Глава 2
Синтаксический анализ языка Си++

В настоящей главе обсуждаются принципы и методы реализации синтаксического разбора, воплощенные в компиляторе переднего плана Си++. На основе анализа синтаксической структуры языка Си++, а также целей и конкретных обстоятельств разработки выдвигаются и обосновываются требования, которым должна удовлетворять синтаксическая часть компилятора. Далее описывается подход, положенный в основу реализации, который заключается в использовании метасинтаксических средств (генератора синтаксических распознавателей). Анализируются важнейшие свойства синтаксической структуры Си++, повлиявшие на реализацию, а также особенности синтаксиса (большой объем и неоднозначность), потребовавшие специальных решений.

2.1 Введение. Общая структура компилятора

Прежде чем переходить к дальнейшим обсуждениям, необходимо сделать несколько замечаний относительно общего устройства реализации. Дело в том, что в процессе реального проектирования компилятора переднего плана проблемы организации отдельных этапов трансляции решались в комплексе с проектированием его общей архитектуры. Поэтому существо решений, принятых при реализации синтаксической части, будет более понятным в контексте более глобальных пректных решений.

В главе 1 отмечалось, что подходы, использованные при реализации фазы лексического анализа, имеют смысл для однопроходной схемы компиляции. Только в этом случае подсистема лексического анализа получает возможность доступа к структурам промежуточного представления, построенным ранее синтаксическим анализатором.

Однопроходная схема, вообще говоря, считается предпочтительной для языка Си++ прежде всего по причине ее большей эффективности; об этом говорят и создатели языка [45]. Опыт проектирования и разработки компилятора подтверждает, что начальные фазы трансляции - лексический, синтаксический и частично семантический анализ - достаточно естественно организуются в рамках однопроходной схемы. Решения, предложенные в главе 1 для фазы лексического анализа, как раз и ориентированы на такую схему. Однако сложность семантики языка не позволяет выдержать однопроходную схему полностью. Если говорить точнее, частичное выделение некоторых глобальных процедур семантического анализа в отдельный проход приводит к более ясной структуре компилятора и практически не снижает эффективности. Более подробное обсуждение этих проблем выходит за рамки настоящей работы.

Таким образом, компонентная схема компилятора переднего плана выглядит так (рисунок представляет собой конкретизацию серии аналогичных схем из главы 1; сплошные стрелки отражают управляющие связи между компонентами, двойные стрелки - потоки данных).

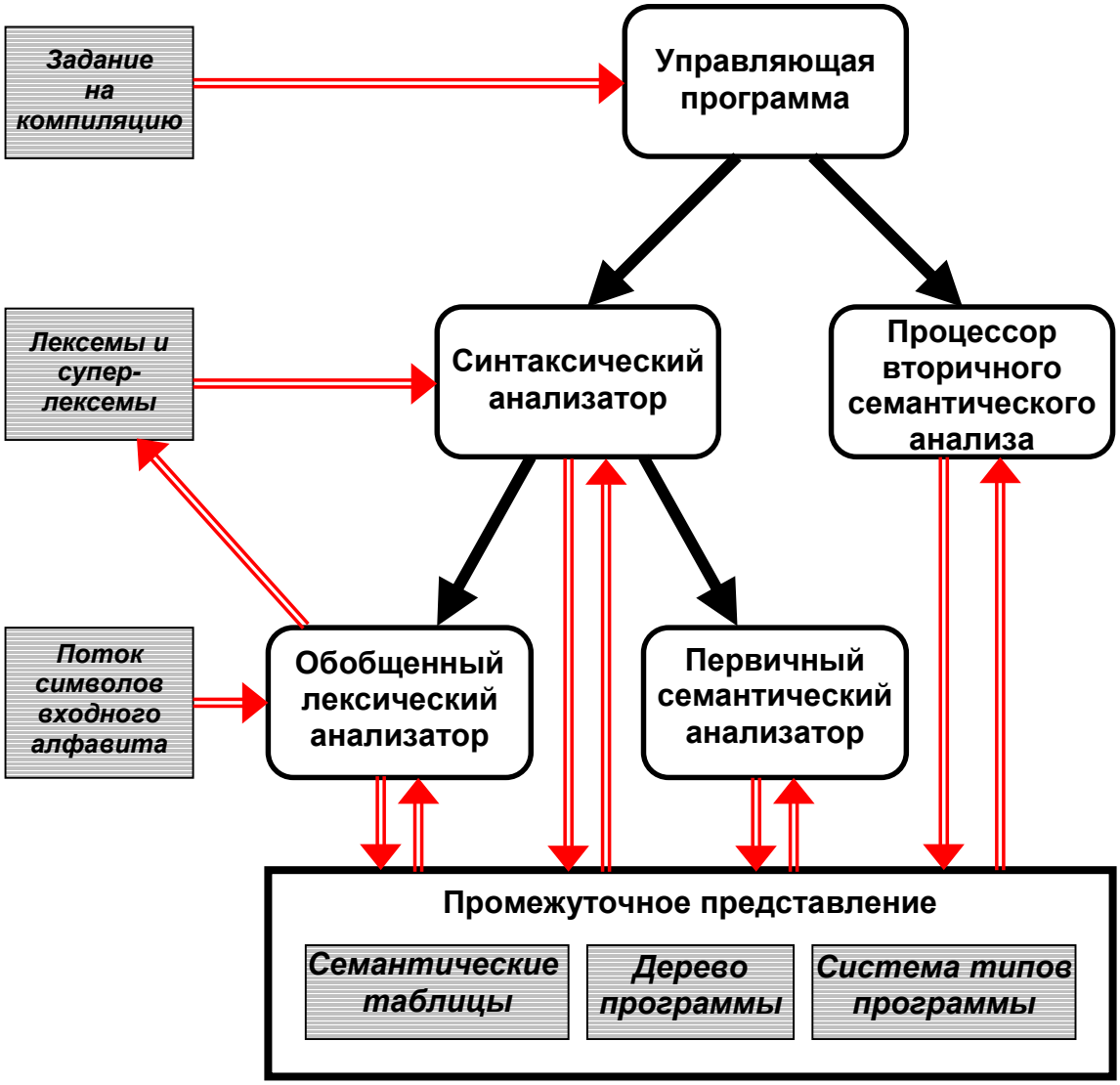


Рис. 12. Структурная схема компилятора переднего плана

Взаимодействие компонент лексического, синтаксического и первичного семантического анализа в компиляторе организовано по подпрограммному принципу. Центральную роль на этом этапе играет процессор синтаксического анализа. Вызывая обобщенный лексический анализатор (в его состав входят первичный и расширенный лексические анализаторы, см. Главу 1, заключение), синтаксический анализатор получает от него очередную лексему. В процессе идентификации лексемы может происходить обращение к промежуточному

представлению единицы трансляции. В этом случае лексема или суперлексема, получаемая синтаксическим анализатором, снабжается некоторыми атрибутами, характеризующими семантику обозначаемого ею имени.

Синтаксический анализатор выполняет сопоставление последовательности получаемых им лексем и суперлексем тем или иным синтаксическим конструкциям языка Си++. В случае успешного сопоставления вызывается соответствующая найденной конструкции процедура первичной семантической обработки. Такие процедуры, в частности, пополняют и модифицируют промежуточное представление программы.

Вторичный семантический анализатор ("второй проход") вызывается в случае успешного завершения первого этапа компиляции и работает по построенному дереву программы. На втором проходе выполняется ряд модификаций этого дерева, реализацию которых по тем или иным причинам оказалось удобным отделить от этапа синтаксического анализа. Это относится, в частности, к алгоритмам поиска наилучшей подходящей функции ("best-matching", [Std, глава 13]), настройки шаблонов функций и т.п.

Управляющая программа производит интерпретацию задания на компиляцию, включая открытие исходной единицы трансляции и установку режимов, инициализацию структур компилятора и их закрытие или сохранение при завершении его работы, а также определяет порядок и необходимость вызова различных компонент компилятора.

2.2 Интерфейс фазы синтаксического анализа и требования к его реализации

На основе общей схемы компилятора, представленной в предыдущем разделе, обсудим место синтаксического анализа в структуре компилятора, а также основные информационные потоки этого этапа. Кроме того, в разделе формулируются важнейшие требования, предъявленные к реализации синтаксического анализатора.

Одним из основных требований является возможность сравнительно легких и безболезненных модификаций компилятора при изменении синтаксиса языка в процессе его стандартизации.

В соответствии с определением порядка обработки единицы трансляции [Std, 2.1, §1], синтаксический разбор составляет содержание седьмой и восьмой фаз (см. также раздел 1.1). Согласно решениям, принятым при реализации лексического разбора (см. разд. 1.5.3), совокупным результатом предыдущих семи фаз трансляции является последовательность *лексем* из базового множества и *суперлексем*, введенных посредством операций, которые обсуждались в предыдущей главе. Часть лексем из базового множества была исключена операциями агрегации и сепарации и не попала в результирующий набор.

Лексемы и суперлексемы концептуально представляют собой агрегаты, которые содержат информацию, извлеченную из исходного

текста. Кроме условного кода лексемы/суперлексемы и ее координат по исходному тексту, они могут содержать некоторую информацию семантического характера, полученную операциями сепарации и динамического расширения.

Напомним, что одной из особенностей организации фазы лексического анализа является выполнение действий по идентификации имен. Такие действия составляют основное содержание операции сепарации. Так как сепарация идентификатора (отнесение его к определенному виду) связана с поиском его определяющего вхождения в таблицах трансляции, семантические атрибуты суперлексем, как правило, имеют вид ссылок в эти таблицы. То же относится и к операции динамического расширения, так как она определена на множестве идентификаторов.

Результирующие структуры (промежуточное представление), получаемые на выходе синтаксического анализатора, схематически показаны на рис. 12. Помимо семантических таблиц, которые являются совокупным результатом работы всех компонент компилятора, синтаксический анализатор строит дерево программы и представление системы типов.

Принципы построения **семантических таблиц** компилятора подробно обсуждаются в последующих главах. Здесь необходимо только отметить, что таблицы содержат атрибуты всех программных сущностей, явно или неявно объявленных в единице трансляции, а также информацию о контекстных зависимостях областей действия этой единицы. Кратко рассмотрим две другие компоненты результирующего ПП.

Дерево программы можно считать образом "исполняемых" фрагментов исходной единицы трансляции - операторов и их агрегатов (тел функций, инициализаторов переменных, ctor-инициализаторов). Заметим, что объявления (формально являющиеся *операторами-объявлениями* [Std, 6.7]) также могут рассматриваться как исполняемые фрагменты, поэтому дерево включает также и образы объявлений. Таким образом, знания о сущностях единицы трансляции распределены между таблицами и деревом; при этом дублирования информации не происходит, так как в таблицах сосредоточены семантические атрибуты сущностей (имя, тип, класс памяти и т.п.), а в дереве - информация о месте объявления среди других конструкций единицы трансляции, а также выражение-инициализатор (если оно имеется).

В целом дерево программы строится по традиционным принципам и образуется из узлов (вершин), соответствующих элементам синтаксических конструкций языка, и связей между узлами (ребер), которые отражают правила построения той или иной конструкции. В зависимости от структуры конретной конструкции она представляется унарным, бинарным или тернарным деревом; конструкции с неопределенным числом составляющих элементов (например, списки) отображаются в поддеревья с соответствующим числом вершин. В случае сложных конструкций исходной единицы трансляции каждый элемент дерева может, в свою очередь, представлять собой

(под)дерево. Реализация дерева сделана таким образом, что поддеревья для всех синтаксических конструкций строятся по единым правилам; это существенно упрощает семантические процедуры, выполняющие рекурсивные обходы деревьев.

Более подробное рассмотрение вопросов, связанных с деревом программы, выходит за рамки настоящей работы.

Система типов единицы трансляции представляется как отдельная структура, в которой в удобном для использования виде сосредоточена информация о множестве типов, используемых в этой единице, а также об отношениях между ними. Необходимость такой структуры непосредственно вытекает, во-первых, из весьма развитых возможностей Си++ по конструированию типов произвольной природы и, во-вторых, из многочисленных и нетривиальных правил языка, касающихся согласования различных типов (эквивалентность, совместимость, возможность преобразования одного типа к другому, различные виды преобразований, возможность преобразований по умолчанию и т.д.).

В целом представление системы типов организовано в виде таблицы, доступ к которой осуществляется посредством системы специально подобранных хеш-функций. Это обеспечивает единственность вхождения каждого типа в таблицу (включая их подтипы). В результате операции над типами (в частности, их структурный и сравнительный анализ) выполняются очень эффективно; большинство операций сводится к сравнению уникальных ключей (хеш-значений) типов. Принципы построения системы типов для Си++ и реализация представления типов подробно рассматриваются в диссертационной работе [59].

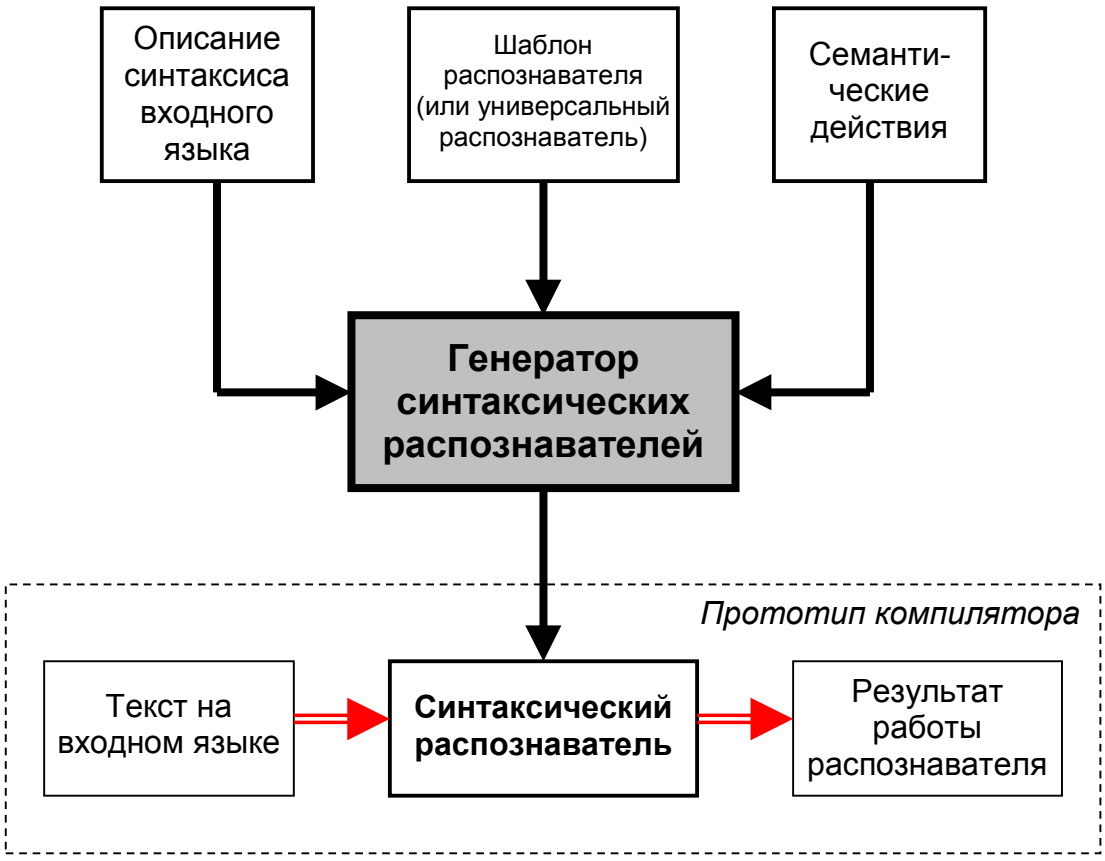
2.3 Основной принцип реализации синтаксического анализатора

Данный раздел содержит обоснование одного из основных проектных решений, принятых при реализации синтаксического анализатора,- использование метасинтаксических средств. Кратко рассмотрим исходные предпосылки такого решения.

Во-первых, анализ ситуации, сложившейся к моменту начала работы по определению принципов реализации Си++ (1993 год), показывал, что в процессе стандартизации языка происходит активная и в ряде случаев очень существенная его модификация. Это касалось как семантики отдельных конструкций Си++, так и его синтаксиса. В качестве примеров нововведений, последовавших как до, так и после начала работы, можно упомянуть механизм пространств имен, развитие понятия исключительных ситуаций (отнесение *ctor-инициализаторов* конструкторов под сферу действия *блоков-с-контролем*), принципиальное расширение возможностей механизма шаблонов, а также многочисленные менее масштабные модификации. Заметим, что процесс совершенствования языка, сопровождавшийся изменениями его синтаксической структуры, продолжался вплоть до принятия Стандарта Си++ (конец 1998 года).

В условиях, когда разработка компилятора переднего плана происходит, по существу, параллельно с процессом стандартизации языка, было крайне важно обеспечить оперативное отражение в реализации возникающих изменений без снижения темпов разработки и без риска внесения ошибок в программный текст. Напомним, что одним из основных требований к реализации (см. главу 1) являлась полное соответствие Стандарту.

Такую возможность предоставляет подход к реализации синтаксической части компилятора, основанный на использовании генератора синтаксических анализаторов. Эта технология позволяет по описанию синтаксиса входного языка на некотором метаязыке получить синтаксический распознаватель, воспринимающий тексты, синтаксис которых удовлетворяет исходному формальному описанию. Связывание распознавателя с алгоритмами семантического анализа производится посредством подпрограмм ("семантических действий"), сопоставляемым тем или иным фрагментам синтаксиса. Схематически процесс создания синтаксических анализаторов можно представить следующей схемой:



Как видно из представленной схемы, данный подход позволяет разрабатывать и модифицировать синтаксическую и семантическую компоненты компилятора достаточно независимо друг от друга, что дает возможность оперативно отражать текущее состояние входного языка в процессе его совершенствования. Кроме того, использование генератора снимает проблему "ручного" программирования синтаксического

анализатора, что само по себе может представлять весьма сложную проблему для больших языков класса Си++ и Ada. Наконец, явное разделение компонент компилятора позволяет организовать их параллельную разработку; спецификация "синтаксической части" компилятора на некотором формальном метаязыке является удобным средством повышения его обозримости, читабельности и, в конечном счете, надежности.

К недостаткам описанного подхода следует отнести сложности, неизбежно возникающие при реализации дополнительных интерфейсов между компонентами, необходимость учета особенностей используемого генератора, а также несколько меньшую эффективность результирующего распознавателя по сравнению с анализатором, реализованным вручную.

Долгое время технология создания распознавателей для ЯП на основе использования метасинтаксических средств не выходила за рамки академических или экспериментальных проектов. Однако к настоящему времени имеются распознаватели промышленного уровня, эффективность и надежность которых подтверждена достаточно долгим сроком их успешного использования. Речь идет о семействе "компиляторов компиляторов" YACC, в которое, помимо одноименной программы, входит генератор Bison [40], а также их многочисленные модификации. Программа YACC [38] является стандартной утилитой в системе UNIX; Bison представляет собой компоненту свободно-доступной системы GNU, распространяется в исходных текстах и может использоваться независимо практически на любой прграммно-аппаратной платформе. Различные версии этих программ либо включают дополнительные возможности (например, поддержку атрибутных грамматик), либо используют в качестве базового не язык Си, а Си++, Pascal, Ada и пр.

Наиболее распространенным и часто используемым на практике и, следовательно, самым апробированным и надежным из семейства является генератор Bison. С другой стороны, "габариты" грамматики Си++ (большое число терминалов, нетерминалов и синтаксических правил в формальном описании языка) превышают возможности оригинальной утилиты YACC, а также некоторых других версий распознавателей. По этим причинам для реализации синтаксической части компилятора переднего плана был выбран Bison. Дополнительным аргументом в пользу выбранного генератора служит его использование в проекте GNU для формирования синтаксической части компилятора Си++ (в разделе 2.4 приводятся некоторые сравнительные характеристики формальных описаний Си++ в компиляторе GNU и в обсуждаемом компиляторе переднего плана).

Программный интерфейс, алгоритм синтаксического разбора, положенный в основу распознавателей, а также внутреннее устройство генераторов семейства YACC очень близки (вплоть до совместимости). Генератор воспринимает на входе описание грамматики языка на некотором формальном метаязыке, транслирует его во внутреннее табличное представление и объединяет с заготовкой универсального

распознавателя и описаниями семантических действий. Результатом работы генератора в общем случае служит программа на языке Си.

Таблицы, построенные генератором, имеют вид описаний константных массивов на языке Си; заготовка распознавателя также представляет собой текст на этом языке. Семантические действия задаются непосредственно в описании грамматики и имеют вид составных операторов языков Си или Си++, сопоставляемых тому или иному синтаксическому правилу.

Синтаксический анализатор, который служит ядром создаваемой генератором программы, способен воспринимать контекстно-свободную грамматику класса LALR(1) и реализует восходящий разбор текста в алфавите этой грамматики методом "перенос-свертка" [41]. Операторы, реализующие семантические действия, имеют доступ к стеку разбора; тем самым YACC-подобный распознаватель поддерживает механизм синтезируемых атрибутов, вычисление которых определяется в терминах верхних элементов стека разбора. Структура "атрибутной" части стека специфицируется разработчиком в формальном описании грамматики, что дает возможность задавать произвольный тип его элементов в терминах языка реализации (Си или Си++).

Формальное описание языка на метаязыке YACC формируется в виде совокупности правил (продукций), каждое из которых сопоставляет некоторому нетерминальному символу последовательность терминальных и/или нетерминальных символов. Каждый нетерминальный символ обозначают некоторое правило (допускается рекурсивное вхождение нетерминалов); терминальные символы обозначают лексемы входного языка.

Интерфейс распознавателя с лексическим анализатором специфицируется посредством функции `yylex()`, которая должна быть задана разработчиком. Распознаватель оперирует с целочисленными кодами лексем, которые, посредством данной функции, должен возвращать лексический анализатор. При необходимости семантические атрибуты лексем могут передаваться через вершину семантического стека, доступ к которой входит в интерфейс распознавателя.

Ниже в качестве примера (в несколько упрощенном виде) приводится фрагмент формального описания конструкции *object-declaration* языка Си++ [Std, Глава 7, §1] на метаязыке YACC, взятый из реализации компилятора переднего плана. Собственно синтаксические правила записываются прямым шрифтом, спецификации семантических действий - курсивом.

```
//-----------------------------
//     ObjectDeclaration
//-----------------------------

SimpleDeclaration :                  lxmSemiColon
                                       { $$.Node  = NULL;
                                         $$.Specs = EmptySpecs(); }

                  | DeclSpecifierSeq lxmSemiColon
                                       { NewTypeDecl($1.Specs);
                                         $$ = $1; }

                  | DeclList         lxmSemiColon
                                       { $$ = $1; }

                  | error            lxmSemiColon
                                       { $$.Node  = NULL;
                                         $$.Specs = EmptySpecs(); }
                  ;

DeclList          : DeclSpecifierSeq    Declarator  Initializer0

                      { NODE* Node = NewObject($1.Specs,$2,
                                               $3.Node,$3.Mode);
                        if ( $1.Node != NULL )
                            $$.Node = AddNode($1.Node,Node);
                        else
                           $$.Node = AddNode(CreateDeclNode(),Node);
                        $$.Specs = $1.Specs;
                        RemoveScopes();   }

                  |                   Declarator  Initializer0

                      { NODE* Node = NewObject(EmptySpecs(),$1,
                                               $2.Node,$2.Mode);
                        $$.Node = AddNode(CreateDeclNode(),Node);
                        $$.Specs = EmptySpecs();
                        RemoveScopes();   }

                  | DeclList  lxmComma  Declarator  Initializer0

                      { NODE* Node = NewObject($1.Specs,$3,
                                               $4.Node,$4.Mode);
                        $$.Node = AddNode($1.Node,Node);
                        $$.Specs = $1.Specs;
                        RemoveScopes();  }
                  ;
```

Левые и правые части правил разделяются двоеточием. Для большей наглядности правила с одинаковой левой частью могут объединяться в одно правило с несколькими альтернативами, которые в этом случае отделяются друг от друга вертикальной чертой. Терминалы и нетерминалы задаются в традиционной нотации идентификаторов (для большей наглядности терминальные символы начинаются с префикса `lxm`). Семантические действия в виде составных операторов Си/Си++ записываются справа от правила (или после каждой альтернативы правила). Доступ к верхним элементам стека разбора (или, что то же самое, к атрибутам, синтезированным для составных частей правила) производится посредством обозначений вида `$N`, где `N` - целое число,

обозначающее порядковый номер соответствующего элемента правила. Атрибут, синтезируемый правилом, обозначается как `$$`.

При задании правил допускается левая рекурсия. Так, чтобы задать список описателей в объявлении, правило `DeclList` определяется рекурсивно (в YACC-системах отсутствуют явные средства задания повторяющихся частей правил).

Генератор Bison содержит встроенные средства нейтрализации синтаксических ошибок. В примере псевдотерминал **`error`**, используемый в правиле `SimpleDeclaration`, сопоставляется с любой синтаксически некорректной последовательностью вплоть до ближайшей точки с запятой (обозначенной посредством терминала `lxmSemiColon`). Тем самым обеспечивается автоматическое восстановление разбора, начиная с этого символа.

В заключение данного раздела отметим, что выбранный подход к реализации синтаксического анализа в компиляторе переднего плана полностью себя оправдал. Практика создания компилятора показала, с одной стороны, несомненную полезность разделения синтаксического описания языка и процедур семантического анализа как в плане повышения обозримости текста реализации, так и в смысле легкости и удобства внесения изменений в синтаксисическую и семантические части компилятора. Кроме того, использование в качестве ядра синтаксического анализатора компоненты апробированной в мировой практике системы Bison (которая является в настоящее время стандартом де-факто) существенно снизило затраты на реализацию.

Вместе с тем, интеграция распознавателя в компилятор переднего плана потребовала учета в компиляторе некоторых особенностей поведения этого распознавателя. Эти особенности проявляются (становятся существенными) только в случае использования Bison для очень больших и сложных входных языков; подробный анализ этих особенностей выходит за рамки настоящей работы. Кратко проблему можно охарактеризовать следующим образом. Усложнение лексического анализатора, в частности, перенос действий по идентификации имен на этап лексического анализа (об этом говорилось в главе 1) приводит к тому, что процедуры вычисления лексем могут сопровождаться побочными эффектами, воздействующими на состояние глобальных структур трансляции. В то же время семантические действия, вызываемые распознавателем автоматически при свертке очередной основы в стеке разбора, также изменяют эти структуры. Тем самым становится необходимым точный учет *относительного порядка* вычисления той или иной лексемы (суперлексемы) и выполнения ближайшего правого семантического действия. Решение описанной проблемы и составило основную сложность при реализации синтаксической части компилятора переднего плана на основе генератора Bison.

2.4 Операция динамического расширения и механизм состояний

Данный раздел посвящен рассмотрению третьей операции по модификации множества лексем, полученных в результате операции сепарации (см. разд. 1.5.6). Эта операция введена в главе 1 как некоторая абстракция действий лексического анализатора по разрешению синтаксических неоднозначностей входного языка.

Вначале кратко суммируем логику рассмотрений разд. 1.5. Обычно лексический анализатор осуществляет низкоуровневое (контекстно-независимое) расчленение исходного текста программы на элементарные лексические единицы и поставляет их синтаксическому анализатору, на который ложится основная тяжесть работы по семантической идентификации лексем, агрегации и статической интерпретации синтаксических конструкций. Особенности синтаксиса Си++ не позволяют выдержать такой подход в полном объеме; в связи с этим этап лексического анализа был дополнен действиями по индентификации имен, обычно свойственными этапу семантического анализа (операция сепарации). В данном разделе продолжим обсуждение с точки зрения этапа синтаксического анализа.

Чтобы решить проблемы, связанные с синтаксической неоднозначностью входного языка, необходимо прежде всего идентифицировать семантическую принадлежность имен. Эти действия реализуются операцией сепарации. Чтобы разрешить неоднозначность (которая, заметим, не может быть разрешена собственным алгоритмом анализатора), следует предпринять специальные усилия.

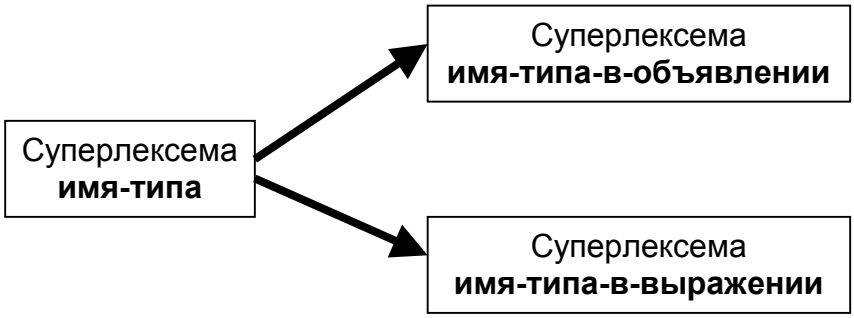
В разд. 1.5.5 как пример неоднозначности рассматривалась конструкция вида ***X*** (***Y***). Пусть в результате сепарации идентификатор ***X*** был опознан как обозначающий некоторый тип и заменен на одну из суперлексем из числа указанных выше. Теперь следует определить синтаксическую принадлежность всей конструкции.

К сожалению, синтаксис Си++ устроен таким образом, что для разрешения неоднозначности в общем случае необходимо проанализировать всю конструкцию (оператор или объявление) целиком. В работе [45] приводится краткое рассмотрение возможной схемы такого разрешения. Оно основано на выделении ближайшего контекста неоднозначности (в нашем примере это сама конструкция ***X*** (***Y***)) и цепочки последующих лексем. Если при достижении закрывающей круглой скобки разрешить неоднозначность не удается, то приходится анализировать "хвост" конструкции.

Эта схема реализована в обсуждаемом компиляторе со следующими особенностями. При обнаружении контекста неоднозначности (вхождения имени типа с последущей левой скобкой) лексический анализатор производит просмотр последующих лексем вперед до тех пор, пока вся конструкция не будет отнесена к тому или иному виду. Как только вид конструкции установлен, лексический анализатор *корректирует первоначальную лексему* в зависимости от вида конструкции. Так, если исходная суперлексема обозначала некоторое ***имя-типа***, а вся конструкция разрешена как оператор, то

лексический анализатор возвращает скорректированную суперлексему вида ***имя-типа-в-выражении*** с теми же атрибутами, что и исходная. Если конструкция идентифицирована как объявление, то возвращается первоначальная суперлексема.

Эти действия и составляют существо операции **динамического расширения**. Лексический анализатор, в зависимости от текущего контекста, динамически изменяет вид результирующей лексемы. В результате суперлексема, полученная в результате применения операции сепарации к имени (простому или квалифицированному идентификатору) типа, может отображаться двояко. Схематически результат операции динамического расширения можно изобразить следующим образом:



Имея в распоряжении суперлексемы, относящиеся к различным случаям вхождения *имен-типов*, можно заметно упростить формальное описание синтаксиса языка.

Заметим, что в некоторых случаях операция динамического расширения не нуждается в процедуре разрешения неоднозначности. Это относится к контекстам, в которых объявление появиться не может. Примером служат выражения-инициализаторы в объявлениях объектов, выражение в операторе `return`, аргумент `throw`-выражения и т.д. Таким образом, любое вхождение *имени-типа* в таких контекстах реально не приводит к неоднозначности и может быть сразу расширено до *имени-типа-в-выражении*. Чтобы повысить эффективность анализа, сократив излишние обращения к процедуре разрешения неоднозначности, распознаватель передает лексическому анализатору информацию о текущем синтаксическом контексте компиляции.

Для целей оптимизации динамического расширения лексическому анализатору достаточно информации вида "выражение либо объявление". Однако реализация сепарации идентификаторов в полном объеме требует более подробной информации. С другой стороны, детальный анализ текущего синтаксического контекста неизбежно потребует дополнительных затрат, поэтому необходимо выбрать некоторый оптимальный вариант, обеспечивающий, с одной стороны, эффективное распознавание имен и, с другой стороны, не приводящий к нетривиальным методам анализа и не усложняющий распознаватель.

В результате экспериментов с компилятором переднего плана и в процессе его тестирования было зафиксировано следующее множество состояний, детектируемых синтаксическим анализатором:

- **выражение**: контекст, в котором допускается появление как выражений (операторов), так и объявлений. Заметим, что только это состояние требует активации механизма разрешения неоднозначности;
- **только выражение**: контекст, в котором допускаются только выражения;
- **объявление**: распознаватель обрабатывает объявление программной сущности;
- **класс**: частный случай состояния "объявление", когда распознаватель обрабатывает объявления членов класса;
- **ctor-инициализатор**: распознаватель обрабатывает список ctor-инициализаторов конструктора, но не сами инициализирующие выражения (учет такого контекста позволяет эффективно отобразить в грамматике ctor-инициализаторов имена членов классов и имена непосредственных базовых классов);
- **шаблон**: распознаватель находится в списке формальных параметров объявления шаблона.

"Обратное" направление передачи информации о текущем контексте - от синтаксического анализатора лексическому наиболее эффективно реализуется посредством введения некоторого глобального контекста, доступного этим двум компонентам. Такое решение нельзя считать бесспорным, так как оно увеличивает информационную взаимозависимость компонент; однако относительно тесная интеграция двух компонент компилятора представляется приемлемой платой за снижение объема и сложности грамматики и достаточно эффективный механизм разрешения неоднозначностей.

В каждый момент распознаватель может находиться в одном из перечисленных состояний, которые представляются в виде значений глобальной переменной компилятора, доступной всем его компонентам. Переключение из одного состояния в другое реализуется в семантических действиях.

В заключение данного раздела необходимо кратко коснуться одного аспекта разрешения неоднозначностей, который связан с просмотром вперед. Существо разрешения в том виде, как оно реализовано в компиляторе, состоит в том, что необходимо идентифицировать некоторую лексему на основе анализа неопределенного количества последующих лексем. После того, как разрешение произведено, лексический анализатор должен вернуться к "стандартному" режиму работы, начиная с лексемы, непосредственно идущей за начальной. Чтобы избежать повторного вычисления пройденных при разрешении лексем, анализатор в процессе разрешения сохраняет пройденные лексемы в специальном буфере (его принципиальная организация обсуждалась в разд. 1.4). После возвращения в "стандартный" режим

анализатор сначала выбирает вычисленные лексемы из буфера, а после его исчерпания переходит к регулярному извлечению лексем из входного текста.

Таким образом, процесс разрешения синтаксических неоднозначностей практически не снижает общей эффективности: во-первых, из-за того, что он активируется только в тех случаях, когда неоднозначность возможна, и, во-вторых, благодаря исключению повторных действий по вычислению лексем при просмотре вперед.

Подводя итог обсуждениям принципов модификации исходного синтаксиса языка, приведем результаты сравнения объемов и структуры формального описания синтаксиса Си++ в компиляторе переднего плана и аналогичного описания, используемого в свободно-распространяемом компиляторе GNU [60]. Такое сравнение показательно прежде всего тем, что в реализации компилятора GNU используется в некотором смысле противоположный подход: в формальном описании языка отображены *все* синтаксические особенности Си++, в результате чего это описание приобрело очень громоздкий, нечитаемый текст, изобилующий многочисленными неочевидными решениями и откровенными программистскими "трюками".

Параметры грамматики Си++	**GCC**	**FE C++**
Количество синтаксических правил	779	474
Количество именованных терминалов	82	117
Количество неименованных терминалов	24	*нет*
Всего терминалов	106	117
Число состояний анализатора	1339	708
Число нетерминалов	202	205
Число неявных терминалов	53	4

Как видно из приведенной таблицы, при небольшом увеличении числа терминальных символов грамматики получено очень существенное сокращение количества синтаксических правил, а также количества состояний анализатора. Именно две последние характеристики влияют на размеры таблиц, генерируемых программой Bison, на внутренний алгоритм распознавателя и, тем самым, определяют быстродействие синтаксического анализатора.

2.5 Разбор выражений языка Си++

В данном разделе мы рассмотрим некоторые дополнительные решения по реализации синтаксического анализатора, направленные прежде всего на повышение его эффективности.

Предлагаемое проектное решение составляет некоторый компромисс с исходными требованиями. Как говорилось выше,

формальное описание синтаксиса языка Си++, по которому генератор Bison строит синтаксический распознаватель, проектировалось с минимально возможными отступлениями от синтаксических правил, предложенных в Предварительном Стандарте. Прежде всего это диктовалось необходимостью поддерживать текущую версию языка в процессе его стандартизации. Однако в одном существенном аспекте было сделано исключение от этого принципа. Речь идет о синтаксисе выражений Си++.

Проблема заключается в том, что точное следование синтаксису, данному в Стандарте, приводит к весьма объемному описанию (около трети текста на входном метаязыке Bison). Однако более существенным недостатком традиционного описания представляется неэффективность определяемого им алгоритма синтаксического распознавателя.

Рассмотрим (в несколько упрощенном виде) фрагмент грамматики Си++ [Std, Приложение А], определяющий структуру выражений языка.

```
primary-expression:
            literal                 // литерал
            id-expression           // имя переменной
            ( expression )
      . . .

multiplicative-expression:
                                              pm-expression
            multiplicative-expression     *     pm-expression
            multiplicative-expression     /     pm-expression
            multiplicative-expression     %     pm-expression

additive-expression:
                                          multiplicative-expression
            additive-expression       +   multiplicative-expression
            additive-expression       -   multiplicative-expression

shift-expression:
                                          additive-expression
            shift-expression          <<  additive-expression
            shift-expression          >>  additive-expression

relational-expression:
                                          shift-expression
            relational-expression     <   shift-expression
            relational-expression     >   shift-expression
            relational-expression     <=  shift-expression
            relational-expression     >=  shift-expression

equality-expression:
                                          relational-expression
            equality-expression       ==  relational-expression
            equality-expression       !=  relational-expression

and-expression:
                                          equality-expression
            and-expression            &   equality-expression

exclusive-or-expression:
                                          and-expression
            exclusive-or-expression   ^   and-expression
```

```
inclusive-or-expression:
                                        exclusive-or-expression
            inclusive-or-expression |   exclusive-or-expression

logical-and-expression:
                                        inclusive-or-expression
            logical-and-expression  &&  inclusive-or-expression

logical-or-expression:
                                        logical-and-expression
            logical-or-expression   ||  logical-and-expression

expression:
            logical-or-expression
            logical-or-expression assignment-operator assignment-expression

assignment-operator: one of
            =   *=   /=   %=   +=   -=   >>=   <<=   &=   ^=   |=
```

Точное отображение синтаксиса, заданного приведенными правилами, в формальное описание для генератора Bison, не представляет сложностей. Достоинства такого точного отображения исходной грамматики заключаются, во-первых, в том, что гарантируется соответствие формального описания данному в Стандарте. Вторым преимуществом является отсутствие необходимости задания приоритетов операций, так как сама структура правил для выражений обеспечивает относительное старшинство операций. Заметим, что в Стандарте приоритеты операций в выражениях не фиксируются (на этот счет имеется специальная оговорка); приоритеты могут быть выведены непосредственно из грамматики выражений.

В то же время генератор, порожденный по такому описанию, будет заведомо неэффективен. Так, для простого выражения, состоящего из единственного идентификатора переменной, генератор будет последовательно помещать в стек разбора основы, соответствующие всей цепочке правил от `expression` до `primary-expression`. Поэтому семантический атрибут лексемы `xlxmIdExpression` (которая в реализации соответствует нетерминалу `id-expression`), вычисленный лексическим анализатором (этот атрибут может представлять собой, например, узел будущего дерева выражения), будет перезаписываться в вершину стека разбора при свертке каждой основы из цепочки правил.

Неформально этот процесс можно описать следующим образом. Пусть по текущему контексту разбора в данной позиции исходного текста ожидается конструкция `expression`. Синтаксический анализатор обращается за очередной лексемой, получая `xlxmIdExpression`. Семантический атрибут лексемы помещается в вершину стека разбора. Далее анализатор выполняет серию сверток: `xlxmIdExpression`, согласно описанию грамматики, сворачивается к (заменяется в стеке разбора на) `primary-expression`, далее сводится к `multiplicative-expression`, `additive-expression`, сводясь в конечном итоге к `expression`. При каждой свертке стек разбора реорганизуется; это, в частности, означает взятие из вершины стека семантического атрибута исходной лексемы, удаление из стека

очередной основы и повторное помещение этого атрибута в вершину стека.

Существенно, что описанные действия будут выполняться независимо от сложности входного выражения. В частности, обработка наиболее частых случаев выражений, состоящих из одного идентификатора будут всегда сопровождаться описанными действиями анализатора.

В качестве альтернативы можно предложить более эффективную схему, которая использует механизм приоритетов, имеющийся в генераторе. Идея предлагаемого решения заключается в том, чтобы реорганизовать исходную грамматику выражений, сведя ее к единственному рекурсивному правилу `GeneralExpression` со многими альтернативами. Схематически преобразованная таким образом грамматика может выглядеть примерно так (текст в несколько упрощенном виде взят из формального описания Си++, используемого в компиляторе):

```
GeneralExpression : GeneralExpression
                      AssignmentOperator
                        GeneralExpression       %prec PrecAssignment
                  | GeneralExpression
                      '?' ExpressionList ':' GeneralExpression
                                                %prec PrecCondition
                  | GeneralExpression '||' GeneralExpression
                                                %prec PrecLogicalOr
                  | GeneralExpression '&&' GeneralExpression
                                                %prec PrecLogicalAnd
                  | GeneralExpression '|' GeneralExpression
                                                %prec PrecBitwiseOr
                  | GeneralExpression '^' GeneralExpression
                                                %prec PrecBitwiseXor
                  | GeneralExpression '&' GeneralExpression
                                                %prec PrecBitwiseAnd
                  | GeneralExpression '==' GeneralExpression
                                                %prec PrecEqual
                  | GeneralExpression
                      RelationalOperator
                         GeneralExpression      %prec PrecRelation
                  | GeneralExpression
                      ShiftOperator
                         GeneralExpression      %prec PrecShift
                  | GeneralExpression
                      AddOperator
                         GeneralExpression      %prec PrecAdd
                  | GeneralExpression
                      MultOperator
                         GeneralExpression      %prec PrecMultiply
                       . . .
                  | xlxmPrimaryExpression    { $$ = $1; }
                  |  lxmThis                 { $$ = $1; }
                  | xlxmIdExpression         { $$ = $1; }

                  |  lxmLeftParenth
                       ExpressionList
                     lxmRightParenth         { $$ = $2; }
                  ;
```

Показанное преобразование грамматики потребовало явного введения относительных приоритетов операций. Согласно схеме, заложенной в генераторе Bison, приоритет операции задается посредством директивы `%prec` для той альтернативы правила `GeneralExpression`, которая содержит вхождение этой операции. Для задания приоритетов в этой директиве используются лексемы (либо из множества лексем исходного языка, либо "псевдолексемы", специально введенные для этих целей и не соответствующие никаким конструктам входного языка). Значение приоритета той или иной альтернативы правила соответствует относительному порядку лексемы в описаниях: лексема, описанная ниже по тексту, имеет более высокий приоритет относительно описанных выше нее.

Таким образом, выбрав подходящий порядок описания лексем (в примере - псевдолексем, имена которых начинаются с `Prec`), можно задать правила предпочтения альтернатив согласно тем приоритетам, которые неявно содержатся в оригинальной грамматике Стандарта.

Как видно из последнего фрагмента, модифицированный синтаксис выражений, несмотря на сокращение объема по сравнению с оригиналом, выглядит несколько менее обозримым. Тем не менее, непосредственное сравнение производительности анализатора, работающего по "старой" и "новой" грамматике, показало почти 20%-ное повышение эффективности. Неофициальные данные, полученные от разработчиков некоторых зарубежных компиляторов Си++, подтверждают полученную оценку.

Заметим, что все вышеприведенные рассуждения полностью справедливы и для синтаксиса объявлений Си++, по крайней мере, в пределах его Си-подмножества. Правила задания типов, используемые в языке Си, аналогичны правилам формирования выражений [45, разд. 8.2] и, тем самым, допускают аналогичные оптимизирующие преобразования. Проведение таких оптимизаций составляет заметный резерв повышения производительности синтаксической части компилятора.

Наконец, легко видеть, что синтаксис выражений в современных языках программирования строится на единых принципах (с точностью до обозначений): инфиксная форма бинарных операций с явно заданными приоритетами, дополненная префиксными и постфиксными унарными операциями с более высоким приоритетом. Поэтому в случае использования восходящих распознавателей описанная выше техника преобразований синтаксиса применима и в других подобных случаях, в частности, для синтаксического распознавателя для языка Ada. Непосредственное сравнение сложности синтаксиса выражений в Си++ и Аде (количество синтаксических правил, ассортимент операций, уровни приоритетов и т.п.) позволяет предположить, что использование оптимизированной схемы при разборе выражений в языке Ада приведет по крайней мере к такому же повышению эффективности, что и для случая Си++.

Выводы главы 2

В основу проектирования и реализации лексико-синтаксической части компилятора были положены следующие проектные решения.

1. Использование метасинтаксических средств при построении синтаксической части компилятора вместе с прямой реализацией лексических компонент.
2. Модификация традиционной схемы синтаксического разбора: перенос схем идентификации имен на этап лексического анализа.
3. Модификация синтаксиса выражений с целью повышения общей эффективности синтаксической части компилятора.

Основной вывод главы 2 заключается в подтверждении обоснованности проектных решений по использованию метасинтаксических средств при реализации этапа синтаксического разбора. Вместе с тем, эффективность получаемого распознавателя должна быть повышена системой мер, основными среди которых являются повышение "интеллектуальности" лексического анализатора и частичная реорганизация формального описания синтаксиса языка Си++ относительно данного в Стандарте.