компьютеров, Интернета и вообще на приложения во всех областях человеческой деятельности. Требования, специфические для этих новых областей применения, влияют как на конструирование новых языков, так и на пересмотр и расширения старых языков.

Методы программирования. Структурное строение языка отражает изменяющееся с течением времени наше представление о том, что является хорошим методом написания большой и сложной программы, а также отра-

жает изменяющуюся со временем среду, в которой осуществляется программирование.

Методы реализации. Усовершенствование методов реализации отражается на выборе тех новых свойств, которые добавляются во вновь разрабатываемые языки.

Теоретические исследования. Исследование концептуальных основ разработки и реализации языка с помощью формальных математических методов углубляет понимание сильных и слабых сторон конкретного языка, что отражается на добавлении тех или иных свойств при создании новых языков.

Стандартизация. Необходимость в стандартных языках, которые могут быть легко реализованы в различных компьютерных системах (что позволяет переносить программы с одного компьютера на другой), сильно влияет на эволюцию принципов разработки языков программирования.

В таблице кратко описаны факторы, оказавшие наиболее важное влияние на развитие языков программирования во второй половине XX столетия.

Таблица 1. Факторы, повлиявшие на развитие языков программирования

Годы	Факторы и новые технологии
1951–1955	**Аппаратная часть**: компьютеры на электронных лампах; память с ртутной линией задержки. **Методы:** языки ассемблера; основные концепции; подпрограммы; структуры данных. **Языки:** экспериментальное использование компиляторов выражений
1956–1960	**Аппаратная часть**: запоминающие устройства на магнитных лентах; память на сердечниках; схемы на транзисторах. **Методы**: ранние технологии компилирования; НФБ-грамматики; оптимизация кода; интерпретаторы; методы динамического распределения памяти и обработка списков. **Языки:** FORTRAN, ALGOL 58, ALGOL 60, LISP
1961–1965	**Аппаратная часть**: семейства совместимых архитектур, запоминающие устройства на магнитных дисках. **Методы**: мультипрограммные операционные системы; синтаксические компиляторы. **Языки**: COBOL, ALGOL 60 (новая версия), SNOBOL, JOVIAL
1966–1970	**Аппаратная часть**: увеличение размера и быстродействия при уменьшении стоимости; микропрограммирование; интегральные схемы. **Методы:** системы с разделением времени; оптимизирующие компиляторы; системы написания трансляторов. **Языки**: APL, FORTRAN 66, COBOL 65, ALGOL 68, SNOBOL4, BASIC, PL/1, SIMULA 67, ALGOL-W
1971–1975	**Аппаратная часть**: мини-компьютеры; запоминающие устройства небольшой емкости; полупроводниковая память.

	Методы: верификация программ; структурное программирование; технологии программирования. **Языки**: Pascal, COBOL 74, PL/1 (стандарт), C, Scheme, Prolog
1976–1980	**Аппаратная часть**: микрокомпьютеры; запоминающие устройства большой емкости; распределенные вычисления. **Методы**: абстракция данных; формальная семантика; технологии программирования: параллельная, встроенная и в режиме реального времени. **Языки**: Smalltalk, Ada, FORTRAN 77, ML
1981–1985	**Аппаратная часть**: персональные компьютеры; рабочие станции; видеоигры; локальные вычислительные сети; ARPANET. **Методы**: объектно-ориентированное программирование; интерактивные среды разработки; синтаксические редакторы. **Языки**: Turbo Pascal, Smalltalk-80, Prolog, Ada 83, Postscript
1986–1990	**Аппаратная часть**: эра микрокомпьютеров, автоматизированное рабочее место (АРМ) проектировщика, архитектуры RISC, Интернет. **Методы**: клиент-серверные вычисления. **Языки**: FORTRAN 90, C++, SML (Standart ML)
1991–1995	**Аппаратная часть**: очень быстрые и недорогие рабочие станции и микрокомпьютеры; архитектура с массовым параллелизмом; звук, видео, факс, мультимедиа. **Методы**: открытые системы, среды разработки. **Языки**: Ada 95, языки создания процессов (TCL, Perl), HTML
1996–2000	**Аппаратная часть:** компьютеры – дешевые приспособления; персональный электронный помощник; Всемирная паутина WWW; домашние кабельные сети; большой объем дисковой памяти (гигабайты). **Методы**: электронная коммерция. **Языки**: Java, Javascript, XML

4. Критерии сравнения языков

Механизмы разработки языка высокого уровня должны постоянно совершенствоваться. Каждый достаточно известный язык имеет свои недостатки, но тем не менее все они относительно удачны по сравнению с сотнями других языков, которые были разработаны, реализованы, использовались какое-то время, а потом были преданы забвению.

Некоторые причины успеха или неуспеха языка могут быть внешними по отношению к самому языку. Так, использование языков COBOL или Ada в Соединенных Штатах для разработки приложений в некоторых предметных областях было регламентировано указом правительства. Аналогично часть успеха языка FORTRAN можно отнести к его большой поддержке различными производителями вычислительной техники, которые тратили много уси-

лий на его изящные реализации и подробные описания. Часть успеха SNOBOL4 в 70_е гг. можно приписать превосходному описанию этого языка. Широкое распространение таких языков, как LISP и Pascal объясняется как их использованием в качестве объектов теоретического изучения студентами, специализировавшимися в области разработки языков программирования, так и реальной практической значимостью этих языков.

Несмотря на большое влияние некоторых из перечисленных внешних причин, в конце концов, именно программисты иногда, может быть, косвенно, решают, каким языкам жить, а каким нет. Существует множество причин, по которым программисты предпочитают тот или иной язык. Рассмотрим некоторые из них.

1. **Ясность, простота и единообразие понятий языка**. Язык программирования обеспечивает как систему понятий для обдумывания алгоритмов, так и средства выражения этих алгоритмов. Язык должен стать помощником программиста задолго до стадии реального кодирования. Он должен предоставить ясный, простой и единообразный набор понятий, которые могут быть использованы в качестве базисных элементов при разработке алгоритма.

С этой целью желательно иметь минимальное количество различных понятий с как можно более простыми и систематизированными правилами их комбинирования. Это свойство мы называем *концептуальной целостностью.*

Синтаксис языка влияет на удобство и простоту написания и тестирования программы, а в дальнейшем способствует ее пониманию и модификации. Центральным моментом здесь является удобочитаемость программы. Слишком лаконичный синтаксис может оказаться удобным при написании программ (особенно для опытного программиста), однако, когда такую программу нужно модифицировать, в ней оказывается нелегко разобраться. Программы на APL обычно настолько непонятны, что даже их разработчики спустя несколько месяцев после завершения работы затрудняются их расшифровать. Многие языки содержат такие синтаксические конструкции, которые сами подталкивают к неправильному восприятию программ, поскольку два почти одинаковых оператора на самом деле имеют кардинально различные значения. Например, появление пробела в операторе языка SNOBOL4 может полностью изменить его смысл. Хороший язык характеризуется тем, что конструкции, *обозначающие* различные понятия, должны и выглядеть совершенно по-разному, то есть семантические отличия должны отражаться в синтаксисе.

2. **Ортогональность**. Термин «*ортогональность*» означает, что любые возможные комбинации различных языковых конструкций будут осмысленными. Например, предположим, что язык позволяет задать выражение, которое вычисляет численное значение, а также задать условный оператор, в котором вычисляется выражение, с единственной целью — получить булево значение true или false. Эти две языковые конструкции (выражение и условный оператор) ортогональны, если любое выражение можно использовать (и вычислять) внутри условного оператора.

Когда конструкции языка ортогональны, язык легче выучить и на нем легче писать программы, поскольку в нем меньше исключений и специальных случаев, требующих запоминания. Отрицательной стороной ортогональности является то, что программа никогда не будет выдавать ошибки при компиляции, даже если она содержит комбинацию возможностей, которые логически не согласованы или крайне неэффективны при выполнении.

3. **Естественность для приложений**. Язык должен иметь такой синтаксис, который при правильном использовании позволяет отражать в самой структуре программы лежащие в основе реализуемого ею алгоритма логические структуры. В идеале должна существовать возможность прямого перевода эскиза такой программы в подходящие программные операторы, отражающие структуру алгоритма. Последовательный, параллельный, логический и другие алгоритмы имеют различные естественные структуры, которые представлены в программах, написанных на языках, предназначенных для реализации этих алгоритмов. Язык должен предоставлять соответствующие решаемой задаче структуры данных, операции, структуры управления и естественный синтаксис. Одной из важнейших причин распространения того или иного языка является его естественность. Язык, соответствующий определенному классу приложений, может сильно облегчить создание отдельных программ в этой области. В качестве примера языков с очевидной направленностью на решение конкретных классов задач можно привести Prolog с уклоном в сторону дедукции и C++, предназначенный для объектно-ориентированных разработок.

4. **Поддержка абстракций**. Даже в наиболее естественном для данной предметной области языке программирования остается некоторый существенный пробел. Это пробел между абстрактными структурами данных и операциями, которые характеризуют решение задачи, и конкретными базовыми структурами данных и операциями, встроенными в язык. Например, язык C вполне может подойти для составления расписания занятий в университете, хотя в нем напрямую не существует естественных для этого приложения абстрактных структур данных, таких как *студент, курс лекций, преподаватель*, *аудитория,* и таких абстрактных операций, как *определить студента в группу* и *назначить аудиторию для группы*. Важной частью работы программиста является разработка конкретных абстракций для решения задачи и их последующая реализация с использованием базовых возможностей реального языка программирования. В идеале, язык должен позволять определять структуры данных, типы данных и операции и поддерживать их как самодостаточные абстракции. В этом случае программист сможет использовать их в других частях программы, зная только их абстрактные свойства и не вникая в их фактическую реализацию. Как Ada, так и C++ были разработаны именно по причине отсутствия этой возможности в более ранних языках, таких как Pascal и C соответственно.

5. **Удобство верификации программы**. Основным требованием является надежность программы, написанной на том или ином языке. Существует множество технологий для проверки правильности выполнения программой

своих функций. Правильность программы можно доказать с помощью формальных методов верификации, проверкой без выполнения (путем чтения текста программы и исправления ошибок), также она может быть *протестирована* путем ее выполнения с тестовыми входными данными и проверкой выходных результатов в соответствии с ее спецификацией и т. д.

Для проверки больших программ обычно используется некая комбинация всех этих методов. Даже если язык обеспечивает на первый взгляд много возможностей для облегчения программирования, но проверка написанных на нем программ затруднительна, он менее надежен, чем язык, поддерживающий и упрощающий проверку программы. Основной фактор, влияющий на упрощение проверки программы, - простота семантики и синтаксических структур.

6. **Среда программирования**. Наличие в языке программирования технически развитых конструкций и структур (выражений, структур управления, типов данных и структур данных) – это только один аспект, влияющий на широту его использования. Наличие подходящей среды программирования может сделать технически слабый язык более легким в применении, нежели сильный язык при незначительной внешней поддержке. Можно составить длинный список разнообразных определяющих факторов, которым должна удовлетворять среда программирования, но возглавляет его, несомненно, требование наличия в ней надежной, эффективной и хорошо документированной реализации языка программирования. Специализированные текстовые редакторы и тестирующие пакеты, которые отражают специфику языка и работы с ним, могут сильно ускорить написание и тестирование программ. Средства для поддержки и модификации нескольких версий программы могут облегчить разработку больших программ.

7. **Переносимость программ**. Одним из важных критериев для многих программных проектов является *переносимость* разработанных программ с компьютера, на котором они были написаны, на другие компьютерные системы.

Удобным средством создания переносимых программ являются широкодоступные языки, определение которых не зависит от возможностей различных машин. Такие языки, как Ada, FORTRAN, C и Pascal имеют стандартные определения, позволяющие реализовывать переносимые приложения.

Другие языки (например, ML) происходят от единственной централизованной реализации (single_source implementation), позволяя разработчику языка осуществлять некоторый контроль над свойствами, определяющими его переносимость.

8**. Стоимость использования**. Стоимость использования является существенным компонентом оценки языка программирования и складывается из нескольких составляющих:

Стоимость выполнения программы. На заре компьютерных вычислений стоимость связывалась в основном только с выполнением программы. Большое значение имели исследования по разработке оптимизирующих компиля-

торов, эффективного использования регистров и механизмов эффективного выполнения программ. Хотя стоимость выполнения программы учитывается и в процессе разработки языка, но в первую очередь она важна для больших производственных программ, которые многократно выполняются. Однако на сегодняшний день для большинства приложений вопрос скорости выполнения не является первостепенным. Если речь идет об улучшении диагностики или упрощении контроля в процессе разработки и сопровождения программы, то при использовании настольных компьютеров, выполняющих много миллионов операций в секунду и большую часть времени находящихся в режиме ожидания, увеличение времени выполнения на 10 или 20 % является допустимым.

Стоимость трансляции программы. Когда такие языки, как FORTRAN или C используются в процессе обучения, первостепенным может оказаться вопрос эффективной трансляции (компиляции), а не эффективного выполнения. Как правило, в процессе отладки студенческие программы многократно транслируются, а выполняются всего несколько раз.

В этом случае важнее иметь быстрый и эффективный компилятор, а не компилятор, создающий эффективный код.

Стоимость создания, тестирования и использования программы. Этот аспект стоимости может быть проиллюстрирован на примере языков Smalltalk и Perl. Для определенного класса задач решение может быть разработано, закодировано, протестировано, изменено и использовано с минимальными затратами времени и сил программиста. Smalltalk и Perl являются примерами эффективных в смысле стоимости языков программирования. И это объясняется тем, что в них минимизировано общее время и объем усилий, требующихся программисту на решение на компьютере какой-либо задачи, даже если время выполнения программы может быть больше, чем для других языков.

Стоимость сопровождения программы. Многочисленные исследования показали, что самую большую часть стоимости программы, используемой в течение нескольких лет, составляет не стоимость начального создания, кодирования и тестирования программы, а стоимость полного жизненного цикла программы, куда входит стоимость как разработки, так и сопровождения программы. Поддержка включает в себя и исправление ошибок, выявленных уже после того, как программа отдана в эксплуатацию, и изменения, которые необходимо внести в программу в связи с обновлением аппаратной части или операционной системы, и усовершенствование и расширение возможностей программы для удовлетворения новых потребностей. Язык, который позволяет без особых проблем вносить многочисленные изменения и исправления в программу и создавать различные расширения (причем разными программистами и в течение многих лет), окажется в конечном счете более выгодным, чем любой другой.

5. Элементарные типы данных

Любая программа, в сущности, представляет собой набор операций, которые применяются к определенным данным в определенной последовательности. Основные различия между языками программирования сводятся к тому, каковы допустимые типы данных и операций и какие механизмы используются для управления последовательностью применения операций к данным. Эти три области — данные, операции и механизмы управления — составляют основу для сравнения языков программирования

Области хранения данных в аппаратной части компьютера (память, регистры и внешние запоминающие устройства) обычно имеют довольно простую структуру в виде последовательности битов, сгруппированных в байты или слова. Однако хранение данных в виртуальном компьютере (который представляет себе программист при написании программы), как правило, организовано более сложным образом — в различные моменты выполнения программы используются такие формы хранения данных, как стеки, массивы, числа, символьные строки и некоторые другие. Один или несколько однотипных элементов данных, объединенных в одно целое в виртуальном компьютере в некоторый момент выполнения программы, принято называть объектом данных. В течение времени выполнения программы существует множество объектов данных различных типов. Более того, в отличие от сравнительно неизменной статической организации хранения данных в аппаратной части компьютера объекты данных и отношения между ними динамически меняются в процессе выполнения программы.

Некоторые объекты данных определяются программистом (например, переменные, константы, массивы, файлы и т. д.); программист создает эти объекты и управляет ими явным образом при помощи объявлений и операторов в программе.

Другие объекты данных определяются системой (то есть автоматически создаются по мере необходимости виртуальным компьютером во время работы программы как вспомогательные, предназначенные для выполнения служебных функций).

К таким объектам данных (например, к стекам, записям активации подпрограмм, файловым буферам и спискам свободной памяти) у программиста нет непосредственного доступа.

Объект данных представляет собой контейнер для хранения значений данных – то есть то место, где эти значения хранятся и откуда они затем извлекаются. Объект данных характеризуется набором атрибутов, самым важным из которых является тип данных. Атрибуты определяют количество и тип значений, которые могут содержаться в объекте данных, а также определяют логическую организацию этих значений.

Значение данных может представлять собой отдельное число, символ или указатель на другой объект данных. Значение обычно представлено конкретной комбинацией битов в памяти компьютера. До сих пор мы предполагали, что два значения совпадают, если представляющие их в памяти компьютера комбинации битов идентичны. Но для более сложных типов данных такого простого определения недостаточно. Различие между объектами и

значениями данных во многих языках недостаточно отчетливо и их довольно легко перепутать. Пожалуй, наиболее ярко это различие проявляется в реализации: объект данных представлен некоторой областью в памяти компьютера, а значение данных представлено комбинацией битов. Если мы говорим, что объект данных А содержит значение В, то имеется в виду следующее: в области памяти компьютера, представляющей объект А, расположена определенная комбинация битов, соответствующая значению В.

Если мы пронаблюдаем за работой программы, мы заметим, что одни объекты данных существуют с самого начала выполнения программы, а другие создаются динамически в процессе ее выполнения. Одни объекты данных в какой-то момент уничтожаются, а другие сохраняются до конца работы программы. Таким образом, у каждого объекта данных имеется определенное время жизни, в течение которого он может быть использован для хранения значений данных. Объект данных называется элементарным, если содержащееся в нем значение всегда фигурирует в программе как единое целое. Если же этот объект представляет собой совокупность некоторых других объектов, то он называется структурой данных.

Любой объект данных за время своей жизни участвует в нескольких связываниях. Хотя атрибуты объекта данных инвариантны в течение его времени жизни, связывания могут динамически изменяться. Ниже перечислены наиболее важные атрибуты и связывания.

1. Тип. Этот атрибут ассоциирует объект данных с множеством значений, которые могут содержаться в этом объекте данных.

2. Местоположение. Связывание объекта данных с определенным местоположением (областью памяти) обычно задается и контролируется специальными вспомогательными программами управления памятью, которые предусмотрены в виртуальном компьютере и недоступны непосредственно для программиста.

3. Значение. Это связывание обычно является результатом операции присваивания.

4. Имя. Связывание объекта данных с одним или несколькими именами, по которым к нему происходят обращения во время работы программы, обычно осуществляется при помощи объявлений и может модифицироваться при входе и выходе из подпрограмм.

5. Компонент. Связывание объекта данных с одним или более объектами данных, компонентом которых он является, часто представлено значением указателя и может быть изменено посредством изменения этого указателя.

5.1 Переменные и константы

Переменная — это объект данных, который явным образом определен и назван программистом. Простая переменная — это элементарный объект данных, имеющий имя. Обычно мы представляем себе переменную как объект, значение (или значения) которого можно менять посредством операции

присваивания (то есть связывание переменной с ее значением может изменяться несколько раз за время жизни переменной).

Константа — это объект данных, имя которого неизменно связано со значением (или значениями) в течение всего времени жизни. Буквальная константа (или литерал) — это константа, имя которой является просто формой записи ее значения (например, «21» — это десятичная форма записи буквальной константы, которая представляет собой объект данных со значением 21).

Определяемая программистом, или именованная, константа — это объект данных, имя для которого программист выбирает произвольным образом при его определении.

Пример. Простые переменные в языке С

Подпрограмма на языке С может содержать следующее объявление:

int N;

что означает объявление простого объекта данных N целого типа (integer). Далее в подпрограмме может встретиться операция присваивания

N = 27;

используемая для присвоения значения 27 объекту данных с именем N. Более полно эту ситуацию можно описать так.

1. Посредством объявления создается элементарный объект данных целочисленного типа.

2. Этот объект данных должен быть создан при входе в подпрограмму и уничтожен при выходе из нее; таким образом, время его жизни равно времени выполнения подпрограммы.

3. В течение времени своей жизни этот объект данных связан с именем N, по которому к нему можно обращаться в подпрограмме, как, например, в приведенной выше операции присваивания. С ним могут быть связаны другие имена, если он передается как параметр в другую подпрограмму.

4. Изначально с объектом данных не связано никакое значение, но оператор присваивания временно связывает этот объект данных со значением 27, пока последующий оператор присваивания значения переменной N не изменит текущее связывание.

5. От программиста остаются скрытыми другие связывания, которые происходят в виртуальном компьютере: объект данных N может стать компонентом записи активации, то есть объекта данных, в котором содержатся все локальные данные для подпрограммы, а под эту запись активации отводится определенное место в создаваемом в момент начала выполнения подпрограммы стеке (еще один скрытый от программиста объект). Когда подпрограмма завершается, выделенная под стек область памяти освобождается для дальнейшего использования и связывание объекта данных с областью памяти разрушается.

Поскольку в случае с константой связывание значения с ее именем остается неизменным в течение всего времени жизни этой константы, информация об этом связывании доступна транслятору. Следовательно, если программист создает программу на языке С и пишет: #define MAX 30, то эта ин-

формация известна уже во время трансляции. Компилятор языка С может извлечь из такой информации определенную пользу — например, если в программе встретится оператор присваивания MAX = 4, это будет воспринято как ошибка, поскольку значение константы не должно меняться, присваивание константе MAX значения 4 имеет столько же смысла, что и оператор присваивания 30 = 4. Иногда компилятор может использовать информацию о значении константы для того, чтобы избежать генерирования кода для некоторого выражения или оператора. Например, в условном операторе

if (MAX < 2) {...} уже содержится информация для транслятора о реальном значении именованной константы MAX и буквальной константы 2, следовательно, транслятор может вычислить, что булево выражение MAX < 2 ложно, и проигнорировать полностью весь код для этого условного оператора if.

Пример. Переменные, константы и литералы в языке С.

В подпрограмму на языке С могут входить следующие объявления:

```
const int MAX=30;
int N;
```

Затем мы можем написать следующие операторы присваивания:

```
N = 27;
N = N + MAX;
```

N — это простая переменная, а MAX, 27 и 30 — константы. N, MAX, 27 и 30 — это имена объектов данных целого типа. Объявление константы определяет, что во время выполнения подпрограммы объект данных, названный MAX, постоянно должен быть связан со значением 30. Константа MAX— это определенная программистом константа, так как программист явным образом определил имя для значения 30. С другой стороны, имя 27 является литералом, который именует объект данных, содержащий значение 27. Такие литералы являются частью определения самого языка программирования. Важно понимать тонкое отличие между значением 27, которое является целым числом, представленным в памяти компьютера во время выполнения программы последовательностью битов, и именем «27», которое состоит из двух символов, 2 и 7, и является десятичной формой записи в тексте программы того же числа. В языке С предусмотрены как объявления констант, подобные приведенным в этом примере, так и макроопределения типа #define MAX 30, которое представляет собой операцию, выполняемую во время компиляции и приводящую к тому, что все ссылки на имя MAX в подпрограмме будут заменены на константу 30.

Обратите внимание на то, что в приведенном примере у константы 30 имеются два имени: определенное программистом имя MAX и буквальное имя 30; оба этих имени могут быть использованы в программе для ссылки на один и тот же объект данных со значением 30.

Следует учитывать, что использование

```
#define MAX 30
```

в C менее предпочтительно, поскольку может приводить к нежелательным побочным эффектам, поскольку заменяет все контекстные вхождения MAX ниже в файле, где она определена.

5.2. Типы данных

Тип данных — это некоторый класс объектов данных вместе с набором операций для создания и работы с ними. Если программа имеет дело с какими-то конкретными объектами данных (например, массив A, целочисленная переменная X или файл F), язык программирования, как правило, по необходимости имеет дело с типами данных – классами массивов, целых чисел или файлов и операциями, позволяющими с ними работать.

В каждом языке имеется некоторый набор встроенных примитивных типов данных. Дополнительно в языке могут быть предусмотрены средства, позволяющие программисту определять новые типы данных. Одно из главных различий между ранними языками программирования, такими как FORTRAN или COBOL, и более поздними, такими как Java и Ada, лежит в области определяемых программистом типов данных. Современный подход к проблеме состоит в том, что в языке должны присутствовать средства для манипулирования типами данных.

Основные элементы спецификации типа данных следующие:

1) атрибуты, которые характеризуют объекты данных заданного типа;

2) значения, которые могут принимать объекты заданного типа;

3) операции, которые определяют возможные манипуляции над объектами данных заданного типа.

Например, в случае спецификации для типа данных «массив» атрибуты могут включать количество размерностей массива, допустимый диапазон изменения индекса для каждой размерности и тип данных элементов массива. Значения могут быть представлены множествами чисел, которые определяют допустимые значения элементов массива, а операции могут включать индексацию элементов массива (каждому элементу соответствует определенный индекс). Возможны также операции для создания массивов, изменения их формы, доступа к некоторым атрибутам (например, верхней и нижней границе диапазона изменения индексов) и выполнения арифметических действий над парами массивов. Ниже перечислены основные элементы реализации типа данных.

1. Способ представления объектов данных этого типа в памяти компьютера в процессе выполнения программы.

2. Способ представления операций, определенных для этого типа данных, через конкретные алгоритмы и процедуры, которые используются для манипуляций с выбранной формой представления объектов данных в памяти.

Реализация типа данных определяет, каким должно быть программное моделирование соответствующих частей виртуального компьютера в терминах более простых базовых конструкций, предоставляемых соответствующим слоем виртуального компьютера. Последний, в свою очередь, может быть представлен либо непосредственно аппаратными средствами, либо ком-

бинацией аппаратных и программных средств, определенных операционной системой или микрокодом.

Последнее, что требуется определить при описании типа данных, — это его синтаксическое представление. И спецификация, и реализация мало зависят от конкретных синтаксических форм, используемых в данном языке. Атрибуты объектов данных обычно представлены синтаксически через объявления или определения типов. Значения могут представляться литералами или именованными константами. Вызов операций может происходить при помощи специальных символов, встроенных процедур или функций, таких как sin или read, или неявным образом через комбинации других элементов языка. Конкретный вид синтаксического представления не имеет большого значения, но синтаксическая информация может быть использована компилятором для определения времени связывания различных атрибутов и, следовательно, позволяет транслятору создать наиболее эффективное представление данных в памяти или выполнить проверку данных на соответствие указанному типу.

5.3 Спецификация элементарных типов данных

Элементарный объект данных может содержать только одно значение. Класс таких объектов, для которых определены различные операции, называется элементарным типом данных. Хотя в каждом языке программирования, как правило, присутствует свой, отличный от других набор элементарных типов данных и хотя конкретный вид их спецификации в различных языках может существенно различаться, тем не менее такие типы, как вещественные и целые числа, булевы значения, перечисления и указатели присутствуют почти во всех языках.

Тип объекта данных определяет множество допустимых значений, которые могут содержаться в этом объекте. Например, целочисленный тип определяет всевозможные целые числа, которые могут служить значениями объектов данных этого типа. В языке C определены четыре класса целых чисел: int, short, long и char. Поскольку в большинстве компьютеров арифметические операции на аппаратном уровне реализованы несколькими способами с различной точностью (например, 16-битные и 32-битные целые числа или 32-битные и 64-битные целые числа), в языке C программист может выбирать между этими различными аппаратными представлениями. Класс short соответствует наиболее короткому представлению числа, класс long — наиболее длинному аппаратному представлению, а класс int использует наиболее эффективный аппаратный способ представления из возможных. Этот класс может совпадать с классом short или long, а может соответствовать некоторому промежуточному представлению числа. Интересно отметить, что в C символы хранятся как 8-битные целые числа типа char, который является подтипом целочисленного типа.

Множество значений, допустимых для элементарных объектов данных, обычно является упорядоченным множеством с наименьшим и наибольшим

элементами. В каждой паре различных значений (элементов упорядоченного множества) одно всегда больше другого. Например, для целочисленного типа данных имеются наибольшее значение, которое может быть представлено в памяти компьютера, наименьшее значение и все промежуточные целые числа в их обычной последовательности.

Операции. Набор операций, определенных для какого-то типа данных, задает возможные манипуляции (действия) с объектами этого типа. Эти операции могут быть элементарными, то есть являться частью определения языка, или они могут определяться программистом в виде подпрограмм или методов как часть определения класса.

Операция, у которой имеются два аргумента и один результат, называется бинарной. Если у операции имеются один аргумент и один результат, то она называется унарной. Количество аргументов операции часто называется арностью операции. Большая часть примитивных операций в языках программирования бинарные или унарные.

Для точной спецификации действия операции обычно требуется подробная информация, чем сигнатура. В частности, способ представления аргументов (точнее, типов данных, к которым относятся аргументы) в памяти компьютера обычно определяет, какие действия можно осуществлять с этими аргументами. Например, алгоритм перемножения двух чисел, представленных двоичным кодом, отличается от алгоритма перемножения десятичных чисел. Поэтому обычно в спецификации присутствует некоторое неформальное описание действия операции.

Точная спецификация действия операции является частью реализации этой операции вместе с определением способа представления аргументов в памяти компьютера.

Иногда бывает непросто представить точную спецификацию операции в виде математической функции. Существуют четыре основных фактора, затрудняющих определение многих операций в языках программирования.

1. Невозможность определить действие операции для некоторых аргументов.

Операция, определенная на некоторой области, может фактически быть не определенной для некоторых элементов этой области (например, операция извлечения квадратного корня не определена для отрицательных целых чисел). Иногда чрезвычайно трудно точно указать область, для которой не определена данная операции. В качестве примера можно привести множества чисел, которые вызывают переполнение или исчезновение значащих разрядов в арифметических операциях.

2. Неявные аргументы.

Обычно при обращении к операции в программе ей передается некоторый набор явных аргументов. Однако в операции могут быть задействованы неявные аргументы — например, глобальные переменные или ссылки на другие нелокальные идентификаторы. Таким образом, полное выявление всех данных, которые влияют на результат операции, значительно усложняется наличием таких неявных аргументов.

3. Побочные эффекты (неявные результаты).

Операция может не только выдавать явный результат (например, сумму чисел как результат операции сложения), но и модифицировать каким-то образом значения, хранящиеся в других объектах данных, как определяемых программистом, так и определяемых системой. Такие неявные результаты называются побочными эффектами. Функция может не только возвращать некоторое значение, но и изменять значения переданных ей параметров. Побочные эффекты являются основной частью многих операций, в частности таких, которые модифицируют структуры данных. Наличие побочных эффектов затрудняет точное определение области значения операции.

4. **Самоизменение (зависимость от предыстории).**

Операция может изменять свою внутреннюю структуру — как локальные данные, которые сохраняются в промежутках между выполнением операции, так и свой собственный код. В таком случае для некоторого набора входных данных (аргументов) результат действия операции будет зависеть не только от этих аргументов, но и от всей предыстории обращений к этой операции в течение работы программы и от переданных ей ранее аргументов. Такая операция называется зависимой от предыстории. Самый очевидный пример подобной операции – это генератор случайных чисел, который определен в качестве операции во многих языках. Обычно аргументом этой операции является некоторая константа, но результаты повторного выполнения операции различны. Действие операции заключается в том, что помимо генерации явного результата она также изменяет внутреннее начальное число (называемое также затравкой), которое влияет на результат последующего выполнения операции. Самоизменение посредством модификации локальных данных встречается достаточно часто; изменение собственного кода операции встречается реже, но все же возможно в таких языках, как LISP.

Подтипы. При описании нового типа данных часто возникает желание указать на его сходство с некоторым другим типом. Например, в языке C определены типы int, short, long и char, которые являются разновидностями целочисленного типа.

Данные этих типов ведут себя сходным образом, и некоторые операции, в частности + и *, было бы логично определить аналогичным образом для всех этих типов.

Если некоторый тип данных является частью более крупного класса, то он называется подтипом этого класса, а сам класс называется супертипом для исходного типа данных. Например, в языке Pascal можно создать подкласс целых чисел следующим образом:

type Small Integer – 1..20;

Этот подкласс состоит из целых чисел от 1 до 20 включительно.

Предполагается, что все операции, определенные для супертипа, допустимы также и для подтипа. Например, можно сказать, что тип char в C наследует операции, определенные для целого типа int.

Объявления

При написании программы программист определяет имя и тип каждого объекта данных, который встречается в программе. Также должны быть определены время жизни каждого объекта данных, то есть часть программы, в которой этот объект используется, и те операции, которые применяются к этому объекту. Объявление — это оператор программы, назначение которого заключается в том, чтобы сообщить транслятору информацию об именах и типах объектов данных, используемых в программе. Размещение объявлений в программе (например, внутри подпрограммы или определении класса) задает время жизни объектов.

Например, в языке C объявление float A, B; в начале подпрограммы указывает, что во время выполнения этой подпрограммы потребуются два объекта данных типа float. Также это объявление связывает объекты данных с именами A и B на период времени их жизни. Приведенное выше объявление языка C называется явным, объявлением. Во многих языках используются также неявные объявления, или объявления по умолчанию. Они вступают в силу только в том случае, если отсутствуют явные объявления. Например, в подпрограмме на языке FORTRAN можно использовать простую переменную с именем INDEX, не объявляя ее явным образом, – по умолчанию компилятор предполагает, что эта переменная имеет целочисленный тип, так как ее имя начинается с буквы из диапазона I-N. В языке Perl объявление переменной происходит, когда ей присваивается некоторое значение:

$abc = 'a string'; # $abc - строковая переменная

$abc = 7; #$ теперь abc - целочисленная переменная

В объявлении также можно задать значение объекта данных, если это константа, или начальное значение объекта данных, если он не является константой. В объявлении также могут быть указаны другие виды связываний для данного объекта: имя объекта или его размещение как компонента внутри другого, более крупного, объекта данных. Иногда в объявлении указываются и некоторые детали, относящиеся к реализации объекта: связывание с конкретным местоположением в памяти или с конкретной формой представления объекта. Например, используемый в языке COBOL модификатор COMPUTATIONAL для целых чисел обычно указывает на необходимость использования двоичного (а не символьного) представления значений этих объектов в памяти, что позволяет использовать более эффективные арифметические операции.

Информация, которая в процессе трансляции требуется в первую очередь, — это сигнатура каждой операции. Обычно для встроенных в язык, то есть элементарных, операций явное объявление типов аргументов и результата не требуется. Эти операции можно вызывать по мере надобности, а типы аргументов и результатов для них по умолчанию определяются транслятором языка. Но для определяемых программистом операций обычно следует сообщить транслятору о типах аргументов и результатов перед тем, как обращаться к подпрограмме. Например, в языке C эта информация помещается в заголовочную часть определения подпрограммы, ее прототип. Так, следую-

щий прототип float Sub(int X, float Y) объявляет подпрограмму Sub со следующей сигнатурой:

Sub: int X, float Y —> float

Функции объявлений

Объявления выполняют несколько важных функций.

1. **Выбор способа представления в памяти**. Если объявление содержит сведения о типе данных и атрибутах объекта данных, то на основе этой информации транслятор часто может оптимизировать представление этого объекта данных в памяти.

2. **Улучшение управления памятью**. Содержащаяся в объявлении информация о времени жизни объектов данных зачастую позволяет использовать более эффективные процедуры управления памятью во время выполнения программы. Например, в языке С все объекты, объявленные в начале подпрограммы, имеют одинаковое время жизни (равное времени выполнения подпрограммы) и, следовательно, могут быть расположены в едином блоке памяти, который создается при входе в подпрограмму и уничтожается при выходе из нее. Другие объекты данных в С создаются динамически посредством специальной функции malloc. Поскольку для таких объектов время жизни не указано, каждому такому объекту место в памяти отводится индивидуально.

3. **Полиморфные операции**. В большинстве языков используются специальные символы для обозначения некоторых основных операций; так, например, символ «+» может обозначать любую из нескольких операций сложения в зависимости от типа переданных аргументов. Например, в С запись A + B может обозначать «выполнить целочисленное сложение», если A и B принадлежат целочисленному типу, или «выполнить вещественное сложение», если A и B — вещественные числа. В этом случае говорят, что символ операции является перегруженным, так как он обозначает не одну конкретную операцию, а скорее некоторую общую операцию сложения (определяющую группу родственных операций), которая может иметь разные формы для аргументов разных типов. В большинстве языков основные символы операций, такие как +, * и /, являются перегруженными (то есть они обозначают общие операции); другие имена операций определяют операции однозначным образом. Но, например, в языке Ada программист имеет возможность определять перегружаемые имена подпрограмм и добавлять дополнительный смысл существующим символам операций. В языке С++ концепция полиморфизма получила полную реализацию, в которой одно и то же имя функции может быть использовано для многих ее реализаций, зависящих от типа передаваемых аргументов. Если говорить об объектно-ориентированных языках, то полиморфизм является главным свойством, которое позволяет программисту расширить язык программирования с помощью новых типов данных и операций.

Объявления обычно позволяют компилятору языка определить во время компиляции, какая именно из группы родственных операций, обозначенных перегруженным символом, подразумевается в каждом конкретном случае.

Например, в языке C на основе объявлений переменных A и B компилятор определяет, какая из двух возможных операций (целочисленное или вещественное сложение) подразумевается в записи A + B. В таком случае не требуется проверки соответствия типов при выполнении программы. В языке Smalltalk, напротив, требуется определять конкретный вид операции сложения каждый раз, когда при выполнении программы встречается символ «+», поскольку в этом языке отсутствуют объявления типов для переменных.

4. **Контроль типов**. С точки зрения программиста, наиболее ценным свойством объявлений является то, что они позволяют производить статический, а не динамический контроль типов.

5.4. Контроль типов и преобразование типов

Способы представления данных, поддерживаемых аппаратной частью компьютера, обычно не предусматривают включение информации о типе данных, а при выполнении элементарных операций над такими данными контроль типов не осуществляется. Например, некоторое слово в памяти компьютера во время выполнения программы может содержать последовательность бит. Эта последовательность может представлять собой целое число, вещественное число, цепочку литер или инструкцию процессора — не существует способа определить, что именно. Элементарная операция целочисленного сложения, встроенная в аппаратную часть компьютера, не может проверить, являются ли переданные ей два аргумента целыми числами, — для нее это просто последовательности битов. Таким образом, на уровне аппаратуры обычные компьютеры весьма ненадежны в плане обнаружения ошибок в типах данных.

Контроль типов означает проверку того, что каждая операция, выполняемая в программе, получает правильное количество аргументов правильного типа.

Например, перед выполнением оператора присваивания

X = A + B * C

компилятор должен проверить, что каждая операция (умножение, сложение и присваивание) получает по два аргумента правильного типа. Если операция сложения + определена только для целых или вещественных чисел, а A принадлежит к символьному типу данных, то произойдет ошибка в типе аргумента. Контроль типов может происходить во время выполнения программы (динамический контроль типов) или во время компиляции (статический контроль типов). Значительное преимущество использования в программировании языков высокого уровня объясняется тем, что в реализации таких языков предусмотрен контроль типов для всех (или почти всех) операций и таким образом программист оказывается защищен от этого весьма трудноуловимого вида ошибок.

Динамический контроль типов для аргументов некоторой операции осуществляется во время работы программы непосредственно перед выполнением данной операции. Реализация динамического контроля обычно осно-

вана на использовании дескриптора типа, который хранится вместе с объектом данных и указывает его тип. Например, в целочисленном объекте данных содержится и его значение, и дескриптор, указывающий па принадлежность объекта к целочисленному типу.

Тогда каждая операция реализуется таким образом, чтобы в первую очередь проверялся дескриптор типа каждого аргумента. Операция выполняется, только если типы аргументов правильны; в противном случае генерируется сообщение об ошибке. После того как получен результат операции, к нему тоже должен быть присоединен соответствующий дескриптор, описывающий его тип, чтобы операции, в которых впоследствии этот объект может участвовать в качестве аргумента, могли бы проверить его тип.

Некоторые языки программирования, например Perl и Prolog, разработаны таким образом, что для них необходим динамический контроль типов. В этих языках не используется явное объявление типов переменных и отсутствуют встроенные (неявные) объявления типов (в отличие от языка FORTRAN, в котором переменные объявляются по умолчанию). Тип переменных A и B в выражении A + B может меняться в процессе выполнения программы. В такой ситуации динамический контроль типов этих переменных необходим для каждой операции сложения, происходящей во время выполнения программы. В таких языках, где отсутствуют объявления, переменные называются не имеющими типа, или бестиповыми, так как у них действительно нет никакого фиксированного типа.

Основным преимуществом динамического определения типов является гибкость программы: объявления типов отсутствуют, и тип объекта данных, связанного с некоторым именем, может меняться по мере необходимости в процессе выполнения программы. Программист, таким образом, лишается большей части забот, связанных с типами данных. Но, с другой стороны, динамический контроль типов имеет несколько существенных недостатков.

1. Затрудняется отладка программы (то есть становится трудно удалить все ошибки, связанные с типами аргументов). Поскольку динамический контроль происходит во время выполнения операции, для некоторых операций типы аргументов могут остаться непроверенными (такая ситуация возникает, если для конкретных входных данных эти операции относятся к невыполняемым ветвям программы). Вообще говоря, в процессе тестирования программы невозможно проверить все ее ветви. Все непроверенные ветви могут содержать ошибки, связанные с типами аргументов.

2. Динамический контроль типов подразумевает, что вся информация о типах должна сохранятся на всем протяжении работы программы, что требует дополнительных объемов памяти, иногда весьма существенных.

3. Динамический контроль типов должен быть, как правило, реализован посредством программного моделирования, поскольку аппаратная часть компьютера чаще всего не обеспечивает таких возможностей. Это приводит к замедлению выполнения программы.

В большинстве языков предприняты попытки свести на нет или минимизировать динамический контроль типов путем замены его на статический

контроль, то есть контроль во время компиляции. Необходимая для статического контроля информация частично поступает из объявлений, которые явным образом создает программист, а частично – из других языковых структур. Необходимая для реализации статического контроля типов информация должна включать в себя следующее.

1. Для каждой операции должны быть указаны количество, порядок и тип данных для аргументов и результатов (иначе говоря, сигнатура операции).

2. Для каждой переменной, представляющей собой имя объекта данных, следует указать тип этого объекта. Тип объекта данных не должен изменяться в процессе выполнения программы (он должен быть инвариантом). Однако при проверке типов для выражения A + B можно предположить, что тип данных для объекта под именем A остается неизменным при каждом вычислении выражения, даже если оно вычисляется в цикле и на каждом его шаге переменная A связывается с некоторым новым объектом данных.

3. Должны быть указаны типы всех констант. Синтаксическая форма записи литерала обычно дает информацию о его типе (например, 2 — это целое число, а 2.3 — вещественное). Для проверки типа именованной константы следует использовать ее определение.

На начальном этапе трансляции программ транслятор собирает информацию о типах объектов данных из объявлений, имеющихся в программе, и вносит ее в различные таблицы, главным образом в таблицу символов, которая содержит сведения о типах переменных и операций.

После того как собрана вся эта информация, происходит проверка всех вычисляемых в программе операций на правильность типов их аргументов. Заметим, что, как отмечалось ранее, для полиморфных операций любой из нескольких возможных типов аргументов будет воспринят как правильный. Если все аргументы для данной операции прошли проверку, то определяется тип результата и полученная информация сохраняется компилятором для проверки последующих операций.

Обратите внимание на то, что в случае полиморфной операции опять-таки общее (групповое) имя может быть заменено именем, специфическим для конкретного типа данных передаваемых ей аргументов.

Поскольку статический контроль типов осуществляется для всех без исключения операций, имеющихся в программе, то проверяются все возможные ветви программы и дальнейшем контроле не возникает необходимости. Таким образом, не требуется включать в объекты данных дескрипторы типов и производить динамический контроль типов. В результате получается значительный выигрыш в скорости выполнения программы и эффективности использования памяти.

Стремление реализовать статический контроль типов оказывает значительное влияние на многие аспекты языка: объявления, структуры управления данными, организацию раздельной компиляции подпрограмм и т. д. Во многих языках не всегда можно осуществить статический контроль типов для некоторых конструкций в определенных случаях. Подобные проблемы со

статическим контролем типов в некоторых языковых структурах можно решать двумя способами.

1. Отказаться от статического контроля в пользу динамического. Цена такого решения часто оказывается высокой в отношении используемой памяти, так как во время выполнения программы требуется хранить дескрипторы типов для всех объектов данных, хотя они и редко проверяются.

2. Вообще отказаться от контроля типов для операций. Не проверенные на соответствие типов аргументы операции могут вызвать серьезные и трудноуловимые ошибки в программах, но иногда приходится использовать этот вариант, если стоимость динамического контроля типов оказывается слишком высокой.

Сильная типизация. Если мы можем при помощи статического контроля обнаружить все без исключения ошибки определения типов и программе па данном языке, то такой язык называется сильно типизированным. Вообще говоря, строгий контроль типов является некоторой гарантией отсутствия соответствующих ошибок в программе. Мы называем функцию f с сигнатурой f : S —> R безопасной в отношении типа, если при вычислении этой функции результат не может выйти за пределы множества R. В случае всех операций, безопасных в отношении тина, мы знаем еще до выполнения программы, что тип результата будет правильным и что динамический контроль типов не требуется. Очевидно, если каждая операция безопасна в отношении типа, то язык в целом является сильно типизированным.

Однако немногие языки соответствуют этому требованию. Например, в языке С, если X и Y принадлежат типу short (то есть короткие целые числа), то результат операции X + У или X * Y может оказаться за пределами этого типа, вследствие чего возникнет ошибка типа. Хотя реализация в языке настоящего строгого контроля типов трудна, можно значительно приблизиться к ней, если ограничить преобразование одного типа в другой.

Выведение типа. В языке ML предложен интересный подход к определению типов данных. Суть его в том, что в случае, если возможна однозначная интерпретация типа данных, его объявление не требуется. Язык реализован таким образом, что недостающая информация о типах данных может быть выведена из других имеющихся объявлений типов. В этом языке для объявления аргументов функций используется достаточно стандартный синтаксис, а именно fun area(length:int, width: int):int = length * width.

В данном случае функция area возвращает целое число — величину площади прямоугольника, которая вычисляется как результат перемножения двух целочисленных аргументов, длины и ширины. В данном случае, если определен тип хотя бы одного из этих трех объектов данных: area, length или width, то можно вывести типы двух других. То есть, даже исключив описание типов каких-то двух параметров, мы все же получаем однозначную интерпретацию этой функции. Поскольку символ «*» может означать и целочисленное, и вещественное перемножение, каждая из приведенных далее форм записи в ML эквивалентна записи из предыдущего примера:

fun area(length, width):int = length * width;

fun area(length:int, width) = length * width;

fun area(length, width:int) = length * width;

Однако такая запись

fun area(length. width) = length * width;

уже ошибочна, так как в ней отсутствует однозначность определения типов данных. В такой записи все используемые объекты данных могут быть либо целыми, либо вещественными.

Заметим, что в последних версиях C++ и C# есть аналогичный механизм определения типа путём выведения, для этого в C# используется ключевое слово var, а в C++ – auto. Эти ключевые слова позволяют вывести тип результата выражения, например:

int a, b;

auto c=a+b;

определяет с как переменную типа int.

Приведение и преобразование типов

Если в процессе проверки типов выявляется несоответствие между фактическим и ожидаемым типами аргумента для операции, то может произойти одно из двух:

1) Несоответствие типов будет воспринято как ошибка, о чем появится сообщение, и будут предприняты предусмотренные для такого случая действия;

2) произойдет приведение (или неявное преобразование) типа фактически переданного аргумента к правильному, ожидаемому для данной операции.

Иначе говоря, преобразование типа заключается в том, что вместо объекта данных одного типа создается соответствующий ему новый объект другого типа. В большинстве языков преобразование типов осуществляется одним из следующих двух способов.

1. Набором встроенных функций, которые программист может вызвать явным образом для преобразования типа какого-либо объекта. Например, в языке Pascal имеется функция round, которая преобразует объект данных вещественного типа в целочисленный объект, значение которого равно округленному значению вещественного числа, В языке С мы принудительно приводим выражение к правильному типу. Например, операция (int) X преобразует вещественное X в объект целого типа.

2. Приведением типа, которое осуществляется автоматически в определенных случаях несоответствия типа. Например, в языке Pascal, если арифметической операции сложения переданы аргументы различных типов – вещественного и целочисленного, то по умолчанию целочисленный тип преобразуется в вещественный и затем осуществляется вещественное сложение. В основе неявного приведения типов лежит концепция недопущения потери информации. Имеется в виду следующее: поскольку в языке С любое короткое целочисленное значение может быть преобразовано в длинное, при автоматическом преобразовании short int —> long int не происходит потери информации. Такое приведение называется расширением. Аналогично, по-

скольку в большинстве языков целые числа могут быть однозначно выражены через вещественные объекты данных, короткие целые числа могут быть расширены до вещественных без какой-либо потери информации.

Но преобразование вещественного числа в целое, как правило, связано с потерей информации. Хотя 1,0 в точности равно 1, число 1,5 уже невозможно преобразовать в целое число. Оно будет преобразовано в 1 или 2, В данном случае приведение типа называется сужением и приводит к потере информации. При динамическом контроле типов приведение осуществляется в момент обнаружения несоответствия типов при выполнении программы. В таких языках сужения могут быть допустимы для некоторых значений объектов данных (например, 1,0 можно преобразовать к целому типу, но 1,5 нельзя). При статическом контроле типов для реализации таких сужений в программу при компиляции в нужных точках вставляется дополнительный код, который во время выполнения программы и осуществляет сужение. Но так как эффективность работы программы обычно является одним из основных требований, программисты стараются избегать применения сужений, чтобы не приходилось выполнять дополнительный код, определяющий их допустимость.

Операции преобразования типов могут потребовать значительных затрат компьютерного времени при выполнении программы. Например, в языках COBOL и PL/I числа часто хранятся в виде символьных строк. В большинстве машин операция сложения реализована таким образом, что эти символьные строки приходится преобразовывать в двоичный код, поддерживаемый аппаратной частью компьютера, а результат сложения снова преобразовывать в символьные строки для хранения в памяти. Затраты времени на преобразование типов могут превосходить затраты на само сложение в сотни раз.

Разработчики трансляторов языков тем не менее иногда смешивают семантику объектов данных и способы их представления в памяти. Примером могут служить десятичные числа в COBOL и PL/I. Трансляторы PL/I обычно хранят данные типа FIXED DECIMAL в упакованном десятичном формате. Это представление обеспечивается непосредственно аппаратной частью компьютера, но все же выполняется достаточно медленно. В компиляторе PL/C данные типа FIXED DECIMAL хранятся в виде 16-значных вещественных чисел двойной точности с плавающей точкой. При таком способе хранения не происходит потери точности (что особенно важно при храпении десятичных чисел). Например, вычисление суммы 123,45 + 543,21 приводит к довольно медленному сложению упакованных десятичных чисел (или еще более медленному программно-моделируемому сложению, если упакованные десятичные данные не поддерживаются непосредственно аппаратной частью компьютера). В компиляторе PL/C это же сложение осуществляется как единое действие, требующее гораздо меньших временных затрат: складываются два числа с плавающей точкой 112345 + 54321, а компилятор следит за положением десятичной точки (в данном случае результат сложения должен быть умножен на 10^2), являющейся атрибутом результата, определяемым во время компиляции.

Существует два противоположных мнения относительно «широты» применения неявного преобразования типов, В языках Pascal и Ada оно почти не используется; любое несоответствие типов, за небольшим исключением, воспринимается как ошибка. В C++ приведение типов является правилом — когда компилятор обнаруживает несоответствие типов, он пытается отыскать подходящую операцию преобразования, которую можно было бы вставить в компилируемый код для того, чтобы изменить нужным образом тип объекта данных. Сообщение об ошибке возникает только в том случае, когда компилятор не может произвести требуемое преобразование типов.

Несоответствие типов – это хотя и незначительная, но распространенная ошибка, и поэтому необходимость в преобразовании типов возникает достаточно часто, особенно в тех языках, в которых определено большое количество типов данных.

Приведение типов часто освобождает программиста от необходимости кропотливой работы, которая потребовалась бы для явного введения в программу операций преобразования типов. С другой стороны, приведение типов может скрыть наличие других, более серьезных ошибок, которые в противном случае могли бы быть замечены программистом при компиляции.

Язык PL/I, в частности, печально известен как язык, компиляторы которого имеют тенденцию так «исправлять» незначительные ошибки (например, неправильно написанное имя переменной) посредством сложных преобразований типа, что их становится очень трудно обнаружить. Поскольку в PL/I допускаются сужающие преобразования типов, иногда получаются удивительные результаты.

Например, 9 + 10/3 является недопустимым выражением! Чтобы понять это, заметим, что 10/3 преобразуется к виду 3.3333333... (цифра 3 повторяется столько раз, сколько допускается реализацией языка на данном компьютере). Но в результате операции сложения 9 + 3.33333... получается ошибка переполнения, так как в целой части результата добавляется еще одна цифра. Если эту ошибку проигнорировать, то результат автоматически преобразуется к виду 2.33333..., а это совсем не то, что ожидалось.