Глава 1
Фазы компиляции и организация лексического разбора

Приступая к обсуждению, необходимо прежде всего сформулировать базовые **требования** к реализации компилятора переднего плана, на удовлетворение которых должны быть направлены решения, предлагающиеся в данной и последующих главах работы:

- **Полнота и конформность Стандарту**: реализация должна поддерживать все свойства языка Си++ (и, в частности, его лексики), определенные в Стандарте, и обеспечивать полное соответствие Стандарту.
- **Универсальность**. Реализация должна обеспечивать необходимую общность с тем, чтобы поддерживать не только задачи трансляции, но и целый ряд других операций над текстами Си++ (анализ программ, извлечение семантической информации и т.д.).
- **Эффективность**. Реализация должна обеспечивать производительность, сравнимую с лучшими современными промышленными компиляторами Си++.

Далее в тексте диссертации ссылки на Стандарт Си++ [11] будут даваться в следующей форме: [Std, *Номер-главы-или-раздела*, §*Номер-абзаца*]. Например, ссылка [Std, 2.1, §1] обозначает первый абзац из раздела 2.1 Стандарта.

1.1. Фазы компиляции. Различные формы реализации фаз

Процесс компиляции специфицирован в Стандарте в виде последовательности операций, сгруппированных в девять фаз [Std, 2.1, §1]. Специальная оговорка подчеркивает концептуальный характер спецификаций: "реализации должны быть устроены так, как если бы эти фазы действительно выполнялись одна за другой, хотя на практике различные фазы могут быть объединены".

В данном разделе приводится краткое описание фаз трансляции, как они определены в Стандарте. На основе анализа содержания этих фаз предлагается подход к частичной реорганизации начальных (1-6) фаз, которые можно назвать **обобщенным лексическим анализом**. Предлагаемая реорганизация не будет нарушать совокупного эффекта указанных фаз и, тем самым, будет соответствовать требованиям Стандарта.

Далее предлагается общий подход к реализации фаз трансляции, который можно назвать "непрерывной компиляцией", в отличие от традиционных подходов, предполагающих реализацию некоторых этих фаз в виде независимых процессоров.

Прежде чем перейти к содержательному рассмотрению фаз трансляции, следует определить **объект** трансляции. В Стандарте

объектом начальных (1-3) фаз служит исходный файл (source file) [Std, 2, §1]. Этот термин не подразумевает однозначного соответствия какому-либо конкретному внешнему представлению, что специально подчеркивается в [Std, 2.1, §1, п.1]. Поэтому для целей нашего рассмотрения определим объект трансляции, не используя терминологии, перегруженной посторонними ассоциациями.

Итак, объектом трансляции считается **исходный текст**, понимаемый как непрерывный неструктурированный поток символов некоторого алфавита в произвольной кодировке. Алфавит исходного текста и вопросы, связанные с его кодировкой, будут подробнее обсуждаться ниже.

Следующая принципиальная схема (см. рис. 7) иллюстрирует последовательность фаз трансляции. Ниже даются краткие комментарии по содержанию этих фаз.

Фаза 1: Предварительная обработка. Отображение символов исходного текста в базовое множество символов внутреннего представления, с заменой триграфов и формированием, при необходимости, *универсальных-имен-символов*.

Фаза 2: "Склеивание" строк. Строка исходного текста, завершающаяся символом "обратная косая черта", соединяется (с отбрасыванием конца строки и данного символа) с последующей.

Фаза 3: Первичный лексический анализ. Декомпозиция исходного текста на лексемы препроцессора (*pp-tokens*) с удалением комментариев и, быть может, заменой последовательностей пробельных символов на один пробел.

Фаза 4: Препроцессирование. Выполнение директив препроцессора и реализация макрорасширений. В частности, обработка директивы `#include` приводит к рекурсивному выполнению фаз 1-4 для исходного текста из заданного в этой директиве файла. На этой фазе, строго говоря, возникает понятие **единицы трансляции** [Std, 2, §1, §2], которое в контексте данного изложения можно определить как исходный текст (см. выше), полученный, возможно, из нескольких исходных файлов [Std, 2, §1] применением операции препроцессирования.

Фаза 5: Обработка литералов. Преобразование содержимого символьных и строковых литералов к внутреннему виду.

Фаза 6: Конкатенация литералов. Конкатенация соседних строковых литералов в единую лексему.

Фаза 7: Собственно трансляция. Окончательный лексический анализ (преобразование лексем препроцессора в лексемы), синтаксический и семантический анализ. "Полученные в результате лексемы синтаксически и семантически анализируются и транслируются" [Std, 2.1, §1, п.7].

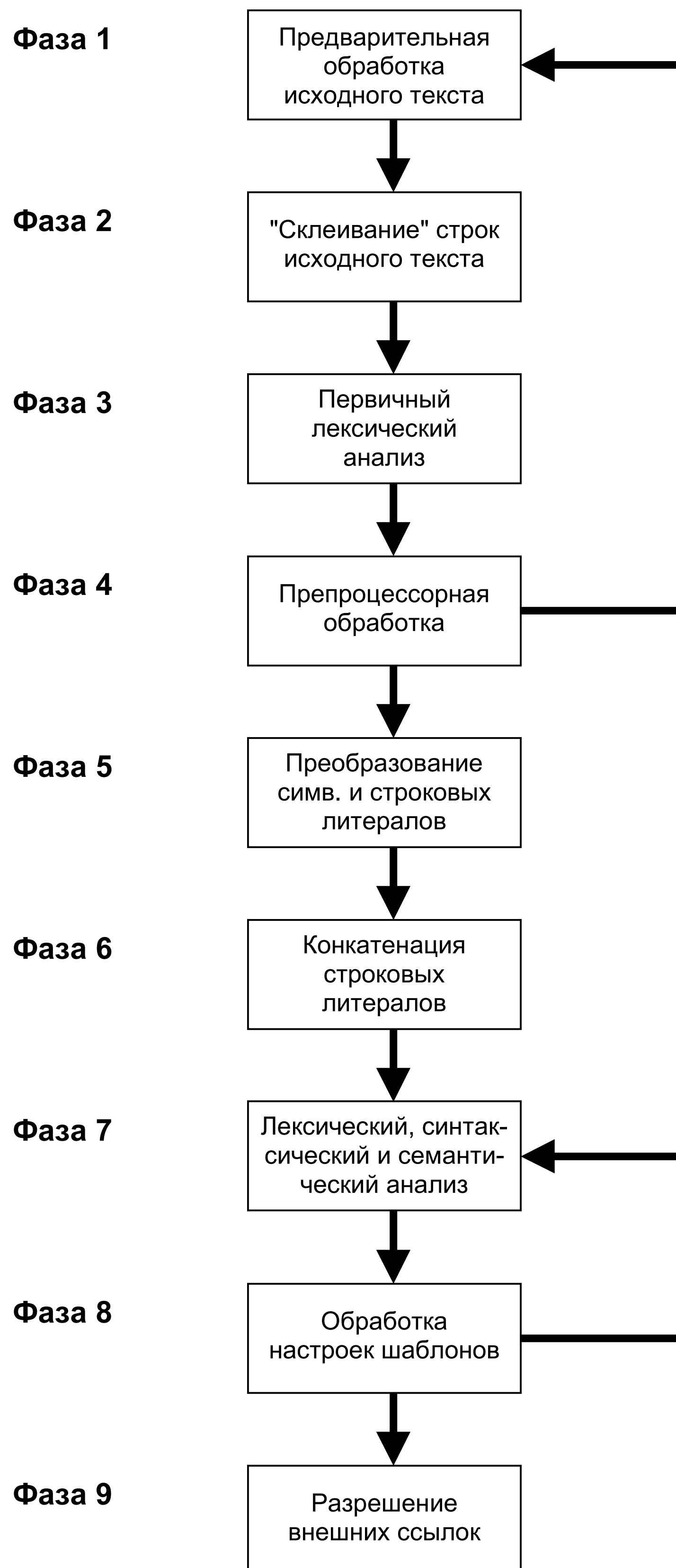

Рис. 7. Фазы трансляции

Фаза 8: Пост-обработка настроек шаблонов. Единица трансляции, прошедшая обработку на фазах 1-7, проверяется на предмет вхождения в нее настроек шаблонов, объявленных в других единицах трансляции. Если такие вхождения имеются, они настраиваются с привлечением информации о таких шаблонах из других единиц трансляции.

Операция настройки, хотя это явно не оговорено в Стандарте, концептуально предполагает некоторый синтаксический и семантический анализ как самих настроек (*имен-шаблонов*), так и настраиваемых шаблонов, то есть, обращение к предыдущей фазе.

Аспекты, связанные с обработкой шаблонов, подробно обсуждаются в главе 4. Здесь только отметим, что данная фаза на практике реализуется либо специальным постпроцессором, либо как некоторая совместная деятельность компилятора и редактора связей (см. Введение, "Современные подходы к архитектуре СП"). Предложенная во Введении (см. рис. 6) архитектура компилятора переднего плана, как будет показано в главе 4, позволит обеспечить "гладкую" компиляцию настроек в рамках собственно компилятора и, тем самым, снять необходимость в специальном постпроцессоре.

Фаза 9: Разрешение внешних ссылок. Устанавливаются связи между использующими вхождениями внешних по отношению к данной единице трансляции сущностей и их определяющими вхождениями в других единицах трансляции. Обработанные на предыдущих фазах единицы трансляции объединяются в единый образ программы, который содержит всю информацию, необходимую для ее выполнения.

Данная фаза традиционно называется *редактированием связей* или *комплексацией*. Системы, принципиально ориентированные на создание многомодульных программ и раздельную компиляцию (большинство систем, основанных на современных ЯП, в том числе, Си++), реализуют эту фазу в виде отдельного процессора - компоновщика (linker) или редактора связей (linkage editor). Недостатки такой организации отмечались во Введении; там же обсуждалась модель, позволяющая исключить этап редактирования связей.

1.2. Предварительная обработка исходного текста: фазы 1-2

Фазы трансляции, указанные в заголовке раздела, специфицируют преобразование исходного текста к виду, удобному для последующего анализа [Std, 2.1, §1, пп.1,2]. Содержательно эти фазы задают следующую последовательность операций:

1. Преобразование символов исходного текста в "базовое множество исходных символов" (basic source character set).
2. Замена триграфов на эквивалентные односимвольные представления.
3. Замена "расширенных" символов соответствующими "универсальными-именами-символов" (*universal-character-name*).

4. "Склеивание" строк исходного текста.

Замена триграфов и "склеивание" строк (удаление пар символов "обратная косая черта" и "конец строки") присутствуют, в основном, по историческим причинам и поэтому в данном рассмотрении не представляют интереса. Таким образом, сосредоточимся на операциях 1 и 3.

Операция 1 [Std, 2.1, §1, п.1], как она специфицирована в Стандарте, фиксирует лишь факт преобразования, никак не задавая его содержательной стороны. При этом специально подчеркивается, что на практике вместо базового множества исходных символов может использоваться любая внутренняя кодировка, если обработка символов в этой кодировке будет выполняться так, как задано Стандартом. Таким образом, Стандарт жестко не определяет ни внешнюю кодировку исходного текста, ни его внутреннее представление. От реализации требуется лишь выдержать функциональность фаз, которая описывается в терминах базового множества исходных символов и *универсальных-имен-символов*.

Внутренний алфавит - базовое множество исходных символов [Std, 2.2, §1] - включает общеупотребительный набор: символ пробела, четыре управляющих символа, латинские буквы в обоих регистрах, десять цифр и традиционные знаки препинания, разделители и символы операций. Повторим, что конкретная кодировка символов этого алфавита не фиксируется. Очевидно, что такое множество символов имеется практически в любой из общеупотребительных кодировок, в частности, в основной и альтернативной DOS-кодировках, CP866, CP1251 и т.д.

Резюмируя сказанное, сформулируем следующие *требования* к реализации операции 1:

а) Если форма представления (кодировка), используемая в исходном тексте, допускает представление символов базового множества, то операция 1 может быть тождественной. Иными словами, в качестве внутреннего алфавита может использоваться алфавит исходного текста.

б) Алфавит исходного текста может не содержать каких-либо символов из числа следующих: `#`, `/`, `'`, `[. ]`, `|`, `{`, `}`, `~`. В этом случае отсутствующие символы могут представляться в исходном тексте с помощью триграфов, а эквивалентные им символы внутреннего представления - *универсальными-именами-символов*. Такой вариант сохраняет тождественность операции 1.

в) Если алфавит исходного текста не включает какой-либо из перечисленных символов, а их внутреннее представление в виде *универсальных-имен-символов* нежелательно, то в качестве внутреннего алфавита может быть выбрана любая кодировка, содержащая указанные символы. В этом случае операция 1 должна представлять собой прямое отображение (mapping) символов входного алфавита в эквивалентные им символы внутреннего алфавита.

Теперь перейдем к рассмотрению операции 3 - обработке "расширенных" символов или, точнее, символов, не входящих в базовое множество. Прежде всего подчеркнем сам факт допустимости дополнительных символов в программах на Си++. Правда, синтаксис языка определяет, по существу, единственное место вхождения таких символов - в идентификаторах [Std, 2.10, §1]. Тем не менее, это обстоятельство весьма существенно для отечественной практики программирования, делая возможным именование программных объектов русскоязычными идентификаторами.

Порядок обработки дополнительных символов следующий: символ исходного текста, не входящий в базовое множество, преобразуется в эквивалентное представление в виде восьми шестнадцатиричных цифр, предшествуемых символами `\U`, то есть, в нотацию *универсального-имени-символа*. При этом число, представленное шестнадцатиричными цифрами, должно служить кодировкой исходного символа в стандарте ISO 10646 ([15]; общеупотребительное название этого стандарта - Unicode). Стандарт Си++ определяет допустимое для программ на Си++ подмножество символов Unicode [Std, Annex E], в которое входят все символы кириллицы.

Таким образом, Стандарт допускает два способа представления дополнительных символов, не входящих в базовый набор: непосредственное, как символов входного алфавита, и в нотации *универсальных-имен-символов*. Из этого можно сделать следующие практические выводы:

а) Во-первых, реализация Си++ может допускать наличие во входном алфавите символов, не входящих в базовое множество. При этом необходимо, чтобы коды Unicode этих символов входили в допустимое Стандартом подмножество. Во-вторых, операция 3 должна обеспечивать соответствующее преобразование таких символов в Unicode.

Однако, такое преобразование не является обязательным, если внешний и внутренний алфавиты совпадают (см. операцию 1). В этом случае достаточно проверить вхождение (Unicode-кодировки) входного символа в подмножество, допустимое Стандартом.

Таким образом, содержание операций 1 и 3 в вариантах реализации Си++, наиболее естественной для отечественной практики программирования (когда входным и внутренним алфавитами служат символы в кодировке CP1251, альтернативной кодировке и т.п.), может представлять, по существу, только контроль вхождения символа из исходного текста в указанное подмножество Unicode.

б) Если входной алфавит не содержит дополнительных символов, можно, тем не менее, в рамках базового множества исходных символов задать вхождение произвольного дополнительного символа. Это возможно потому, что все реализации, согласно Стандарту, должны поддерживать явную нотацию *универсального-имени-символа*. Это дает принципиальную возможность готовить Си++-программы с использованием редакторов, поддерживающих Unicode, и, наоборот,

позволяет интегрировать компилятор, реализующий Стандарт, с современным текстовым инструментарием.

1.3. Особенности препроцессорной обработки: фазы 3-4

Препроцессорная обработка текстов имеет свою историю, отраженную в многочисленных публикациях [97], [98]. Первоначально препроцессирование применялось, в основном, как средство придания большей мнемоничности текстам на языках ассемблера. Относительно невысокий уровень оригинальной версии языка Си [93] также послужил причиной введения этапа препроцессирования перед компиляцией Си-программ, позволив внести в этот язык средства, отсутствующие в нем (например, константы периода компиляции, встраиваемые функции), и частично компенсировать явно недостаточную лексическую выразительность Си. Мобильность Си-программ, являющаяся одним из наиболее убедительных достоинств языка Си, на практике в значительной степени достигается (помимо адекватности его вычислительной модели традиционным аппаратным процессорам) благодаря средствам условной компиляции, реализованным в препроцессоре. Наконец, механизм текстовых вставок (директива препроцессора `#include`) служит средством межмодульного связывания и обеспечивает языку Си некий суррогат модульности (подробнее этот вопрос будет обсуждаться в главе 4).

В подавляющем большинстве систем программирования этап препроцессирования реализуется в виде отдельной компоненты - препроцессора, выход которого (текст-результат препроцессирования) передается на вход собственно Си-компилятору. Такой подход соответствует идеологии ОС UNIX (первоначальной "среды обитания" языка Си), согласно которой результирующая функциональность достигается посредством стандартной конвейеризации независимых компонент, каждая из которых выполняет одну строго определенную задачу. Кроме того, спецификации функций препроцессора в значительной степени независимы от определения собственно языка Си, что дает принципиальную возможность использовать препроцессор для общих целей преобразования текстов, не связанных с задачами компиляции.

Однако, на настоящем этапе развития языков программирования практически все преимущества препроцессирования программ до их компиляции утратили свое значение. Любой ЯП общего назначения (это относится, в частности, и к Си++) так или иначе включает свойства, которые для языка Си обеспечивались на уровне препроцессора. Более-менее систематическое проектирование ЯП обязательно предполагает обеспечение должного уровня читабельности программ (так, при создании языка Ада это требование относилось к числу наиболее важных). Конструкция языка Ада дает также пример решения проблемы мобильности (так называемые "спецификации представления"), не снижающего уровень языка и не требующего механизма условной компиляции. Модульность также обеспечивается на уровне самого

языка; наряду с Адой, пример адекватного решения проблемы модульности дает Java [99].

По мере утраты преимуществ модели "препроцессор-компилятор", ее существенные недостатки в очень многих случаях выходят на первый план и становятся критическими. Двумя фундаментальными пороками этой модели следует считать, с одной стороны, бесконтрольную возможность модификации синтаксиса языка (что является обратной стороной повышения мнемоничности), а с другой - отсутствие синтаксического и, что самое главное, семантического контроля корректности макроопределений, в частности, контроля типов для макросов с параметрами. Показательным является отсутствие средств препроцессирования в языках Ада и Java.

Однако безусловная необходимость совместимости с языком Си послужила причиной сохранения препроцессорных механизмов в Си++. Поэтому следует оценить подходы к реализации этих возможностей с позиций современных требований.

Прежде всего, нет никаких причин реализовывать препроцессор как отдельную компоненту СП. Препроцессирование как таковое, в отрыве от компиляции, в настоящее время практически не представляет интереса. С другой стороны, интеграция этих механизмов в лексический анализатор сама по себе способна дать заметное повышение общей производительности, так как исключит по крайней мере активацию дополнительной компоненты и затраты на передачу текста от препроцессора компилятору. Кроме того, обсуждаемая ниже концепция непрерывной компиляции, которая, в частности, предполагает интеграцию препроцессора и компилятора, дает дополнительные преимущества в плане эффективности. Что же касается встречающихся на практике потребностей компиляции без препроцессирования или, наборот, только препроцессорной обработки исходного текста, то такие возможности легко реализовать в виде специальных режимов компилятора.

Следующий аспект проблемы связан со способом интерпретации препроцессором исходной информации. Исторически сложились два подхода, которые можно назвать байтовым и лексемным [45]. Согласно первому подходу, входной поток препроцессора трактуется как последовательность байтов, подлежащая анализу. Во втором случае на вход препроцессору поступает последовательность лексем, полученная в процессе некоторой предварительной обработки исходного текста. Долгое время выбор той или иной модели специально не оговаривался; книга [45], которая послужила отправной точкой процесса стандартизации Си++, давала только общие рекомендации (см. разд 16.2.1с) о предпочтительности лексемного подхода, специфицированного стандартом ANSI C.

В окончательной версии Стандарта Си++ содержится недвусмысленное указание на лексемную трактовку. Фаза 3, предшествующая собственно препроцессированию ([Std, 2.1, §1, п.3], см. также рис. 7), как раз специфицирует декомпозицию исходного текста на лексемы препроцессора (*pp-tokens*), которые поступают на вход

препроцессору (фаза 4). Как следует из спецификации фаз 5, 6 и начала фазы 7, выход (результат работы) препроцессора представляет собой последовательность таких же лексем препроцессора.

Резюмируя сказанное, можно обобщенно описать начальные фазы трансляции следующим образом. Последовательность символов входного алфавита, преобразованных во внутреннее представление (фазы 1-2), подвергается декомпозиции (фаза 3) на элементарные лексические единицы - лексемы препроцессора, согласно их синтаксису [2.4]. Полученная последовательность лексем препроцессора поступает на вход препроцессору (фаза 4), который выполняет обработку поступившей последовательности согласно правилам из главы 16 Стандарта. Результатом обработки является последовательность единиц, семантика которых совпадает с семантикой лексем препроцессора. Далее лексемы препроцессора из полученной последовательности, которые представляют собой строковые литералы, подвергаются двум очевидным преобразованиям (фазы 5-6). Наконец, лексемы препроцессора преобразуются в лексемы (начало фазы 7), которые и поступают на синтаксический и семантический анализ.

Учитывая сравнительную простоту и очевидность фаз 1 и 2 (см. разд. 1.2), представим наиболее существенную часть описанных действий в виде следующей схемы:

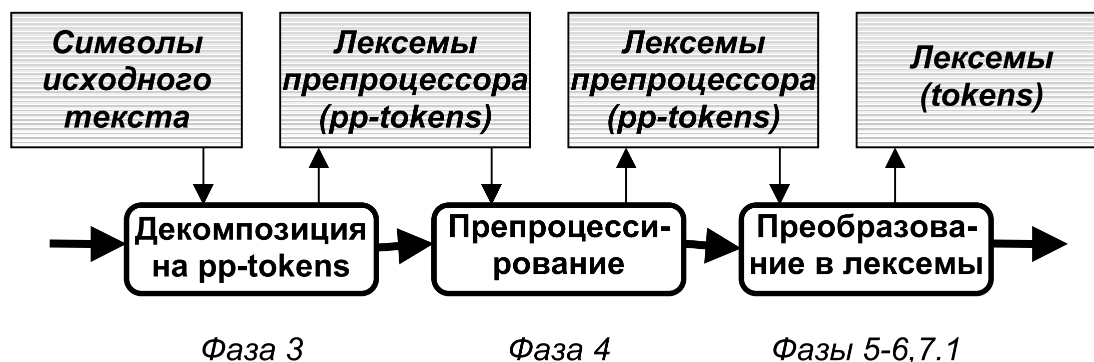

Рис. 8 Схема организации фаз 3-7.1 лексического разбора

1.4. Завершение лексического разбора. Концепция непрерывной компиляции

Чтобы завершить обсуждение фаз трансляции, необходимо кратко сказать о двух промежуточных фазах - пятой и шестой. Согласно [Std, 2.1, §1, пп.5,6], фаза 5 специфицирует преобразование содержимого символьных и строковых литералов к так называемому множеству символов выполнения (execution character set). Речь идет о том обстоятельстве, что оттранслированная программа может быть предназначена для исполнения в операционной среде, отличной от среды компиляции. Это характерно прежде всего для технологии кросс-

компиляции. Однако разница между базовым множеством символов и множеством символов выполнения проявляется и в более простых ситуациях: например, программа на Си++ компилируется в среде Windows с активной кодовой страницей CP1251, а исполняется в окне эмуляции DOS (которое, в свою очередь, работает под управлением Windows), где действует кодировка CP866. В этом случае (хотя бы для того, чтобы при выводе строковых литералов получить ожидаемый эффект) при трансляции необходимо отображение одного множества символов в другое. Понятно, что содержательная сторона этой операции не представляет специального интереса.

То же относится и к семантике фазы 6, которая предусматривает конкатенацию двух соседних строковых литералов в единый литерал. Необходимость этой операции непосредственно вытекает из устройства лексики Си++, которое было унаследовано от языка Си.

В контексте последующих рассуждений интерес вызывает только операция преобразования лексемы препроцессора (*pp-token*) в лексему (*token*), специфицированная как начало фазы 7 [Std 2.1, §1, п.7] - непосредственно перед синтаксическим анализом. Существо такого преобразования становится ясным при сравнении синтаксиса этих двух понятий (см. [Std, 2.4] и [Std, 2.6]). Результат сравнительного анализа упомянутых синтаксических правил представлен в виде следующей таблицы:

Таблица 3. Соответствие конструкций *pp-token* и *token*.

Варианты конструкции *pp-token* [Std, 2.4]	**Соответствующие варианты конструкции** *token* [Std, 2.6]	**Комментарии**
identifier	*identifier* *keyword*	Лексема препроцессора *identifier* в процессе преобразования будет отнесена либо к той же категории, либо к категории *keyword*. Заметим, что некоторые операции (например, **`sizeof`** или **`new`**), а также булевские литералы также изображаются посредством служебных слов.
header-name	Нет	При препроцессировании заменяется на лексемы (*tokens*) из файла, имя которого задано в данной лексеме препроцессора.
pp-number	*integer-literal* *floating-literal* недопустимая лексема	Синтаксис *pp-number* является по существу обобщением синтаксиса *integer-literal* и *floating-literal*. [Std, 2.9, §1]. Синтаксис *pp-number* может включать конструкции, не являющиеся правильными литералами [Std, 2.4, §§3-4]; в этом случае они классифицируются как некорректные лексемы.
character-literal *string-literal*	*character-literal* *string-literal*	При преобразовании не изменяются

preprocessing-op-or-punc	*operator punctuator*	Препроцессорные операции # [Std, 16.3.2] и ## [Std, 16.3.3] обрабатываются на этапе препроцессирования и в выходной поток не попадают. Знаки остальных операций и пунктуаторы при преобразовании не изменяются.
each non-white-space character that cannot be one of the above	Недопустимая лексема	Лексемой препроцессора считается, помимо перечисленных выше конструкций, также любой непробельный символ. Иными словами, препроцессор должен пропускать такие символы в выходной поток без всяких изменений. Только последующие фазы трансляции могут классифицировать такой символ как некорректный.

Таким образом, преобразование лексемы препроцессора в лексему, поступающую на вход синтаксического анализа, включает всего две содержательные операции: преобразование *pp-number* в целочисленный или плавающий литерал и классификация идентификатора. Первая операция носит характер проверки корректности *pp-number* с точки зрения синтаксиса литералов. Способы реализации второй операции хорошо известны (см., например, [34], [35]) и, как правило основываются на использовании подходящей хеш-функции, отображающей фиксированное множество служебных слов в некоторый диапазон целых значений.

Проведенные рассмотрения приводят к следующим общим *выводам* относительно реализации лексического разбора.

1. Вместо серии промежуточных представлений, характерных для фаз лексического этапа трансляции, ввести единое внутреннее низкоуровневое понятие **лексемы**. Тогда операции, специфицированные для этих фаз, могут быть выражены как последовательность преобразований над одной и той же структурой, реализующей понятие лексемы.

2. Таким образом, вместо последовательности операций, преобразующих исходный текст (и, далее, единицу трансляции) из одного представления в другое, лексический разбор рассматривается как единый неразрывный процесс, определенный над общей структурой лексемы. Этот процесс реализуется одной компонентой компилятора - *лексическим анализатором*. Различные варианты последовательностей преобразований (например, с препроцессированием или без него) задаются в виде режимов такого анализатора.

3. Взаимодействие лексического анализатора с другими компонентами компилятора (прежде всего, с синтаксическим анализатором) может быть организовано на различных принципах, однако наиболее простой и удобной схемой представляется обычная подпрограммная связь: синтаксический анализатор обращается к лексическому анализатору за очередной лексемой.

Перечисленные положения образуют ту часть концепции **непрерывной компиляции**, которая относится к этапу лексического разбора. В главе 4 эта концепция будет расширена применительно к другим этапам обработки программы.

В заключение сделаем ряд замечаний технического характера. В развитой системе программирования клиентом лексического анализатора может выступать не только синтаксический анализатор; широкая номенклатура требуемых операций над текстами программ может привести к введению дополнительных процессоров, которые могли бы воспринимать лексемы, полученные из исходных текстов. Типичным примером служит различные форматтеры и анализаторы текстов, работающие без привлечения семантической информации.

Такие процессоры также могут взаимодействовать с лексическим анализатором по подпрограммной схеме, получая от него одну лексему при каждом обращении; однако их специфическая функциональность может привести к формированию дополнительных структур на основе лексем.

Так, если представить себе некоторый форматтер, предназначенный для структурирования текста программы на Си++ согласно определенным стандартам стиля, то в качестве одной из его опций мог бы быть режим, при котором он выполняет форматирование всей единицы трансляции (включая обработку макроопределений и include-файлов), но физически не вставляет содержимое include-файлов в основной текст, а выводит их, например, сразу после основного текста.

Независимая обработка таким форматтером основной единицы трансляции и включенных в него файлов может привести к потере информации, влияющей на макрообработку включенных файлов; поэтому решением может служить организация буфера лексем, в который "складываются" обработанные лексическим анализатором лексемы из include-файлов. Такой буфер может быть организован как в самом лексическом анализаторе, так и в его клиентах.

Проиллюстрируем механизм буфера лексем рисунком. Пусть имеется единица трансляции Си++, содержащая следующий текст:

```
#define DECL(T,V) T V;
int a;
#include "FileName.h"
DECL(int,i)
```

Пусть файл `FileName.h`, указанный в директиве `#include`, содержит следующую строку:

```
DECL(float,x)
```

Тогда описанный текстовый форматтер, будучи клиентом лексического анализатора и работая в режиме, описанном выше, мог бы сформировать из полученных лексем структуру, подобную следующей (см. рис. 9). На этом рисунке сплошные стрелки отображают основной поток лексем без вставки содержимого include-файлов; пунктирная стрелка обозначает последовательность лексем, получающихся в

результате полного препроцессирования. Наконец, двойная пунктирная стрелка отмечает возможный путь просмотра исходного текста вообще без учета вставляемых файлов.

Существенно, что такая структура естественным образом формируется за один проход лексического анализатора.

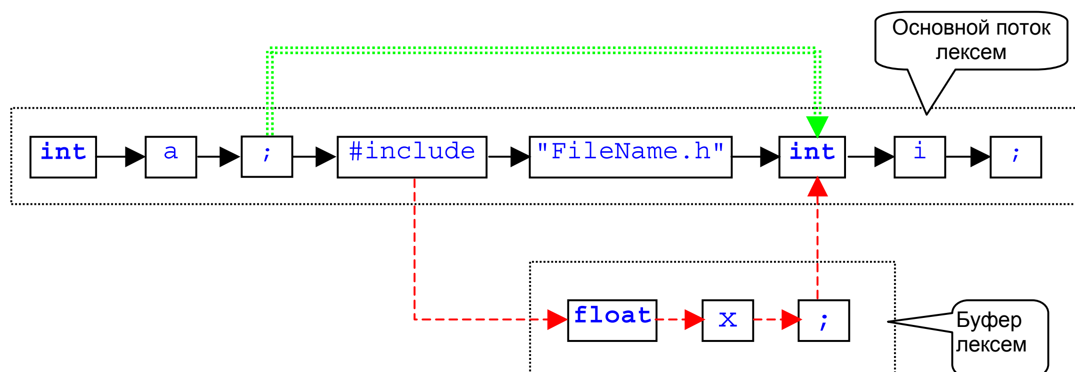

Рис. 9. Логическая организация буферов лексем

Заметим, что подобная структура буферов лексем используется и в самом компиляторе в двух специальных случаях: для "отложенной" обработки тел функций-членов классов и для разрешения неоднозначностей. Первая проблема заключается в том, что, согласно правилам Си++ [Std, 9.X], тела функций-членов, описанных в теле класса, должны транслироваться в контексте всего класса. Из этого правила следует, что тела таких функций-членов необходимо сохранить (без компиляции) до завершения обработки всего класса, после чего вернуться к их компиляции. Эффективнее всего не оперировать в таком случае с исходным текстом, а сохранять тела в виде последовательностей лексем в специальных буферах.

Вторая проблема связана с реализацией алгоритма разрешения синтаксических неоднозначностей [Std, 6.8], который был предложен в книге [45]. Здесь буфер лексем используется для организации просмотра вперед для идентификации вида конструкции (оператор или объявление). Этот аспект использования механизма буферов будет рассматриваться в разд. 2.4.

1.5. Лексическая структура Си++

В предыдущем разделе мы определили функциональность лексического анализатора, совмещающего функции препроцессирования и собственно лексического разбора. Теперь мы можем сосредоточиться на рассмотрении лексической структуры языка Си++ как таковой, абстрагируясь от особенностей препроцессорной обработки и рассматривая фазы 1-6 и начало фазы 7 как единый процесс.

Итак, основным содержанием указанных начальных фаз трансляции является декомпозиция исходного текста и, позднее, единицы трансляции (которые, повторим, можно считать неструктурированным непрерывным потоком символов входного алфавита) на последовательность отдельных **лексем** - минимальных лексических элементов языка, обладающих в нем конкретной семантикой. Лексический анализатор поставляет лексемы для обработки на последующие фазы трансляции; в компиляторе потребителем лексем выступает компонента, называемая синтаксическим анализатором. Принципиальную схему описанного взаимодействия можно изобразить следующим образом (см. рис. 9):

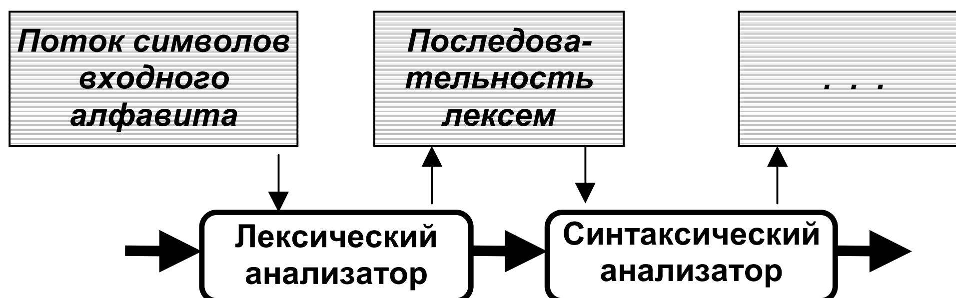

Рис. 10 Взаимодействие лексического и синтаксического разбора

1.5.1. Подход к определению базового множества лексем

Переходя к анализу лексической структуры Си++, необходимо прежде всего уточнить характер требований к реализации соответствующих программных компонент. С этой точки зрения можно выделить следующие аспекты лексики Си++:

- Свойства, которые необходимо поддерживать безусловно в любой реализации;
- Свойства, которые не являются абсолютно необходимыми, однако их поддержка может повысить качество компилятора;
- Свойства, избыточные для решения задач собственно компиляции, однако существенные для прочих возможных операций над Си++-программами.

Первая группа свойств включает множество лексем (tokens) как таковых, в том виде, как они описаны в Стандарте: именно такие лексемы поступают на последующие фазы трансляции.

Вторую группу свойств образуют сущности, которые можно рассматривать как **атрибуты** лексем из первой группы. К таким атрибутам можно отнести, например, координаты лексемы по единице трансляции (имя первичного исходного файла или имя `include`-файла, номер строки в файле, номер позиции лексемы в строке). Учет атрибутов

лексем, строго говоря, не является необходимым для трансляции, однако позволяет повысить диагностические возможности компилятора, а также может потребоваться другим, "некомпиляционным" компонентам СП. Кроме того, выбранная схема компиляции и, прежде всего, механизм восстановления разбора после обнаружения ошибок может потребовать введения специальных дополнительных лексем ("псевдолексем") для целей маркирования ошибок или замены некорректных последовательностей лексем.

Наконец, третья группа свойств включает последовательности символов входного алфавита, образующие **пробельные символы** (white spaces): пробелы, допустимые Стандартом управляющие символы и комментарии. Согласно Стандарту, пробельные символы играют роль разделителей лексем и должны быть полностью элиминированы (заменены на пробелы) на фазе 7, непосредственно перед синтаксическим и семантическим анализом. Однако лексический анализ, в силу своего универсального характера, может выступать как начальный этап работы различных языковых процессоров - от текстовых форматтеров до статических анализаторов, систем визуализации и т.п.

Кроме того, необходимо учесть следующее обстоятельство. Некоторые лексемы, формально являющиеся лексемами препроцессора (pp-tokens) и в принятом в предыдущем разделе подходе обрабатывающиеся внутри фазы лексического анализа ("на лету"), могут, тем не менее, потребоваться компонентам-клиентам лексического анализатора. Простейшим примером служит задача форматирования исходного текста без учета содержимого `include`-файлов. В этом случае лексема препроцессора *header-name* будет выступать как полноправная лексема (или последовательность лексем), поставляемая анализатором. В этом случае также понадобится представление некоторых пробельных символов (например, комментариев) в виде дополнительных лексем. Наличие описанных "лексем" также следует отнести к третьей группе особенностей лексики Си++.

Резюмируя сказанное, можно предложить следующее принципиальное решение:

1. Выходной алфавит (обобщенной) фазы лексического анализа и, одновременно, входной алфавит последующих фаз обработки образуется множеством лексем **всех** видов, из числа перечисленных выше. Частным случаем последующих фаз обработки является синтаксический и семантический анализ в компиляторе.

Конкретная компонента-потребитель лексем, в зависимости от своей проблемной ориентации, может воспринимать как все поставляемое множество лексем, так и некоторое его подмножество. Алфавит фазы лексического анализа представляет собой конкретизацию абстракции "лексема", введенной в предыдущем разделе.

Такое решение, принципиально не усложняя реализацию синтаксического анализа, позволяет сохранить практически всю необходимую информацию об исходной единице трансляции и, тем самым, учесть все возможные требования к данной фазе.

2. Минимальный набор **атрибутов**, который необходим для реализации понятия лексемы, включает:

- Вид лексемы: код, условный номер и т.п., в зависимости от конкретной реализации;
- Изображение лексемы: для элементарных лексем может отсутствовать; для таких лексем, как идентификатор или литерал, может представлять собой ссылку в таблицу имен или в некоторое хранилище; то же относится к "дополнительным" лексемам вида "комментарий" и т.п.
- Координаты первого символа лексемы по исходному тексту единицы трансляции: либо в абсолютном виде, либо относительно предыдущей лексемы.

3. Особый случай образует способ представления "неотображаемых" пробельных символов: пробелов, символов табуляции, новой страницы и новой строки.

Следуя принципу универсальности, пробелы (символы `' '`) можно было бы представлять в виде дополнительных лексем. Однако, такое решение привело бы к неоправданной перегрузке внутреннего представления и, как следствие, к снижению эффективности алгоритмов, работающих с ним. Поэтому пробелы не считаются "полноправными" лексемами, а их вхождения однозначно выражаются в координатах ближайшей правой лексемы.

С другой стороны, "лексическая семантика" прочих перечисленных пробельных символов не столь однозначна. Так, символ горизонтальной табуляции, как правило, обозначает некоторое (определенное окружением) число пробелов. Поэтому имеет смысл преобразовывать вхождения символов горизонтальной табуляции в соответствующее количество пробелов (это количество может поставляться анализатору по запросу к окружению либо передаваться ему в виде параметра при вызове). Далее полученные пробелы обрабатываются обычным образом.

Что же касается вертикальной табуляции, символов новой строки (line feed) и новой страницы (form feed), то они присутствуют в лексике Си++ только по историческим причинам (для поддержки совместимости), и их первоначальная семантика практически утеряла свое значение. Поэтому представляется оправданной упрощенная трактовка таких символов: предлагается интерпретировать их как символы новой строки (за исключением случаев, когда непосредственно перед ними располагается обратная косая черта, так как комбинации вида "`\`конец-строки" имеют специальный смысл, см. фазу 2 в разд. 1.1).

Таким образом, из всех конструктов фазы лексического анализа специальным образом должны быть представлены только комментарии.

1.5.2. Классификация лексем базового множества

Основываясь на принятом выше подходе, рассмотрим состав базового множества лексем. Напомним, что, наряду с лексемами, явно определенными в Стандарте, мы включили в базовое множество некоторое количество дополнительных лексем ("псевдолексем") для придания фазе лексического анализа необходимой общности. В частности, в базовое множество включен ряд лексем, определенных в Стандарте как лексемы препроцессора (*pp-tokens*).

Итак, базовое множество состоит из следующих групп лексем:

- **Служебное слово**. Конструкция с синтаксисом идентификатора, смысл которой предопределен в языке и, вообще говоря, отличен от смысла идентификатора.
- **Идентификатор**. Лексема, используемая для обозначения некоторой программной сущности, которая вводится посредством объявления или описания.

  В эту категорию включены также идентификаторы, обрабатываемые на фазе препроцессирования - имена директив препроцессора, имена макросов и прагм и их параметров. В зависимости от конкретного режима лексического разбора, в эту группу могут попадать и идентификаторы, формируемые на этапе препроцессирования (так называемые "склеенные" идентификаторы, образованные при обработке лексем ##).
- **Литерал**. Лексема, используемая для непосредственного выражения некоторого значения базового типа.
- **Знак операции**. Лексема, обозначающая некоторую операцию, определенную в языке. (Заметим, что некоторые операции обозначаются посредством служебных слов.)
- **Разделитель (пунктуатор)**. Лексема, используемая для отделения синтаксических конструкций друг от друга, и/или для повышения наглядности конструкций языка, и/или для снятия синтаксических неоднозначностей.

  В данную группу включены лексемы препроцессора # и ##, обрабатываемые на фазе препроцессирования.
- **Комментарий**. Дополнительная лексема, образованная согласно синтаксису короткого или длинного комментария, определенному в Стандарте [Std, 2.7].

1.5.3. Модификация базового множества. Суперлексемы

Продолжим обсуждение лексической структуры языка Си++. До сих пор наши рассмотрения не выходили за рамки определений соответствующей главы Стандарта. Однако практическое использование базового множества лексем, описанного в предыдущем подразделе, наталкивается на ряд сложностей.

Прежде всего, следует указать на значительный объем лексической части. Это неизбежно повлечет сравнительно высокую сложность реализации соответствующих программных компонент, причем эта сложность проявится как при использовании тех или иных генераторов лексических анализаторов, например, утилиты *lex*, входящей в состав системы UNIX, или фукционально идентичной программы *Flex* из комплекса GCC, так и в случае "ручного" программирования. Необходимо сразу отметить, что при практической реализации компилятора именно большой объем и нерегулярная структура лексической части языка послужили важнейшими аргументами в пользу более эффективной "ручной" разработки. Однако даже в этом случае предварительные усилия по упрощению лексической структуры могут дать заметный эффект в плане повышения производительности.

Тезис о сложности синтаксиса Си++, превосходящего сложность всех других современных ЯП, вообще говоря, нуждается в обосновании. Однако, чтобы подтвердить этот тезис, необходимо, во-первых, определить некоторую недвусмысленную и формально вычисляемую меру сложности и, во-вторых, провести строгий сравнительный анализ, например, грамматик Си++ и Ада95. Учитывая, что в Стандартах этих ЯП грамматики специфицированы неформально, подобное исследование выглядит достаточно масштабным и может составить тему отдельной работы. Поэтому аргументом в пользу приведенного утверждения может служить только неформальные оценки, высказываемые авторитетными разработчиками компиляторов, а также личный опыт автора, в течение многих лет активно работавшего с обоими языками. Частичным подтверждением сказанного могут служить результаты работы [88], хотя строгость рассмотрений в данной работе также недостаточна. В Главе 2 мы вернемся к обсуждению этих вопросов.

Далее, одной из существенных черт языка Си++ является неоднозначность его синтаксиса (подробнее эти аспекты будут обсуждаться ниже, а также в главе 2). Некоторые явные неоднозначности отмечены в самом тексте Стандарта [Std, 6.8], однако в языке имеется ряд скрытых проблем, проявляющихся только при проектировании синтаксического анализатора. Некоторые из них будут подробно рассмотрены далее в этом подразделе. Тем не менее, один принципиальный момент следует отметить уже сейчас: многие неоднозначности синтаксической структуры Си++ могут быть сняты уже на уровне лексического разбора, за счет удачной модификации базового множества лексем.

Существо последующих операций над базовым множеством лексем заключается во введении в него лексем специального вида, называемых далее **суперлексемами**. Суперлексемы конструируются на основе конкретных лексем или групп лексем из базового множества путем применения к ним операций, которые назовем **агрегацией**, **сепарацией** и **динамическим расширением**.

Рассмотрим перечисленные операции подробнее.

1.5.4 Агрегация

Самый первый анализ синтаксических правил Си++ показывает, что в языке имеется некоторое число стандартных последовательностей

лексем. Так, после служебных слов `continue`, `break` всегда следует символ '`;`', после служебных слов `if`, `while`, `for` идет левая круглая скобка, и т.д. Если рассматривать такие комбинации как последовательности отдельных лексем, то синтаксическому анализатору потребуется отдельное обращение за каждой такой лексемой (если взаимодействие между лексической и синтаксической компонентами компилятора организовано на принципе "поставщик - потребитель"). Кроме того, это приведет к разрастанию решающих таблиц синтаксического анализатора или (в случае его построения на принципах рекурсивного спуска) заметному увеличению кода.

Альтернативное решение состоит в том, чтобы считать подобные (безальтернативные) комбинации **полноправными лексемами**, заменив ими первоначальные последовательности.

Важно отметить, что такое решение реально не приведет к снижению эффективности лексического анализатора, так как символы, образующие комбинацию лексем, так или иначе должны быть прочитаны. Вопрос заключается только в определенной реорганизации программ лексического разбора применительно к предложенной схеме.

Следующий аспект агрегации лексем в суперлексемы связан с несколько более сложными фрагментами синтаксиса, но также вполне очевиден. Примером служит оператор безусловного перехода. Он состоит из служебного слова `goto`, следующего за ним идентификатора метки и завершающей точки с запятой:

```
goto OnceMore;
```

Такого рода последовательности также можно агрегировать в единственную суперлексему; однако, в отличие от предыдущих случаев, такая суперлексема будет содержать дополнительный атрибут, связанный со "смыслом" идентификатора. Однако это обстоятельство является тривиальным, так как этот атрибут непосредственно "наследуется" из лексемы *identifier* и никак не усложняет алгоритм агрегации.

Наконец, еще один резерв агрегирования составляют часто встречающиеся на практике последовательности лексем. В отличие от конструкций вида "`break;`", они не являются безальтернативными, но, тем не менее, достаточно легко и эффективно выявляются в процессе лексического разбора. Примерами таких конструкций служат пустой список формальных параметров функции `( )` или `( void )`, произвольное число и типы формальных параметров `(...)`, "хвост" агрегатного инициализатора `,...)` и т.д.

В заключение следует отметить, что принцип агрегации можно было бы распространить на значительно большее число конструкций. Однако высокая сложность языка Си++ и его насыщенность семантическими понятиями объективно ограничивают применение этого принципа. В этом смысле весьма показательным является механизм шаблонов, который с точки зрения лексического разбора функционально аналогичен макросредствам.

В принципе, любое вхождение константного выражения можно было бы трактовать как суперлексему, так как такое выражение, по определению, должно полностью вычисляться на этапе компиляции.

Если не учитывать шаблоны, в языке имеется *пять* случаев возможного вхождения константных выражений [Std, 5.19, §1]: задание размерности массива в объявлении, альтернатива в операторе **switch**, инициализирующее значение перечислителя, размер битового поля, инициализатор статического члена класса. Все эти константные выражения, вместе с окружающими лексемами, можно было бы рассматривать как суперлексемы, например:

```
case <конст-выражение> :

[ <конст-выражение> ]
```

К сожалению, в общем случае это невозможно, так как константное выражение может содержать вхождения нетиповых параметров шаблона (которые, по определению, считаются константами):

```
template < int N >
void f ( void )
{
    . . .
    switch ( <выражение> )  {
        . . .
        case N+1 :
        . . .
    }
}
```

Вычислить значение N+1 и, тем самым, свернуть константное выражение (вместе с лексемами **case** и ':') в одну суперлексему можно только в контексте настройки шаблона. В то же время, синтаксический анализ шаблона и вычисление значения выражения должны, как правило, выполняться в точке его объявления (подробнее об этом см. в главе 4).

В заключение приведем список суперлексем, получаемых лексическим анализатором посредством операции агрегации.

Имя суперлексемы	**Первичные лексемы, образующие суперлексему**
xlxmPtrOperator	* ***const** ***const volatile** ***volatile** ***volatile const**
xlxmConversionFunctionId	Служебное слово **operator**, если за ним следует *type-id*
xlxmAsmDefinition	**asm** ("строка") ;
xlxmPrimaryExpression	**this** **false** **true** *Literal* **operator** *Знак* в контексте выражения *identifier* в контексте выражения

xlxmDirectMembSelector	-> *identifier* -> *Qualification* :: *identifier*
xlxmIndirectMembSelector	. *identifier* . *Qualification* :: *identifier*
xlxmMemberAccessSpecifier	**private :** **protected :** **public :**
xlxmFunctionSpecifiers	**inline** **virtual**
xlxmStorageClassSpecifier	**mutable** **auto** **register** **static** **extern**
xlxmConstVolatile	**const** **const volatile** **volatile** **volatile const**
xlxmLinkage	**extern** "*строка*"
xlxmArgTerminator	) **const**) **const volatile**) **volatile**) **volatile const**
xlxmEmptyParameterList	(**void**)
xlxmUndefParameterList	(...)
xlxmEmptyList	( )
xlxmCommaEndCompound	, }
xlxmStandardTypeSpecifier	**void** **int** **unsigned** **signed** **short** **char** **long** **double** **float** **wchar_t**
xlxmAssignment	*= += -= /= >>= <<=
xlxmEqual	== !=
xlxmRelation	< > <= >=
xlxmShift	>> <<
xlxmAdd	+ -
xlxmDivide	/ %
xlxmPtrToMemb	.* ->*
xlxmCase	**case** (
xlxmSwitch	**switch** (
xlxmIf	**if** (
xlxmWhile	**while** (
xlxmFor	**for** (
xlxmLabel	*label* :
xlxmDefault	**default** :
xlxmBreak	**break** ;
xlxmContinue	**continue** ;
xlxmGoto	**goto** *label* ;
xlxmTemplate	**template** <
xlxmNew	**new** ::**new**
xlxmDelete	**delete** **delete**[] ::**delete** ::**delete**[]

Из представленной таблицы видно, что в ряде случаев решение об агрегации некоторой последовательности лексем принимается на основе ближайшего контекста разбора. Об этом будет идти речь при описании операции динамического расширения (разд 1.5.6), а также в главе 2. Так, последовательность лексем вида **`operator`** `+` может агрегироваться в лексему `xlxmPrimaryExpression`, если в текущий момент идет обработка некоторого выражения. Это соответствует случаю явного вызова функции-операции, например, **`operator`**`+(a,b);`. С другой стороны, вхождение такой же последовательности в заголовок функции-операции (то есть, в контексте объявления) будет свернуто операцией сепарации (см. след. раздел) в суперлексему `xlxmId`. Подобные взаимозависимости подчеркивают необходимость совместной реализации операций над лексемами.

1.5.5 Сепарация

Данный вариант конструирования множества суперлексем касается единственной лексемы базового множества "идентификатор".

Основная причина необходимости сепарации идентификаторов заключается в том, что синтаксис Си++, опирающийся на лексему "идентификатор" в чистом виде, является неоднозначным. Простейшим и в то же время типичным примером может служить следующий фрагмент:

X (***Y***)

где ***X*** и ***Y*** - идентификаторы. Синтаксис Си++ допускает две трактовки данной конструкции: либо как *объявление* объекта ***Y*** с типом ***X***, либо как *преобразование типа* объекта ***X*** к типу ***Y*** (преобразование, заданное в "функциональном" стиле). В большинстве контекстов допускается вхождение обоих конструкций, так как и объявление, и выражение (частным случаем которого является преобразование) считаются операторами; точнее говоря, первая конструкция - это оператор-объявление (*declaration-statement*, [Std, 6.7]), вторая - оператор-выражение (*expression-statement*, [Std, 6.2]).

Полный перечень трактовок конструкции ***X*** (***Y***) выглядит следующим образом:

- Объявление объекта: `T(a);` , где `T` - *имя-типа*;
- Преобразование типа в контексте выражения: `x = T(a);`, где `T` - *имя-типа*, `x` - объект типа `T`;
- Вызов (частный случай *постфиксного-выражения*): `f(x)`, где `f` - имя функции или имя объекта типа "указатель на функцию";
- Инициализатор в конструкторе (*ctor-initializer*):

  ```
  C::C() : m(x), B(y) { ... }
  ```

  где `C` - класс, `m` - имя члена класса `C` , `B` - имя его непосредственного базового класса.

Быть может, еще более наглядным примером служит конструкция вида ***a*****`*`*****b***. Здесь вид сущностей, именуемых идентификаторами ***a*** и ***b***, решающим образом влияет на синтаксис: если ***a*** - некоторый тип, то синтаксический анализ должен интерпретировать данный фрагмент как *объявление* объекта ***b*** типа "указатель на ***a***". В противном случае перед нами простое выражение. В высшей степени показательно, что для тех случаев, когда интерпретацию ***a*** произвести невозможно в принципе (если ***a*** квалифицировано именем типового параметра шаблона), в язык введено специальное служебное слово **`typename`**, уточняющее вид сущности. Подробнее об этом см. в разд 4.2.

Обсуждение этих и подобных неоднозначностей содержится в [Std, 8.2] и [Std, 6.8]. Проблема заключается в следующем: хотя процесс разрешения неоднозначностей является "чисто синтаксическим" [Std, 6.8, §3], то есть, смысл идентификаторов при этом не используется, тем не менее, необходима информация "о различиях между *именами-типов* и прочими именами" [там же]. Заметим, что приведенные утверждения Стандарта противоречат друг другу, так как информация о том, что данный идентификатор обозначает тип, вообще говоря, не является "чисто синтаксической".

Так или иначе, для упрощения структуры синтаксиса Си++ и для разрешения синтаксических неоднозначностей в общем случае удобно или необходимо привлекать определенную информацию о смысле идентификатора. Одним из вариантов решения этой задачи и является *сепарация* лексемы "идентификатор". Суть этой операции состоит в том, что лексический анализатор, распознав лексему "идентификатор", не отдает ее непосредственно на синтаксический разбор, а производит попытку смыслового анализа полученного идентификатора. В результате этого анализа выдается лексема-эквивалент первоначальной лексемы, которая отражает "смысл" идентификатора.

Необходимо уточнить три обстоятельства. Во-первых, сам по себе семантический анализ идентификатора (идентификация имени) так или иначе производится компилятором; речь идет о том, что такое действие выполняется не на этапе синтаксического или семантического анализа, как это традиционно делается в компиляторах языков класса Pascal (или даже Java), а в процессе лексического разбора, практически сразу после извлечения идентификатора из исходного текста.

Вторая особенность заключается в том, что произведенная идентификация не сохраняется в семантических атрибутах лексемы "идентификатор", а влияет на сам вид результирующей лексемы. Эту идею схематически можно пояснить так: если очередной идентификатор ***T*** из входного текста был идентифицирован как тип, то результатом лексического разбора становится не лексема "идентификатор" с семантическими атрибутами вроде `Имя=`***T***, `Семантика=`*Указатель-в-таблицу-типов*", а лексема "спецификатор типа" со свойственным именно такой лексеме семантическим атрибутом.

Эта разница может показаться сугубо реализационной и не слишком существенной, однако именно она решающим образом влияет и на значительное упрощение синтаксиса Си++, используемого в

компиляторе, и на алгоритмы разрешения неоднозначностей. Если синтаксический анализатор строится путем его автоматического синтезирования из некоторого формального описания грамматики входного языка, то, как правило, он использует именно вид (условный код) лексемы, а семантическая инорфмация для разбора не привлекается. В случае "ручного" создания анализатора, например, на принципах рекурсивного спуска анализ семантики такой "многосмысленной" лексемы как "идентификатор", приведет к переусложнению анализатора.

Наконец, последнее, третье обстоятельство заключается в том, что предложенная идея может быть реализована только для однопроходной схемы компиляции. Иными словами, идентифицировать полученное имя лексический анализатор может, только обратившись к уже построенной к данному моменту части семантических таблиц. Если же лексический разбор составляет отдельный проход компилятора, а синтаксический анализ начинает работать только после того, как из исходного текста построена завершенная последовательность лексем, то для выполнения сепарации отсутствует необходимая информация.

Итак, операция сепарации, реализованная в лексическом анализаторе, выполняет семантический анализ поступившей лексемы "идентификатор", привлекая информацию из построенных к данному моменту семантических структур, отражающих текущий контекст компиляции. Основная идея, согласно процитированным выше положениям Стандарта, заключается в том, чтобы разделить идентификаторы, обозначающие некоторый **тип**, от идентификаторов других программных сущностей. Однако ассортимент изобразительных средств Си++ дает возможность также многих других видов идентификаций имен. Конкретная функциональность данной операции становится ясной из следующей таблицы:

Суперлексема	**Позиция/контекст исходного идентификатора и вид именуемой сущности**
`xlxmQualifiedClassSpecifier`	Простой или квалифицированный идентификатор, обозначающий имя классового типа (*class-name*).
`xlxmQualifiedTypeSpecifier`	Простой или квалифицированный идентификатор, обозначающий typedef-имя (*typedef-name*).
`xlxmQualifiedEnumSpecifier`	Простой или квалифицированный идентификатор, обозначающий имя перечислимого типа (*enum-name*)
`xlxmConstructorId`	Квалифицированное имя вида `C::C`, где `C` - идентификатор классового типа.
`xlxmExplicitConstructorId`	Простое имя классового типа, предшествуемое служебным словом **`explicit`**.
`xlxmDestructorId`	Квалифицированное имя вида `C::~C`, где `C` - идентификатор классового типа.
`xlxmIdExpression`	Простой или квалифицированный идентификатор в контексте выражения.
`xlxmId`	Простой или квалифицированный идентификатор (в том числе, имя функции-операции) в контексте объявления.

xlxmClassSpecifier	Конструкция **class** *Qualification* :: *identifier* { в объявлении классового типа.
xlxmElaboratedClassSpecifier	Конструкция **class** *Qualification* :: *identifier* в объявлении объекта.
xlxmEnumSpecifier	Конструкция **enum** *Qualification* :: *identifier* { в объявлении перечислимого типа.
xlxmElaboratedEnumSpecifier	Конструкция **enum** *Qualification* :: *identifier* в объявлении объекта.
xlxmAccessSpecifier	*Access-modifier*: квалифицированный идентификатор как объявление в теле класса.
xlxmDirectBaseClass	Простой или квалифицированный идентификатор непосредственного базового класса в контексте заголовка конструктора.
xlxmMemberName	Простой или квалифицированный идентификатор члена класса в контексте заголовка конструктора.
xlxmTemplateName	Имя-шаблона (*template-name*), за которым, быть может, следует левая угловая скобка < .
xlxmTemplateTypeParameter	Идентификатор типового параметра шаблона (в контексте объявления шаблона).

1.5.6 Динамическое расширение

Третья операция по модификации исходной грамматики Си++ определена для некоторого подмножества суперлексем, полученных в результате операции сепарации (см. предыд. подраздел). Эта операция введена как абстракция действий лексического анализатора по разрешению синтаксических неоднозначностей входного языка.

Ввиду большей "синтаксической" ориентации данной операции ее существо будет подробно описано в разделе 2.4 следующей главы.

Заключение. Выводы главы 1

В предыдущем разделе были определены операции, модифицирующие лексическую структуру языка Си++. Эти модификации, не изменяя "лексическую семантику" языка и не усложняя общую структуру компилятора, позволяют:

- Существенно упростить синтаксис языка. Это дает повышение эффективности синтаксического анализатора как в случае его построения некоторым генератором (за счет сокращения решающих таблиц и числа состояний анализатора), так и в случае "ручной" реализации (за счет сокращения кода анализатора).
- Обеспечивают разрешение синтаксических неоднозначностей за счет явной идентификации имен. Это обстоятельство будет проиллюстрировано в разд 2.4

С учетом операций, определенных в разд. 1.5, выходной алфавит лексического анализатора языка Си++ можно описать следующим образом:

- Лексемы (*tokens*) из базового множества, как они определены в Стандарте, *за вычетом* лексем, исключенных операциями агрегации и сепарации;
- Подмножество лексем препроцессора (*pp-tokens*);
- *Суперлексемы*, введенные операциями агрегации, сепарации и операцией динамического расширения.

Сравнение количественных характеристик лексики и синтаксиса языка Си++ после проведенных модификаций с исходными характеристиками приводятся в разд. 2.4 следующей главы.

Реализация введенных операций над лексической структурой Си++ концептуально составляет содержание дополнительной компоненты компилятора, которую можно назвать **расширенным лексическим анализатором**. С учетом этой компоненты схему организации лексической части компилятора можно представить следующим образом:

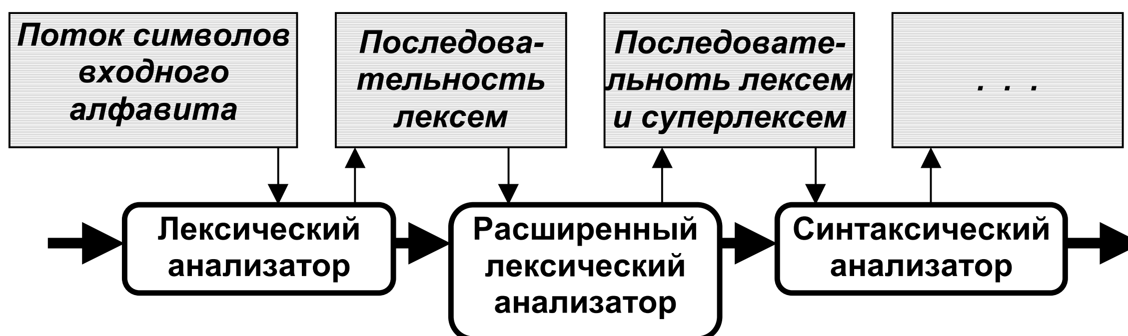

Рис. 6 Принципиальная схема организации лексического разбора

Заметим, что данная схема показывает только принципиальную организацию лексического разбора; в целях повышения эффективности этапы лексического и расширенного лексического анализа могут быть полностью или частично совмещены.

Основные **результаты** первой главы могут быть сформулированы следующим образом.

1. На основе анализа соответствующих положений Стандарта предложена концепция **непрерывной компиляции** программ Си++, в противоположность традиционному решению о последовательных стадиях компиляции. Проведено обоснование большей эффективности данной схемы при отсутствии концептуального усложнения общего дизайна компиляции.
2. В рамках концепции непрерывной компиляции рассмотрены фазы, относящиеся к препроцессорной обработке исходных программ. Вместо отдельных понятий "лексема препроцессора" (*pp-token*) и "лексема" (*token*), введенных в Стандарте и соответствующих различным фазам компиляции, предложено единое понятие

лексемы. Показывается функциональная эквивалентность двух исходных и нового понятий.

3. Проведен подробный анализ лексической структуры языка Си++ и на основе этого анализа введено понятие **суперлексемы**. Рассмотрены виды суперлексем и результирующая лексическая структура языка с учетом нового понятия. Основной вывод заключается в концептуальном упрощении лексической структуры и большей эффективности ее программной реализации.
4. На основе введенных понятий описывается принципиальная схема реализации лексической части компилятора. Делается вывод о том, что предложенная схема удовлетворяет требованиям, сформулированным в начале главы, и обеспечивает достаточно высокую эффективность первичной обработки исходных текстов.