Введение

Развитие и совершенствование языков программирования (ЯП) и соответствующих инструментальных средств разработки представляет собой одно из основополагающих направлений современной информатики, технологии и практики индустриального программирования. Потребность в мощных, гибких и надежных языковых инструментах, адекватных предельному уровню сложности и ответственности задач, решаемых программным обеспечением (ПО), определяет высокую активность исследований и разработок в данной области.

В настоящее время сложились две основных тенденции, связанные с проектированием и реализацией ЯП.

1. Центр тяжести в этой сфере смещается от проектирования и освоения новых языков программирования в сторону строгой (насколько это возможно), исчерпывающей и недвусмысленной спецификации известных и зарекомендовавших себя ЯП. Это направление тесно связано с процессом международной **стандартизации** ЯП.

2. Все возрастающее внимание обращается как на эффективность и адекватность традиционного инструментария программирования - компиляторов, так и на качественное изменение задач, которые призваны решать современные системы (среды) разработки. Последнее выражается в существенном расширении сферы применения традиционных принципов, методов и технологий компиляции.

Рассмотрим эти направления подробнее.

Стандартизация ЯП

За последние пять лет в области языков программирования сложилась явная тенденция, которую можно охарактеризовать как процесс приведения распространенных ЯП промышленного назначения к унифицированным и международно признанным формам. Эта тенденция проявилась в активности международной организации (ISO - International Standard Organization) и авторитетных национальных институтов (прежде всего, ANSI - American National Standard Institute), воплотившейся в успешной стандартизации нескольких крупных ЯП. Ряд других ЯП находятся в настоящее время на различных стадиях процесса стандартизации.

Далее мы кратко рассмотрим наиболее интересные и существенные аспекты этой тенденции. Но сразу же необходимо отметить, что тенденция стандартизации имеет глубокие причины, которые, на наш взгляд, заключаются в следующем:

Сложность. Программно-управляемые системы промышленного, оборонного, научного и научно-технического назначения достигли некоего порога сложности, за которым степень адекватности языковых инструментов становится предельно критичной. Ходом технического

прогресса постиндустриальной эры создателям и реализаторам языков программирования был брошен вызов в виде потребности в таких средствах, которые были бы в состоянии справиться с масштабом, сложностью и ответственностью задач, решаемых современными системами. Такие языковые средства должны быть (по крайней мере, де-факто) признаны международным профессиональным сообществом.

Глобализация. Ситуация, когда выбор средств (и, в частности, языка программирования) для решения определенной задачи был в большой степени субъективен или определялся без учета возможных интеграционных аспектов, когда создаваемая система в течение всего жизненного цикла должна была функционировать практически неизменно и в достаточно изолированном окружении, в настоящее время все менее характерна для индустриального программирования. Степень интегрированности даже разнородных систем значительно повышается, их взаимодействие принимает регулярный и интенсивный характер, а вероятность модификации систем в связи с изменением задач в процессе жизненного цикла существенно возрастает. Типичным требованием в настоящее время становится идентичность поведения программы на самых разных платформах и в произвольном операционном окружении. Поэтому, наряду с унификацией интерфейсов и протоколов взаимодействия между системами на различных уровнях (от физических до пользовательских), задача выработки единых, строгих и международно признанных спецификаций на инструменты разработки систем становится одной из наиболее актуальных.

Промышленный характер. Переход от понимания программирования как творческого акта к трактовке процесса разработки программных систем как вида производственной деятельности, который был предметом многих профессиональных дискуссий, можно считать совершившимся, по крайней мере, в сфере создания и сопровождения систем промышленного назначения. Жесткие исходные требования, масштабные и ответственные задачи, большие коллективы разработчиков с высокой степенью разделения труда и большой объем затрат - характерные черты современного процесса создания ПО, которые роднят его с промышленным производством. Типичные же параметры любого крупного производства подразумевают строгие и недвусмысленные спецификации, в частности, на инструментарий. Поэтому корпоративные разработчики осознают важность стандартизации этого процесса, которая снижает издержки и риски и дает дополнительные гарантии успеха того или иного проекта.

Приведенные соображения не исчерпывают всех аспектов тенденции стандартизации ЯП. В частности, такие сферы, как профессиональное обучение, вопросы публикации архитектурных и алгоритмических решений также настоятельно требуют адекватного современным потребностям решения.

Годы, предшествовавшие последнему пятилетию, дали несколько примеров стандартизации, заслуживающих упоминания в контексте позднейшей тенденции. Наряду с выдающимся успехом, связанным с созданием и стандартизацией языка Ада [1], следует отметить

относительную неудачу стандартизации языков Pascal [7] и Modula-2 [9]. Так, по общему мнению язык Паскаль был стандартизован до того, как были осознаны требования к этому языку как к инструменту практического программирования. В результате в стандарт не вошли некоторые важные черты (в частности, модульность и гибкие средства работы с массивами), предложенные позднее и зарекомендовавшие себя с положительной стороны. Язык, определенный стандартом, получился недостаточно мощным и развитым и оказался практически непригоден для промышленного использования. Это привело к существенному обесценению принятого стандарта и не послужило преградой появлению многочисленных диалектов и расширений, например Pascal фирмы Borland, объектно-ориентированные версии языка (входной язык системы Delphi, Pascal Plus [89], Pascal++ [90]) и т.п. Этими причинами было вызвано принятие уже через год нового стандарта расширенного Паскаля [13], которое в целом не изменило ситуацию к лучшему.

Следующий этап этого процесса, начало которого можно условно датировать серединой 90-х гг., знаменует собой успешное завершение ряда крупных проектов, среди которых можно упомянуть стандартизацию новой редакции языка Ада [2] (далее - Ада95), языка LISP [10] и языка Си++ [11]. Завершается стандартизация новой редакции языка Си [12], начался процесс стандартизации языка Java [99].

Из упомянутых выше проектов наиболее интересно проследить историю возникновения, развитие и стандартизацию таких мощных и развитых общецелевых языков программирования, как Ада и Си++.

Процесс стандартизации Ада и Си++

Язык Си++ в настоящее время является, безусловно, одним из наиболее широко используемых общецелевых языков. Он вобрал в себя многие концепции и подходы к разработке ПО, соответствующие современному пониманию программирования как систематической деятельности по конструированию сложных программных систем. Здесь необходимо отметить прежде всего реализованный в Си++ объектно-ориентированный подход, предоставляющий мощные средства создания сложных схем обработки данных и взаимодействия и лежащий в основе методов разработки переиспользуемых (reusable) программных компонент.

Что касается Ада95, то это в настоящее время практически единственный универсальный язык программирования, сравнимый с Си++ как по широте возможностей, общецелевому характеру предполагаемого применения, так и по достаточно полному и строгому концептуальному базису. Си++ и Ada95 являются наиболее развитыми современными проектами языков программирования с объектно-ориентированной парадигмой.

Оба этих языка возникли не на пустом месте. Их основой являются, соответственно, язык Си и предыдущая (1983 года) редакция языка Ада (далее - Ада83), хорошо зарекомендовавшие себя в практическим

программировании. Как уже говорилось, оба языка-предшественника были стандартизованы как ANSI, так и ISO [14], [1]; стандарт языка Ада95 принят в 1995 году, стандарт Си++ - в 1998 г.

Между языками Си++ и Ада95 имеется ряд прямых аналогий. Помимо таких общих фундаментальных принципов, как строгая типизация и (с некоторыми оговорками для Си++) модульная структура, ключевые понятия обоих языков, прежде всего, полная реализация объектно-ориентированной парадигмы и механизм параметризуемых типов и алгоритмов (родовые модули Ада95 и шаблоны классов и функций Си++) семантически и функционально весьма близки.

Вместе с тем, характер проектирования и история развития этих языков существенно различны. Язык Си - предшественник Си++ - первоначально был создан как инструмент разработки новой операционной системы UNIX и предназначался для использования коллективом разработчиков в качестве системного языка реализации среднего уровня как альтернатива устаревшим к тому времени языкам типа BCPL [91]. Ничуть не умаляя достоинств Си, можно утверждать, что своим успехам он в значительной степени обязан выдающемуся успеху ОС UNIX. Стандартизация Си в 1989 году, скорее, была призвана зафиксировать сложившийся образ языка и остановить его "расползание" в виде многочисленных диалектов и фирменных расширений.

Язык Ада83, напротив, изначально проектировался как стандартный инструмент реализации широкого класса программных систем, причем конкретный вид языка определялся в ходе систематического процесса проектирования, начиная от выверенной концептуальной базы и совокупности исходных требований [71] до структуры языка и синтаксиса и семантики его конкретных конструкций.

Аналогичная ситуация сложилась с созданием Си++ и Ада95. Си++ возник в той же организации, что и Си и по существу в том же коллективе разработчиков. Первоначально язык представлял собой попытку просто расширить Си понятиями объектно-ориентированного пргграммирования, как они сложились к началу 80-х годов. Название языка - "Си с классами" [44] - в точности отражало вполне скромные намерения авторов, что подтверждает и препроцессорная схема его реализации, предложенная в той же статье.

Последующие этапы развития языка [118], которые можно проследить по работам [47], [45], [114], а также начавшийся в 1990 году процесс его стандартизации [16-17], наглядно иллюстрируют хаотический и спонтанный характер превращения "Си с классами" в полноценный объектно-ориентированный язык общего назначения, в котором его создатели сремятся, с одной стороны, преодолеть ряд известных недостатков языка Си [77-79], сохранив в то же время совместимость с ним, с другой стороны, обеспечить приемлемую степень надежности, достаточную для использования Си++ в ответственных проектах, и, наконец, реализовать в языке как можно больше концепций и возможностей, возникших и утвердившихся в программировании к настоящму моменту. В результате типичной

становится ситуация, когда в некоторой версии предварительного стандарта (после длительного процесса обсуждения, завершающегося голосованием) вводится новая языковая возможность, в процессе последующих дискуссий выясняется, что описание этой возможности содержит множество неясностей, двусмысленностей, противоречий и конфликтов с другими свойствами. Следующая версия стандарта уже включает соответствующие исправления, уточнения, оговорки и исключения, связанные с этой языковой возможностью, и... другое нововведение. Процесс повторяется.

Несмотря на то, что процесс стандартизации языка начался еще в 1990 году, срок завершения этого процесса неоднократно переносился. В результате Международный Стандарт был принят только в 1998 году, более чем с четырехлетним опозданием. Чрезмерно затянутая процедура формирования и принятия Стандарта привела к тому, что язык вступил в конфликт со своими реализациями: за это время появилось значительное число компиляторов Си++ промышленного уровня, содержащих собственные интерпретации многочисленных непроясненных Стандартом мест языка. В результате окончательная версия Стандарта содержит большое число компромиссов между решениями различных фирм-производителей компиляторов, а также между рабочей группой WG21 ANSI/ISO и разработчиками компиляторов. Примером может служить использование настроек шаблонов в многомодульных программах (проблема раздельной компиляции шаблонов), описанная в разд. 4.8.

Заметим, что процесс стандартизации языка Си++, как таковой, включая историию принятия и отказа от тех или иных черт языка, содержание и характер обсуждений и дискуссий, отраженные в многочисленных документах рабочих групп комитетов ISO и ANSI, наконец, эволюцию языка в виде последовательных редакциях предварительного стандарта, по нашему мнению, заслуживают самого тщательного изучения. Такое исследование было бы очень полезно для понимания современных взглядов на программистскую деятельность, текущего уровня развития науки программирования, характера требований, предъявляемых к инструментальным языковым средствам со стороны прикладных областей, и, между прочим, для оценки уровня квалификации и глубины анализа, выказываемых мировыми специалистами в программировании.

В противоположность Си++, проектирование языка Ада95 носило в высшей степени тщательный, последовательный и систематический характер. Как и в случае предыдущей версии языка - Ада83,- были сформулированы причины, приведшие к необходимости развития языка (было решено именно развивать существующий язык, а не проектировать новый), и требования к его новой версии [72]. Новые возможности введены в язык очень аккуратно; как правило, они ортогональны уже имеющимся, не пересекаясь с ними. Уточнения и расширения существующих свойств языка производились только при наличии существенных потребностей. Полностью актуальным оставался принцип "Программа - прежде всего средство общения людей друг с другом и только после этого - средство общения человека с компьютером". Иными словами, Ада95, обогатившись новыми концепциями и возможностями и избавившись от некоторых недостатков своего предшественника, нисколько не потерял в читабельности и наглядности программ.

Помимо собственно стандарта языка и параллельно с ним был разработан целый ряд важных дополнительных документов как нормативного, так и рекомендательного характера, в том числе, обоснование языка [5], аннотированный стандарт [4], описание изменений языка по сравнению с предыдущей его редакцией [3], руководство по стилю программирования [6], порядок перехода к новой редакции [73] и т.д. Подробные сведения об истории создания языка Ада95 содержатся в работе [76].

В результате, Ада95, нисколько не проигрывая Си++ по набору возможностей, выглядит заметно стройнее, целостнее, надежнее и проще для изучения и использования. Предмет диссертационной работы не предполагает детального сравнения этих языков; содержательный сравнительный анализ Си++ и Ада95, проведенный как с позиций противников и сторонников одного из этих языков, так и нейтральные по своему характеру, можно найти в работах [80-87]. Здесь заметим только, что, несмотря на лучшую проработанность проекта Ада95, этот язык уступает в распространенности Си++. Причиной тому является множество обстоятельств, в том числе ряд устоявшихся предрассудков относительно языка Ада, большинство которых убедительно опровергается в [74].

Вообще говоря, вопрос о том, какой язык лучше, по нашему мнению, носит отвлеченный и схоластический характер. В подавляющем большинстве случаев выбор языка для реализации того или иного проекта практически полностью определяется внешними обстоятельствами, не зависящими от разработчиков и исключающими их произвол. К таким обстоятельствам следует, отнести, во-первых, существо решаемой задачи и требования к ее решению и во-вторых, весьма вероятное наличие отраслевых или фирменных стандартов и традиций, сложившихся в коллективе разработчиков.

Завершая данный обзор, отметим, что рассмотренные тенденции и результаты стандартизации не были бы возможны без осознанной потребности со стороны компьютерной индустрии в унификации инструментальных средств, используемых в промышленных разработках. Поэтому следование международным стандартам во всем их объеме становится обязательным требованием для проектировщиков и разработчиков компиляторов и систем программирования. Использование стандартных ЯП и проблемно-ориентированных интерфейсов, основанных на этих языках, считается одним из наиболее адекватных способов достижения мобильности и интероперабельности программ.

ЯП, определенные в виде международных стандартов, фактически являются своего рода островами стабильности в бурном море сменяющих друг друга программных технологий, парадигм и даже поколений аппаратных средств. Установленный ISO пятилетний срок пересмотра стандартов на ЯП предоставляет достаточную перспективу для использования таких стандартов. С другой стороны, анализ конкретных случаев пересмотра стандартов (например, Ada 83 [1] и Ada 95 [2], Си [14] и проект стандарта C9X [12]) показывает, что новые стандарты обеспечивают весьма тесную преемственность со своими предшественниками (вплоть до совместимости снизу вверх). Понятно, что такая совместимость увеличивает реальный "срок службы"

стандартного ЯП и придает дополнительную уверенность при выборе такого языка в качестве инструмента промышленного программирования.

Что же касается направления исследований, заявленных в названии диссертационной работы, и конкретной реализации Си++, лежащей в ее основе, то приведенные соображения одновременно являются аргументами для выбора Си++ как предмета исследований и объекта компиляции. Ниже в данном Введении мы рассмотрим ряд дополнительных аргументов в пользу выбора этого языка.

Новый взгляд на задачу компиляции

Долгое время основным инструментарием разработчика ПО служил традиционный компилятор некоторого языка программирования, который, воспринимая на входе текст программы (или ее части - модуля) на этом ЯП, генерировал, соответственно, либо готовую к исполнению программу, либо объектный код модуля, который подлежал комплексированию с другими модулями, образующими программу. Согласно такому подходу, компилятор представлялся в виде единой монолитной программы, выполняющей единственную ограниченную функцию - построение машинного кода, семантически эквивалентного исходной программе. Дерево программы, таблицы символов и прочие структуры, формируемые компилятором в процессе работы, рассматривались как сугубо внутренняя информация, необходимость в которой исчезала непосредственно после формирования результата компиляции. Принципиальная схема описанной технологии представлена на рис. 1.

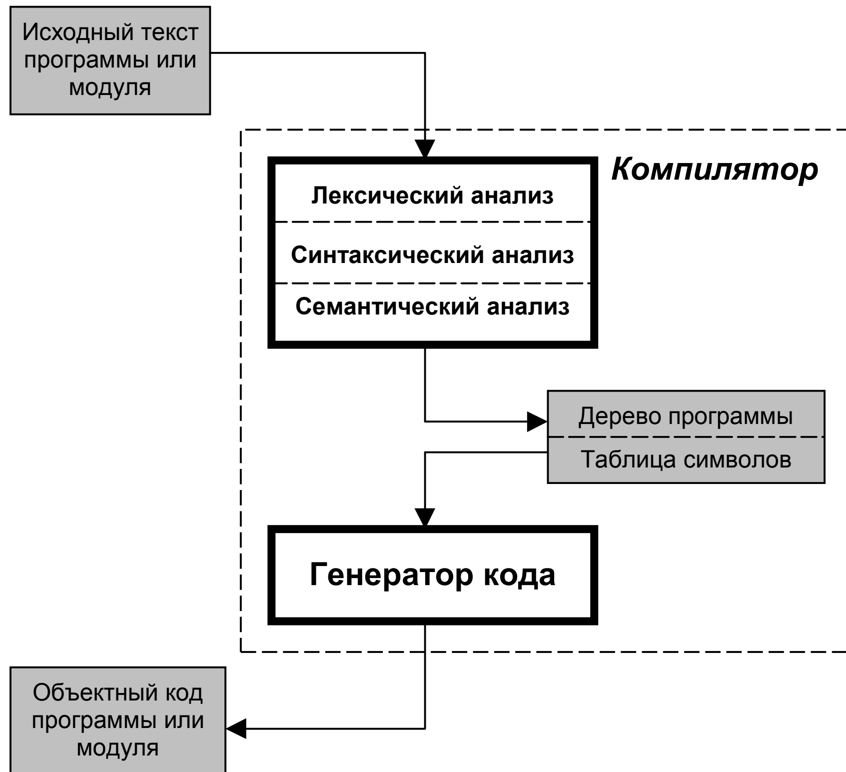

Рис.1 Традиционное понимание задачи компиляции

Возрастающая сложность программных систем, индустриализация процесса их создания, повышенные требования к их надежности и необходимость увеличения производительности труда разработчиков ПО привели к необходимости "некомпилирующих" системных средств дополнительного характера. Примером служат символьные отладчики, которые позволяли бы проводить отладку программ в терминах исходного ЯП. Требования расширения арсенала разработчиков привели к частичному переосмыслению описанной схемы. Теперь результирующий код, сформированный компилятором, трактуется не только как объект, подлежащий выполнению, но и как исходная информация для проведения ряда качественно иных работ.

Так, объектный код программы, дополненный информацией, связывающей его с исходным текстом, служит основой для отладочного исполнения программы с выдачей (в терминах исходного ЯП) разнообразной информации о процессе ее выполнения. Далее, компилятор может включить в объектный код дополнительную функциональность, направленную на снятие динамических характеристик ("профиля") программы (утилита gprof в системе GNU, Turbo Profiler фирмы Borland). Еще один характерный пример - так называемое обратная компиляция (reverse compilation, decompilation [31]) или "дизассемблирование" - восстановление программы по ее объектному коду в удобном для восприятия человеком формате (в частности, в виде текста на исходном ЯП).

Общим для перечисленных и других подобных дополнительных средств является то, что все они базируются на сгенерированном компилятором объектном коде, возможно, дополненным структурами, которые традиционно называются *отладочной информацией*. Это общее свойство является одновременно и принципиальным недостатком подобных средств. В распространенных форматах объектного кода, в частности, в COFF [108] и ELF [109] отладочная информация носит вторичный, дополнительный характер и содержит далеко не полное знание об исходном тексте программы. Специальные форматы для хранения отладочной информации, например, STABS [110], и в особенности DWARF [111], предоставляют более развитые средства, однако и их возможности ограничены. Характерными выглядят попытки преодолеть этот недостаток, например, в компиляторе GNAT для языка Ада95: объектный код, формируемый этим компилятором, содержит *полный исходный текст* единицы компиляции!

С другой стороны, легко видеть, что структуры, генерируемые компилятором в процессе работы, прежде всего, таблицы символов и дерево программы, по своему назначению как раз и призваны максимально адекватно соответствовать исходной программе. Более того, любой компилятор в процессе семантического анализа выявляет ряд скрытых (неявно присущих программе) свойств и отображает их в указанных структурах. Типичными примерами таких неявных свойств служат обработка совместно используемых операций Си++ [11, глава 13], семантика которых предполагает вызов специальных функций-операций, а также настройка шаблонов функций, в процессе которой может осуществляться глубокое перестроение их алгоритмов.

Далее, возрастающая сложность создаваемого ПО и, что самое главное, расширение задач, стоящих перед индустрией программирования, постепенно привели к изменению во взглядах на то, что в отечественной литературе традиционно называется системой программирования [19] или окружение программирования [36]. Более адекватным современному пониманию целей и задач СП является трактовка задачи компиляции в более широком смысле, нежели ранее - не только как генерацию кода, пригодного для исполнения, но и как широкий спектр операций над текстами программ.

Кратко рассмотрим наиболее типичные задачи, связанные с обработкой программных текстов.

1. **Компиляция** (в узком смысле - как получение исполняемого кода) остается основной функцией систем программирования. При этом особое значение придается таким задачам, как межъязыковое связывание (конструирование программ, компоненты которых написаны на различных ЯП, интероперабельность (динамическое взаимодействие программ) и в особенности межмодульная и глобальная оптимизация получаемого кода (например, [61], [37]). Мы вернемся к этим задачам немного далее.

2. Возникает целый комплекс задач, связанных с *пониманием* программ человеком. Несмотря на существенные усилия, предпринятые для повышения читабельности программ, анализ больших программных

текстов, приобретая очень важное значение в современной индустрии ПО, остается одним из самых трудоемких и тяжелых задач. В связи с этим за последние годы был предложен ряд методик, призванных отобразить различные характеристики программ (структуру, связи между подпрограммами и модулями, информационные потоки, потоки управления и т.п.) в форме, облегчающей их восприятие человеком. Такие методики (наиболее известной и развитой является нотация Г.Буча), как правило, являются обратимыми, то есть могут использоваться как для анализа существующих программ, так и для проектированиия нового ПО. С некоторой долей огрубления указанный комплекс задач можно назвать **визуализацией**. Помимо методологии и графической нотации UML, основанной на работах Г.Буча и реализованной для широко распространенных ЯП (см., например [30]), имеется ряд менее масштабных, но весьма практичных разработок, предлагающих удобные способы графического представления программ, например, система GRASP [26], созданная в университете Auburn и поддерживающая Ada95, Си++ и Java.

3. Относительно известной, но остающейся принципиально важной является задача **верификации** программ. Под этим понимается многоаспектная деятельность как по выявлению тонких ошибок, случаев использования допустимых, но потенциально опасных свойств того или иного языка, сомнительных с точки зрения эффективности фрагментов программ, так и проверка программы на предмет соблюдения корпоративных или отраслевых стандартов на стиль программирования (например, Ada Style Guide [6]) или локальных ограничений. Одним из наиболее известных подобных средств является стандартная в ОС UNIX утилита lint для языка Си и ее позднейшие версии для Си++ [28]. Хотя формально некоторые из перечисленных задач являются, скорее, задачей традиционной компиляции, с практической точки зрения бывает более целесообразно выделить их в отдельную функциональность.

4. Сохраняет свою актуальность и задача **статического анализа** программ. С момента появления первых исследований, связанных с формальной оценкой программ по различным метрикам [27], было предложено много различных метрик и систем метрик (например, [29]). Во многих случаях статический анализ является необходимым элементом жизненного цикла разработки ПО и потому нуждается в соответствующей программной поддержке.

5. В последнее время, прежде всего в связи с успехом языка Java [99], вновь возник интерес к интерпретационной схеме исполнения программ. Существенно возросшая производительность вычислительных систем поставила интерпретацию в ряд практичных подходов для многих реальных применений, не требующих предельной эффективности. Для некоторых ЯП, например, для Си++ интерпретационная схема (или интерпретирующая трансляция, interpretive translation [32]), не отменяя очевидных достоинств непосредственного выполнения, потенциально может дать ряд преимуществ. Основной выгодой интерпретационного подхода является *повышенный диагностический сервис* периода исполнения, недостижимый в полном объеме для традиционного исполнения на

процессоре. Тем самым становится возможным адекватно выявлять (и, следовательно, преодолевать) множество типичных для Си++ динамических ошибок (некорректная работа с указателями, утечки памяти при динамическом управлении, неинициализированные переменные и т.д.). Можно сказать, что интерпретация позволяет превратить Си++ из мощного, но ненадежного языка в мощный и надежный.

Как известно, модель исполнения, разработанная для Java, основана на концепции виртуальной машины ([104], [105]). Для повышения уровня и возможностей отладки, а также для целей повышения надежности при исполнении программ на Си++ вполне практичным и актуальным становится создание **интерпретирующих и отладочных систем** на основе *виртуальной машины Си++* [126], [127].

Решение перечисленных задач базируется на совокупности алгоритмов лексического, синтаксического и семантического анализа исходных программ, то есть, на алгоритмах, составлявших существо традиционного понимания компиляции. Таким образом, перечисленные компоненты традиционного компилятора, часто называемые **компилятором переднего плана** (front end compiler [18]), выступают в качестве ядра современных систем разработки. Информационное взаимодействие между компилятором переднего плана и другими компонентами такой системы осуществляется посредством информационных структур (основные из которых - дерево программы и таблица символов), которые в данном случае перестают быть внутренними структурами компилятора и логически выносятся вовне его, образуя **промежуточное представление** программы (ПП, intermediate representation, IR [100]). Описанная модель представлена на рис. 2.

Еще раз отметим преимущества такой схемы. Во-первых, промежуточное представление, сформированное фазами анализа, по своему назначению и смыслу в точности соответствует исходной программе на ЯП. Оно несет в себе полное знание об исходной программе, в том числе, и такие его аспекты, которые присутствуют в исходной программе неявно.

Во-вторых, промежуточное представление содержит только ту информацию, которая относится к синтаксису и семантике исходной программы, и не включает каких-либо узкоспециализированных структур (например, объектный код). Тем самым, ПП может выступать в роли универсального интерфейса между компилятором переднего плана и разнообразных языко-ориентированных компонент систем разработки.

В-третьих, промежуточное представление, в отличие от многих форматов (где отладочная информация носит дополнительный и вторичный характер), может быть сконструировано таким образом, чтобы обеспечить максимально эффективный доступ ко всем его элементам.

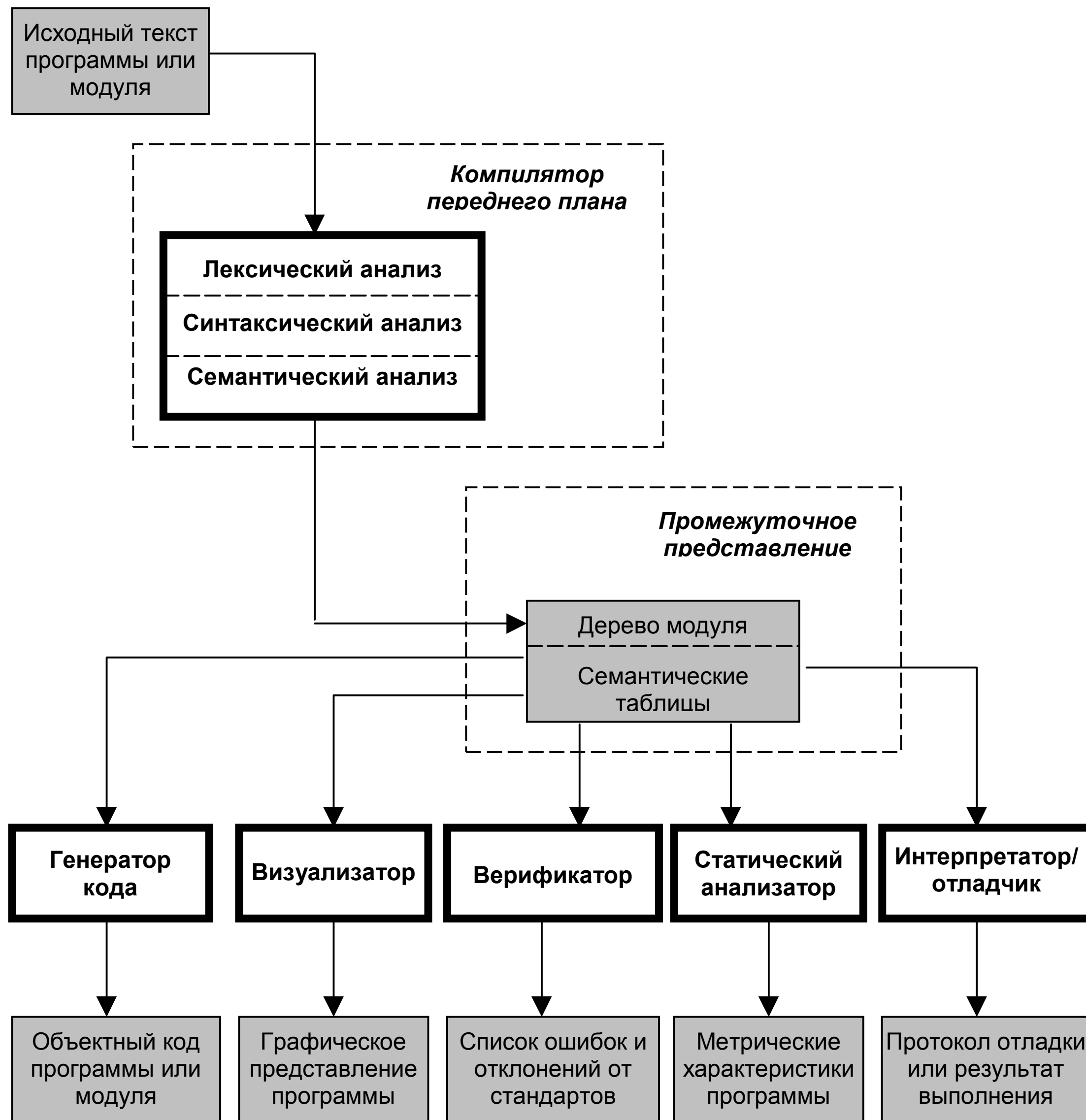

Рис.2 Компилятор переднего плана как ядро системы разработки ПО

Современные подходы к архитектуре СП

В дополнение к проведенному рассмотрению необходимо упомянуть еще о трех весьма важных и характерных подходах к созданию систем программирования, в которых модель "компилятор переднего плана - промежуточное представление программы" играет центральную роль. Речь идет о многоязыковых и многоплатформенных системах программирования, о продвинутых схемах комплексирования программ и о концепции семантического интерфейса.

Исторически понятие компилятора переднего плана возникло (см., например, [18]) именно в связи с созданием **многоязыковых систем программирования**, ориентированных на различные аппаратные архитектуры. Наиболее ярким примером многоязыковой системы служит комплекс GCC, создаваемый в рамках проекта GNU [24] под эгидой Фонда свободного ПО (Free Software Foundation). Система GCC подерживает такие входные языки, как Си, Objective C, Си++, Ада, Ада 95, и обеспечивает генерацию оптимизированного объектного кода для более полутора десятков программно-аппаратных платформ, среди которых процессоры компаний Inel, Motorola, Sun и т.д.

На аналогичной архитектуре основаны компиляторы семейства TopSpeed (языки Си, Pascal и Modua-2 для PC-совместимых персональных компьютеров), а также разработки компании ACE: система CoSy [22] для операционных систем Solaris, Linux, HP-UX, Windows NT; ее входными языками являются Си различных диалектов, Си++, Fortran-95, HPF, Java.

Основной идеей, лежащей в основе подобных систем, является сокращение общего числа компонент за счет выделения общей для всех компонент функциональности. Следствием этого является введение универсального интерфейса компонент переднего плана и "конечных" компонент (back-ends), в роли которого выступает промежуточное представление.

Выгода такой архитектуры наглядно представлена на следующих таблицах. Если система программирования рассчитана на ***n*** входных языков и ***m*** целевых процессоров, то при традиционной организации необходимо ***m*n*** компиляторов (Таблица 1).

Таблица 1. Традиционное построение системы программирования.

	Входной язык L_1	Входной язык L_2	. . .	Входной язык L_N
Целевой процессор P_1	**Компилятор L_1->P_1**	**Компилятор L_2->P_1**	. . .	**Компилятор L_n->P_1**
Целевой процессор P_2	**Компилятор L_1->P_2**	**Компилятор L_2->P_2**	. . .	**Компилятор L_n->P_2**
. . .	. . .	. . .	. . .	. . .
Целевой процессор P_m	**Компилятор L_1->P_m**	**Компилятор L_2->P_m**	. . .	**Компилятор L_n->P_m**

Если же строить систему программирования как конструкцию с явно выделенными языко-зависимыми частями (компиляторами переднего плана) и аппаратно-зависимыми генераторами объектного кода, взаимодействующими посредством единого промежуточного представления (см. Таблицу 2), то общее число компонент сокращается до ***m+n***. При этом, чтобы добавить в систему дополнительные возможности, например, реализовать поддержку нового ЯП, то достаточно разработать для этого языка компилятор переднего плана. В результате почти автоматически обеспечивается реализация нового ЯП для всех аппаратных архитектур, уже поддерживаемых таким комплексом. Аналогично, добавление генератора кода для некоторой новой архитектуры дает возможность разрабатывать программы для этой архитектуры на любом ЯП из числа реализованных.

Таблица 2. Система программирования как композиция языко- и платформенно-зависимых компонент

	Входной язык L_1	Входной язык L_2	. . .	Входной язык L_n
	Компилятор переднего плана L_1->IR	**Компилятор переднего плана L_2->IR**	. . .	**Компилятор переднего плана L_n->IR**
Целевой процессор P_1	**Генератор кода IR->P_1**			
. . .	. . .			
Целевой процессор P_m	**Генератор кода IR->P_m**			

Заметим однако, что на структуру промежуточного представления в подобных системах накладываются специфические ограничения. По очевидным причинам ПП должно быть архитектурно-нейтральным [112], то есть иметь относительно высокий уровень, и быть достаточно мощным и развитым, чтобы отображать семантику языковых конструкций всего набора поддерживаемых ЯП. На практике удовлетворение этих требований осуществляется за счет определенных отступлений от идеальной схемы, описанной выше: так например, компилятор GNAT формирует собственное промежуточное представление, отличное от принятого в GCC, и обеспечивает совместимость со всей системой GCC за счет введения дополнительной фазы конвертации своего ПП в стандартный формат системы (программа gigi).

Второе направление, в рамках которого концепция промежуточного представления программы играет центральную роль,- нетрадиционный подход к **комплексированию программ**.

Стандартная схема формирования исполняемых программ выглядит следующим образом (см. принципиальную схему на рис. 3): для каждого модуля программы компилятор порождает отдельный объектный код, причем он обрабатывает каждый модуль полностью

независимо от других модулей. Редактор связей воспринимает множество объектных кодов модулей, разрешает межмодульные ссылки и формирует исполняемый код всей программы.

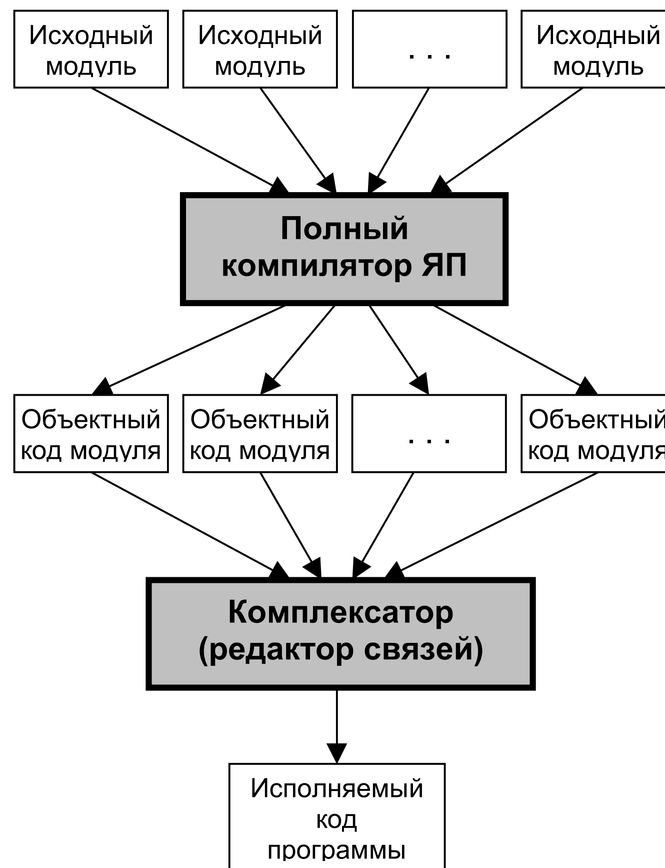

Рис. 3 Традиционный подход к формированию исполняемой программы

Такая схема в настящее время является доминирующей, однако она страдает несколькими фундаментальными недостатками. Во-первых, низкоуровневая структура объектного кода не обеспечивает необходимых диагностических возможностей; тем самым значительное число ошибок в межмодульных связях остаются невыявленными. Это обстоятельство хорошо известно и представляется настолько непреодолимым в рамках традиционного подхода, что, например, в стандарте языка Си++ специально оговаривается допустимость отсутствия диагностики при нарушении многих правил, связанных с межмодульными связями в программе.

Во-вторых, некоторые возможности современных языков, в частности, параметризованные типы и алгоритмы (шаблоны классов и функций в Си++, настраиваемые модули в Аде), с трудом вписываются в представленную схему. Например, шаблоны Си++ являются весьма мощным и крайне необходимым средством индустриального программирования. Однако их использование в рамках традиционной

схемы комплексирования может приводить к неконтролируемому разрастанию результирующего кода ("code bloat"), когда один и тот же глобальный шаблон настраивается в различных модулях идентичным набором аргументов.

Для ликвидации этого эффекта (который невозможно обнаружить в случае раздельной компиляции модулей) предлагается ряд методов и специализированных инструментов, некоторые примеры которых описаны в работах [66], [20], [69]: к ним относятся пост-линкеры, многоступенчатые схемы работы компилятора и комплексатора, особая дисциплина именования и хранения модулей, специальные правила программирования и т.д. Стратегия и тактика применения этих методик и средств часто практически целиком ложатся на разработчика, требуют глубокого анализа структуры всей программы и в целом не гарантируют полной оптимизации настроек шаблонов.

Предлагаются и в некотором смысле противоположные методики, в которых основная нагрузка по ликвидации размножения кода ложится на оптимизирующие комплексаторы, выявляющие повторные вхождения настроек и удаляющие их из результирующего кода. Заметим, что такой подход требует существенного усложнения функциональности комплексатора и не приводит к полному решению проблемы, хотя бы потому, что идентичные настройки остаются в объектных кодах отдельных модулей.

Наконец, третий существенный недостаток традиционной схемы с раздельной компиляцией и комплексировнием низкоуровневых объектных кодов заключается в ограниченных возможностях межмодульной и глобальной оптимизации программы [37]. В условиях, когда значительная часть информации об исходной программе потеряна, редактор связей (либо специальный оптимизатор, который работает по скомплексированной программе), имея дело, по существу, с машинным кодом, как правило, не в состоянии провести должный анализ особенностей исходной программы в целом. В частности, не располагая семантическим деревом всей программы и таблицами объектов, крайне затруднительно анализировать такие характерные особенности программы, как использование встраиваемых функций и шаблонов, полный граф вызовов функций, контролировать характер использования глобальных и статических объектов, границы распространения исключительных ситуаций, оптимизировать работу с динамическими объектами и т.д.

Необходимость преодоления указанных недостатков традиционной схемы комплексирования привела к появлению альтернативной схемы, которая (с теми или иными вариациями) реализована в таких системах, как компилятор Ultra C++ для операционной системы OS-9 фирмы Microware [61] и VisualAge for C++ фирмы IBM [62].

Например, компилятор Ultra C++, в отличие от традиционных, компилирует исходный текст единицы трансляции не в ассемблерный код конкретного процессора, а в специальное промежуточное представление (I-Code), не зависящее от входного языка и целевого процессора. Промежуточные представления модулей комплексируются

(заметим, что эта функциональность может быть реализована как в виде отдельной компоненты, так и встроена непосредственно в компилятор), в результате чего формируется промежуточное представление всей программы в целом. Далее этот программный образ обрабатывается оптимизатором и генератором кода. Таким образом, на завершающих этапах – при формировании результирующего кода – имеется полная информация обо всей программе, что позволяет выполнять обширный комплекс локальных, глобальных и межпроцедурных оптимизаций. Описанная схема представлена на рис. 4.

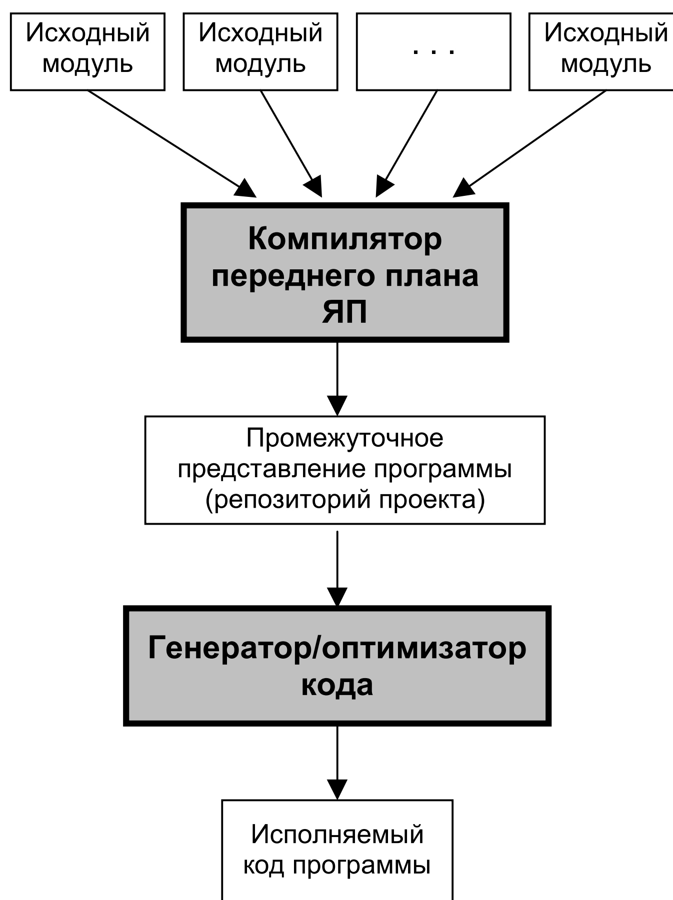

Рис. 4 Усовершенствованная схема комплексирования программ

Более того, такой подход позволяет значительно усовершенствовать методы разработки программ и стиль программирования. Так, механизм межмодульного взаимодействия в языках Си и Си++, специфицируемый посредством include-файлов, приводит к трудноуловимым ошибкам и многочисленным проблемам и, в частности, вынуждает программиста постоянно поддерживать в актуальном состоянии схему зависимостей модулей, используя средства, подобные make [25] и gendep [23]. По этим причинам данный механизм давно подвергается обоснованной критике [79]. Продвинутая схема комплексирования позволяет решить указанные проблемы, используя единый репозиторий программного проекта, представленный

в виде промежуточного представления. Например, компилятор VisualAge for C++ позволяет формировать и компилировать модули проекта способом, наиболее адекватным с точки зрения логической структуры программы, без учета межмодульных зависимостей. В результате использование include-файлов может быть практически полностью исключено.

Следствием такого подхода является существенное сокращение излишних перекомпиляций проекта, что обычно требует заметных временных затрат. При внесении даже незначительных изменений в некоторый заголовочный файл, как правило, необходимо повторно откомпилировать все модули, содержащие вхождение этого файла, независимо от того, влияют ли реально эти изменения на модуль. При использовании единого промежуточного представления становится возможным определять влияние внесенных изменений с точностью до оператора или объявления и, следовательно, сводить до минимума объем необходимых перекомпиляций.

Наконец, кратко рассмотрим третье важное понятие, реализация которого существенно основывается на промежуточном представлении программ. Речь идет о концепции **семантического интерфейса**.

Возникновение этой концепции связано как раз с возрастанием потребностей в “некомпиляционных” операциях над текстами программ, о которых говорилось выше. Общим в рассматриваемых операциях (статический анализ, верификация, визуализация и пр.) является необходимость доступа к тем или иным составным частям входной программы или модуля. Например, типичной задачей при вычислении метрических характеристик или проверке правильности стиля программирования является декомпозиция программы и анализ ее отдельных конструкций (подпрограмм, объявлений, операторов, выражений и т.д.). При этом, несмотря на наличие универсальных средств подобного рода, достаточно частой является потребность разработки специфических инструментов, работающих с программными текстами, для тех или иных внутрифирменных или коллективных нужд.

К настоящему времени потребность стандартизованного семантического интерфейса к программам на том или ином языке программирования осознана в достаточной степени, чтобы учитывать ее при рассмотрении вопросов компиляции. Дело в том, что для реализации практически полезного интерфейса доступа к программам в большинстве случаев недостаточно простого синтаксического анализа. Значительная часть сведений, которые необходимо получать посредством подобного интерфейса, носит существенно семантический характер. Очевидным примером служит задача получения любой информации, связанной с типами объектов и с отношениями между этими типами. Дополнительную нетривиальность добавляет этой задаче развитая система типов современных ЯП и, в частности, объектная ориентация промышленных языков (в частности, множественное и виртуальное наследование), а также наличие механизма настраиваемых типов; благодаря этой нетривиальности извлечение знания об иерархии и

свойствах типов возможно только на основе алгоритмов семантического анализа, характерных для компиляторов.

Заметный толчок концепция семантического интерфейса получила в последнее время, прежде всего в связи с процессом разработки и принятия в 1998 году Международного Стандарта ASIS (Ada Semantic Interface Specification) [8]. Аналогичные проекты для языка Си++ пока не носят характер стандартизации и находятся в стадии разработки; в качестве наиболее известной системы такого рода можно упомянуть SAGE++ [57].

Тесную связь между задачей компиляции и реализацией семантического интерфейса можно проиллюстрировать подходом, принятым при реализации стандарта ASIS [75], [120], для Ада-компилятора GNAT. Интерфейсные запросы, входящие в состав ASIS, реализуются на основе доступа к промежуточному представлению (аннотированному дереву программы и таблице символов), формируемому компилятором GNAT и доступному его клиентам. Таким образом, компилятор переднего плана берет на себя всю работу по синтаксическому и семантическому анализу входной программы, а интерфейс ASIS выступает в качестве стандартной "прослойки" между промежуточным представлением и специализированными конечными процессорами ("back-ends").

Принципиальную архитектуру системы, основанной на концепции семантического интерфейса, можно представить в виде следующей схемы:

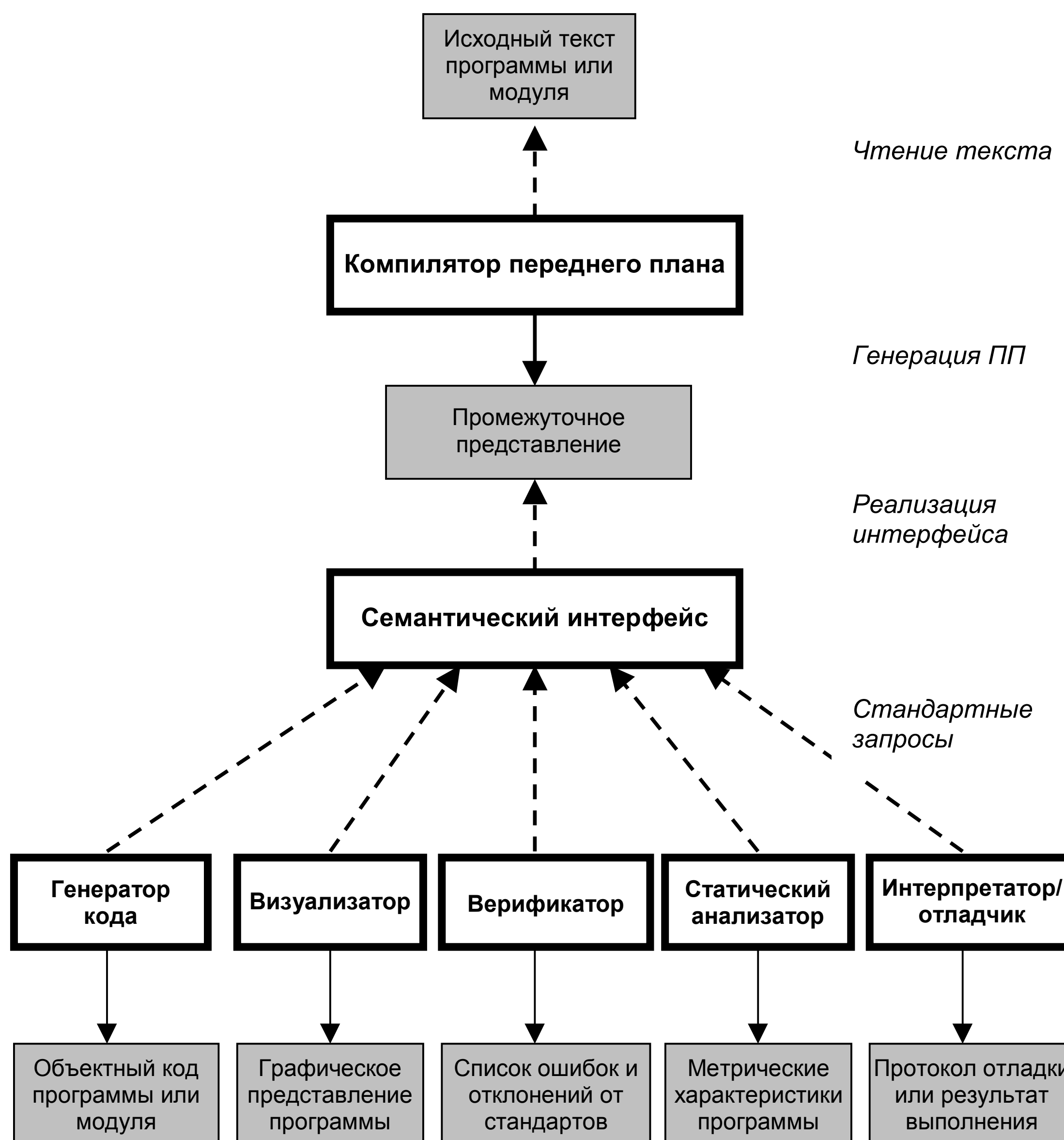

Рис.5 Система программирования на основе семантического интерфейса

Общий вывод из проведенных рассмотрений можно сформулировать следующим образом. **Компиляция из узко поставленной задачи получения кода, пригодного для исполнения на процессоре, превращается в центральную часть широкого спектра разнообразных семантических операций над текстами программ на ЯП**. Генерация кода из единственной цели компиляции превращается в одну из многочисленных задач, определенных над структурами промежуточного представления. Поэтому центр тяжести при реализации некоторого языка программирования в значительной степени смещается в сторону проектирования такого способа представления исходных программ, который был бы адекватен задачам, стоящим перед современными системами программирования.

В заключение данного раздела кратко остановимся на современном опыте проектирования и реализации промежуточного представления. В настоящее время эта область оформляется в самостоятельное направление исследований: создается большое разнообразие структур ПП, в зависимости от задач, стоящих перед разработчиками, проводятся конференции и семинары по данной проблематике, например [100]. Спектр подходов можно охарактеризовать следующим образом. На одном конце спектра располагается низкоуровневый код, представляющий собой, по существу, обобщенный ассемблер для процессора с традиционной архитектурой [94], либо язык некоторой абстрактной стековой машины [103-105], [107]. Другой конец этого спектра образуют полноценные языки программирования, в качестве которых может выступать, например, Си [21] или некоторый специально спроектированный ЯП [113]. В некоторых многоязыковых и многоплатформных системах встречаются сочетания этих "краевых" подходов [21].

Упомянутые подходы имеют очевидные основания и преимущества, однако вызывают также справедливую критику. "Обобщенный ассемблер" удобен для многоплатформных систем программирования, однако уже обсуждавшиеся проблемы соответствия с исходным текстом не позволяют считать его приемлемым решением для современных СП. ПП, ориентированное на абстрактную машину, критикуется во многом по тем же причинам; кроме того, такой подход слишком однозначно ориентирован на интерпретационную стратегию исполнения программ и, следовательно, затрудняет традиционную реализацию ЯП. Анализ недостатков подхода, связанного с понятием абстрактной машины, можно найти в известной работе [106].

С другой стороны, использование в качестве промежуточного представления некоторого ЯП приводит к ряду проблем, связанных с неизбежным "семантическим зазором" между исходным и целевым языком; для преодоления этого зазора разработчикам приходится прибегать к различного рода "трюкам" [21]. Это может существенно обесценить преимущества такого подхода. Кроме того, данная схема, по существу, переносит проблемы адекватного представления информации об исходной программе на систему программирования (компилятор и пр.) целевого языка. Наконец, использование ЯП в качестве целевого языка весьма неэффективно. Так, по оценке компании Microtec [69] переход от компиляции Си++ в Си к прямой компиляции в промежуточное представление дает 30-процентное повышение общего времени компиляции.

По этим причинам наиболее перспективным представляется подход, в котором в качестве промежуточного представления выступают структуры, традиционно формируемые компилятором в процессе обработки текстов исходных программ,- абстрактное синтаксическое дерево (или дерево, "аннотированное" семантическими атрибутами, или семантический граф программы), а также семантические таблицы (в зарубежной литературе обычно называемые таблицами символов). Например, дерево программы и таблицы символов использует в качестве ПП семейство компиляторов компании Green Hills Software [67].

Показательным примером служит GNAT - компилятор языка Ada95. Первоначально GNAT, будучи интегрированным в систему GCC, вырабатывал общее для всех компиляторов этой системы низкоуровневое промежуточное представление в формате RTL (Register Transfer Language). Однако в процессе реализации интерфеса ASIS для этого компилятора разработчики обеспечили в качестве альтернативного результата трансляции полное дерево исходной Ada-программы, которое GNAT формирует в процессе работы.

Язык Си++ и компилятор переднего плана Си++

Базовым языком для обсуждения в диссертационной работе выбран язык Си++. Это решение основывается на ряде обстоятельств, некоторые из которых были отмечены выше.

Как уже говорилось, язык Си++ является в настоящее время наиболее распространенным и перспективным языком промышленного программирования. Он содержит наиболее полный набор свойств и возможностей, выработанных всей историей развития ЯП. К существенным характерным свойствам Си++ следует отнести прежде всего мощную поддержку объектно-ориентированного подхода к разработке программ и механизм параметризации типов и алгоритмов. Широкий диапазон типов и развитые возможности построения пользовательских типов позволяют адекватно отразить особенности предметной области; строгие правила обращения с константными типами способствуют надежности программ. Повышению надежности создаваемых программ служит простой и гибкий аппарат управления исключительными ситуациями. Развитые схемы преобразования и приведения типов позволяют обеспечить достаточный компромисс между строгой типизацией и эффективностью исполнения программ. Средства явного управления обастями действия ("пространства имен") предоставляют удобный механизм структурирования больших программ.

Си++ является прямым преемником языка Си [14] и фактически включает его как подмножество. Тем самым, Си++ целиком содержит хорошо зарекомендовавшую себя традиционную модель вычислений языка Си, в том числе, развитый общеалгоритмический базис, широкие возможности конструирования новых типов и гибкие средства работы с памятью, включая арифметику над указателями. Это обстоятельство обеспечивает сохранение в актуальном состоянии миллионы строк программного текста, разработанного на Си, и дает дополнительные гарантии широкого использования Си++.

Помимо широкой распространенности и популярности, в том числе, и в отчественной практике программирования, язык Си++ служит технологической основой перспективной парадигмы, возникшей в недавнее время,- **обобщенного программирования** (generic programming, [50]). Основным инструментом реализации обобщенного программирования на языке Си++ служит механизм шаблонов [11, глава 14]; наиболее развернутым и убедительным примером использования этой парадигмы является Стандартная Библиотека Шаблонов (Standard

Template Library, STL), разработанная А.Степановым и М.Ли [52] и вошедшая в 1994 г. [51] в состав стандартной библиотеки Си++.

Еще одно обстоятельство, обусловившее выбор языка Си++ как базового для данной диссертационной работы,- принятие в конце 1998 г. Международного Стандарта ANSI/ISO этого языка. Факт стандартизации для такого большого, сложного и современного языка, как Си++, трудно переоценить. Если говорить коротко, Си++ становится инструментом промышленного программирования в общемировом масштабе. Приверженность все большего числа корпоративных разработчиков ПО к использованию типовых решений и стандартизованных инструментальных средств дает твердую уверенность в успешных перспективах Си++, по крайней мере, на ближайшие десять-пятнадцать лет (примерный срок смены поколений языков программирования).

Дополнительным аргументом является опыт, полученный автором в процессе проектирования и реализации компилятора переднего плана языка Си++ [114], [117-119], [121-124], [126-127]. Найденные проектные и технические решения послужили практической основой и подтверждением рассмотрений и выводов, составляющих содержание последующих глав.

При этом необходимо заметить, что информация о принципах и методах трансляции Си++, реализованных в конкретных зарубежных компиляторах, крайне скудна, прежде всего, по причине коммерческой ориентации большинства таких проектов. Имеется ограниченное количество публикаций преимущественно академического характера, среди которых следует отметить [56] и [54], относящиеся к ранним этапам эволюции Си++. Среди других работ интерес представляют [55] и [45], которые рассматривают отдельные аспекты компиляции Си++. Сведения о коммерческих проектах, основанных на языке Си++ [63] [65-69], хотя и позволяют получить представление о современном уровне решения проблем, связанных с реализацией Си++, как правило, ограничиваются изложением общих принципов и подходов и потребительскими характеристиками компиляторов.

Тем не менее, анализ отмеченных во Введении особенностей и тенденций в построении современных окружений программирования позволяет предложить следующую принципиальную модель ядра системы программирования для языка Си++. Модель (см. рис. 6) включает две основные сущности: компилятор переднего плана Си++ и промежуточное представление программы на Си++.

Рис. 6. Принципиальная модель компиляции

Промежуточное представление, конструируемое компилятором, представляет собой композицию трех основных компонент:

- *Дерево программы*, отражающее синтаксическую структуру исходной программы на Си++. Узлы дерева содержат дополнительные атрибуты, уточняющие семантические свойства конструкций.
- *Семантические таблицы*, содержащие семантические атрибуты всех объектов программы и представляющие иерархию областей действия.
- *Система типов* - структура, хранящая информацию обо всех типах, явно или неявно введенных в программе (как базовых, так и объявленных пользователем), и об отношениях между ними.

По начальным буквам английских эквивалентов перечисленных компонент (Trees, Tables, Types) промежуточное представление получило аббревиатуру ТТТ.

Компилятор переднего плана, помимо традиционных компонент - *лексического*, *синтаксического* и *семантического* анализа - включает специальную компоненту, ответственную за формирование

промежуточного представления,- *конструктор/анализатор ТТТ*. Эта компонента (которая может быть либо интегрирована в компилятор, либо реализована в виде отдельной системной библиотеки в составе СП) позволяет как строить ТТТ, так и воспринимать ранее построенное. Такая особенность дает компилятору принципиальную возможность обрабатывать отдельную единицу трансляции в контексте других, ранее откомпилированных модулей всей программы.

Такая архитектура компилятора, помимо удовлетворения потребности в полном сохранении информации об исходных программах, позволит достичь следующих важных целей:

Во-первых, компиляция единицы трансляции в контексте ранее откомпилированных единиц ("непрерывная" компиляция) устранит потребность в отдельном этапе обработки программы - разрешении внешних ссылок и, тем самым, позволит обойтись без использования традиционного редактора связей (linkage editor). Исключение этапа комплексации должно существенно увеличить общую эффективность системы программирования, а возможные ошибки в межмодульных связях будут выявляться непосредственно компилятором, что повысит точность и качество диагностики.

Во-вторых, снимается проблема "размножения" кода, когда идентичные настройки шаблонов присутствуют в нескольких единицах компиляции. Необходимо только обеспечить компиляцию описания шаблона перед обработкой его настроек. В связи с этим практически полностью исключается специальный этап поиска таких идентичных настроек и их удаление из результирующего образа программы.

В-третьих, единое промежуточное представление программы открывает возможность глубокой глобальной оптимизации, основанной на доступе к полной информации о программе.

Структура компилятора переднего плана, основанная на представленной выше принципиальной модели, рассматривается в начале главы 2. Механизм, реализующий непрерывный подход к компиляции, более подробно обсуждается в главе 4.

Цели и задачи диссертационной работы

На основе предложенной выше принципиальной модели компилятора переднего плана Си++ можно следующим образом сформулировать цели и задачи настоящей диссертационной работы.

Целью диссертационной работы являлась разработка компилятора переднего плана полного Стандарта Си++ как ядра системы программирования, содержащей набор технологических инструментов, нацеленных на широкий спектр операций над текстами программ, а также структуры результирующего промежуточного представления.

В соответствии с этой целью были определены следующие конкретные **задачи**:

- Разработать подход к реализации начальных фаз трансляции, обеспечивающий более высокую, по сравнению с традиционными реализациями, эффективность лексического анализа.
- Предложить методы реализации синтаксического анализа, учитывающие особенности синтаксиса и семантики входного языка.
- Описать и реализовать модель семантических таблиц компилятора, адекватную понятию области действия Си++ и правилам видимости имен и эффективно поддерживающую правила поиска имен Си++.
- Обосновать применимость модели семантических таблиц для реализации механизма шаблонов Си++ и для нетрадиционного подхода к компиляции.

Некоторые другие аспекты сформулированной цели, в частности, приницпы отображения системы типов в промежуточном представлении, реализация поисковых алгоритмов компилятора и семантика механизма исключительных ситуаций, рассматриваются в диссертационной работе [59].

Результаты, апробация и новизна работы

В процессе проведения исследований автором получены следующие **результаты**:

- Спроектирована общая структура промежуточного представления программ на Си++, включающая дерево программы, семантические таблицы и систему типов.
- Реализована система семантических таблиц, используемая для анализа, хранения и последующей обработки семантической информации о программных сущностях Си++ и иерархии контекстов, а также интерфейсный пакет функций доступа к семантическим таблицам.
- Разработана общая структура компилятора переднего плана Стандарта Си++.
- Разработаны и реализованы компоненты компилятора переднего плана, выполняющие лексический, синтаксический и (частично) семантический анализ. Общий объем компонент, реализованных лично автором, составляет более 70.000 строк на языке Си++.
- Создан первый в России промышленный компилятор языка Си++ и один из первых компиляторов, реализующих полный стандарт Си++.

Результаты, полученные в работе, изложены в ряде печатных публикаций, докладывались на научных конференциях и семинарах, в частности:

- на Первой российской конференции "Индустрия программирования'96", Москва, 3-4 октября 1996 г.;
- на IV Международной конференции "Развитие и применение открытых систем", Н.Новгород, 27-31 октября 1997 г.;
- на Интернет-форуме компании Инфоарт "Языки программирования и Интернет", июль 1998;
- на Ломоносовских чтениях в МГУ им. Ломоносова в 1993-1999 гг.

Научная новизна представляемой работы может быть охарактеризована следующими тезисами:

1. Исследованы современные тенденции создания средств и систем программирования, ориентированных на промышленное использование.

2. На основе выполненных исследований разработаны архитектура и принципы построения компилятора переднего плана языка Си++ как системообразующего ядра многоцелевой системы программирования, включающей набор компонент для выполнения различных операций над текстами программ.

3. На основе анализа синтаксиса Си++ разработан и реализован подход к построению лексического и синтаксического анализаторов Си++, названный непрерывной компиляцией, который обеспечивает высокую эффективность анализаторов. В частности, предложен и обоснован перенос алгоритмов идентификации имен на фазу лексического анализа.

4. Проведен подробный анализ понятия области действия Си++, предложена и реализована модель семантических таблиц компилятора, адекватно отражающая это понятие.

5. Исследована семантика параметризуемых типов и алгоритмов Си++, предложены и реализованы принципы реализации механизма шаблонов в компиляторе переднего плана. Показана адекватность модели семантических таблиц семантике шаблонов классов и функций.