Санкт-Петербургский государственный политехнический университет

Институт информационных технологий и управления

Кафедра компьютерных систем и программных технологий

Цыган В.Н.

Транслирующие системы

Текст лекций

Санкт-Петербург
2014 г.

В лекциях рассматриваются алгоритмы функционирования, а также принципы построения и разработки трансляторов различных типов: компиляторов, интерпретаторов, ассемблеров. Курс изучается в рамках бакалаврской подготовки в восьмом семестре. Направление подготовки – 230100 Информатика и вычислительная техника (ФГОС).

Содержание

Введение

Вопросы, подлежащие изучению.

1) Системы программного обеспечения, этапы их развития.
2) Понятие "система программирования".
3) Классификация систем программирования по различным признакам.
4) Типы трансляторов (ассемблеры, интерпретаторы, компиляторы).

Замечание: по ходу изучения данного курса будем вводить необходимые понятия и термины, каждый раз определяя их значения (содержание) в контексте именно данного курса.

В первую очередь определим понятие "системное программное обеспечение" ("системное ПО"). В публикациях можно встретить различные определения указанного термина. Преимущественно системное ПО определяется как средство для управления вычислительными процессами и/или как средство обеспечения интерфейса с пользователями. Иногда встречаются краткие, но удачные определения, например, такое – "системное ПО – это то программное обеспечение, которое поставляется вместе с компьютером". Еще одно терминологическое замечание: в рамках нашего курса лекций не будем пытаться провести строгую границу между такими понятиями, как "ЭВМ", "компьютер", "вычислительная система" и т.п., так как подобное разграничение существенно при изучении "аппаратных" дисциплин или при освоении курса, посвященного операционным системам (ОС), а при изучении транслирующих систем – не существенно.

Введем понятие "система программного обеспечения" ("система ПО"). Будем называть системой ПО совокупность программ, которые должны или могут функционировать совместно. Выделим среди систем ПО те, которые имеют отношение к системному ПО, и рассмотрим этапы их развития (см. раздаточный материал). Исторически первым типом подобных систем были библиотеки стандартных подпрограмм (БСПП). Возникновение и внедрение БСПП позволило существенно повысить эффективность использования ЭВМ в непривычных (и трудно представимых) для современного пользователя условиях отсутствия языков программирования, отсутствия операционных систем и т.п.

Исторически следующий этап развития систем ПО связывают с возникновением так называемых "транслирующих систем". Речь идет о том периоде, когда началась разработка машинно-независимых языков программирования и соответствующих трансляторов. На нашей диаграмме началу этого этапа поставлены в соответствие 1953-1954 г.г., так как именно в это время появился язык программирования FORTRAN, который, в свою очередь, был первым языком программирования, для которого де-факто возникли общепризнанные стандарты. В дальнейшем бурное развитие языков

программирования потребовало интенсивных и непрерывных разработок соответствующих трансляторов для различных типов ЭВМ, что привело к резкому увеличению объёма системного ПО.

Следующий этап развития систем ПО обычно связывают с появлением так называемых "операционных систем" ("ОС"). В рамках данного курса мы не будем пытаться строго определить понятие "операционная система", так как изучению ОС будет посвящен отдельный учебный курс, где, вероятно, и будут даны необходимые определения. Пусть пока данное понятие воспринимается на основе уже изученных дисциплин (например, дисциплин по изучению языков и технологий программирования). Заметим лишь, что первые прототипы того, что теперь принято называть операционными системами, возникли в конце 1950-х – начале 1960-х годов.

Наконец, современный этап развития систем ПО обычно связывают с разработкой ПО для последующих поколений ЭВМ. Наиболее важной тенденцией подобных разработок является все большая "интеллектуализация" ПО.

Заключая обсуждение этапов развития систем ПО, сделаем одно замечание: каждый последующий этап не отвергал результатов предыдущих этапов. Действительно, если для примера рассмотреть состав внутреннего (системного) ПО современных ЭВМ, то основные компоненты его таковы:

1) Программы общего пользования (БСПП).
2) Управляющие программы, составляющие основу ОС.
3) Транслирующие программы.
4) Обслуживающие программы (драйверы, утилиты и т.п.).

Указанный состав имеет место в так называемых ЭВМ общего назначения. В специализированных (в частности, во встраиваемых) ЭВМ состав системного ПО может отличаться от приведенного выше.

Введем понятие "система программирования". Будем называть системой программирования совокупность языка программирования и транслятора. Пусть понятие языка пока воспринимается интуитивно, а вот транслятор определим как программу, которая осуществляет перевод программ, написанных на каком-либо языке программирования в другую форму, в частности, в форму машинных команд.

Классификация систем программирования может быть осуществлена различными способами. Для обсуждения предлагаются три способа (см. раздаточный материал).

Краткие комментарии к диаграмме. При классификации систем программирования "по степени удаленности" от системы команд наиболее

приближенными к системе команд считаются так называемые машинно-ориентированные системы или, иначе говоря, ассемблеры. Отметим здесь многозначность термина "ассемблер", так в зависимости от контекста термин "ассемблер" может обозначать только машинно-ориентированный язык программирования, или только транслятор, или систему программирования в целом. Использование понятия "машинной ориентированности" применительно к ассемблерам вполне естественно, так как исполняемые операторы какого-либо языка ассемблера представляют из себя мнемонические отображения машинных команд соответствующей ЭВМ, а чаще – семейства ЭВМ.

Что касается процедурно-ориентированных систем программирования, то этот тип, вероятно, наиболее хорошо знаком слушателям настоящего курса благодаря занятиям по изучению программирования. Процедурно-ориентированные системы программирования строятся на основе так называемых императивных языков программирования. Основное свойство последних – это необходимость явного описания на языке программирования алгоритмов всех подлежащих исполнению вычислительных процедур. Процедурно-ориентированные системы программирования достаточно разнообразны по своему назначению и возможностям. Классификация подобных систем рассматривается в рекомендованной литературе. Здесь лишь кратко перечислим основные типы процедурно-ориентированных систем программирования: ориентированные на инженерно-технические расчеты; ориентированные на решение экономических задач; ориентированные на моделирование; ориентированные на обработку списковых структур данных; ориентированные на абстрактные типы данных и т.п.

В так называемых проблемно-ориентированных системах программирования не обязательно явно описывать алгоритм решения задачи. Фактически большинство операторов подобных систем представляют собой вызовы заранее составленных и отлаженных программ. Наиболее ярким примером систем указанного типа являются всевозможные системы автоматизированного проектирования (САПР).

Продолжим рассмотрение способов классификации систем программирования. Во втором способе предлагается их классифицировать по характеру использования. Наиболее известный способ – для написания программ. При этом следует выделить в отдельный подтип системы программирования, пригодные для разработки системного ПО. Что касается так называемого пользовательского ПО, то оно может быть разработано с применением системы программирования соответствующей ориентации, либо (что чаще встречается на практике) с использованием системы

программирования наиболее хорошо знакомой коллективу программистов или отдельному программисту.

Вспомогательные системы программирования обычно не используются для разработки программ, основное их назначение – обеспечение эффективного интерфейса между пользователем и различными компонентами системного ПО.

Наконец, в рамках третьего способа классификации предлагается разделить системы программирования на алгоритмические и неалгоритмические. Основное свойство неалгоритмических систем состоит в том, что в программе описывается не алгоритм решения задачи, а сама задача. Для подобного описания используются специальные средства, например, язык логических предикатов. В соответствии с утвержденной программной в данном курсе будут изучаться только алгоритмические системы программирования, а неалгоритмические будут рассмотрены в последующих курсах.

Перейдем к вопросу о типах трансляторов. Выделим для будущего рассмотрения три типа трансляторов:

1) Ассемблеры.
2) Интерпретаторы.
3) Компиляторы.

Дадим краткие определения.

Очевидно, что ассемблеры – это трансляторы с машинно-ориентированных языков.

Основное свойство интерпретаторов состоит в том, что в них совмещены процессы трансляции и исполнения программ. При этом обычно на очередном шаге работы интерпретатора из входного текста выделяется очередной оператор, выделенный оператор транслируется в машинные команды и немедленно исполняется, а результаты трансляции не запоминаются, и так далее на каждом очередном шаге. Следует заметить, что при определении интерпретаторов мы никак не оговариваем уровень входного языка (машинно-зависимый или машинно-независимый), так как главное в определении интерпретатора – это принцип его функционирования, а не уровень входного языка.

Компиляторами будем называть такие трансляторы с машинно-независимых языков ("языков высокого уровня"), которые сначала всю входную программу просматривают от начала до конца, всю ее транслируют, а лишь затем передают на исполнение.

1. Компиляторы

1.1. Фазы (стадии) работы компиляторов

Вопросы, подлежащие изучению:

1) Достоинства языков высокого уровня.
2) Функциональная последовательность фаз работы компиляторов (диаграмма).
3) Виды объектных модулей на выходе компилятора; количество проходов; формы передачи информации от одной фазы работы компилятора к другой.
4) Краткая характеристика отдельных фаз работы компилятора.

Выше мы определили компиляторы как трансляторы с языков высокого уровня (ЯВУ), а потому кратко перечислим достоинства подобных языков.

1) ЯВУ более естественны (по сравнению с ассемблерами), а потому легче изучаются.
2) Программы, написанные на ЯВУ, легче отлаживаются.
3) Программы на ЯВУ во многом самодокументированы.
4) Повышается производительность труда программистов, выраженная в количестве команд выходного объектного кода.
5) С учетом оптимизирующих возможностей современных компиляторов возможно получение высоко эффективных выходных кодов.

Теперь перейдем к рассмотрению так называемых фаз (или, иначе, стадий) работы компилятора. Заметим, что предлагаемое обсуждение будет лишь первой итерацией в рассмотрении фаз работы компиляторов, а более детальное изучение будет осуществлено после введения необходимого формального аппарата.

Выходной продукт компилятора (как, впрочем, и ассемблирующего транслятора) принято называть объектным модулем. Так называемый "абсолютный" (неперемещаемый) модуль может быть загружен для корректного исполнения только в конкретную и заранее определенную область памяти. Абсолютный модуль содержит в себе только машинные команды и данные, полученные в результате трансляции входного текста. Необходимо заметить, что абсолютные модули имеют смысл только для тех ЭВМ и семейств ЭВМ, где внутренняя структурная организация такова, что имеет место так называемое "единое адресное пространство" главной памяти.

Что касается "перемещаемого" объектного модуля, то этот тип, вероятно, наиболее хорошо знаком слушателям настоящего курса (благодаря занятиям по предыдущим дисциплинам). Перемещаемый модуль может быть загружен в

произвольную область памяти, но в общем случае при подобной загрузке потребуется "настройка" модуля на конкретные адреса загрузки. Обычно настройке подлежат адресные поля команд, ссылающихся на данные или другие команды. Таким образом, структура перемещаемого объектного модуля неизбежно усложняется по сравнению со структурой абсолютного модуля. Перемещаемый модуль кроме машинных команд и данных должен содержать представленную в той или иной форме информацию об элементах модуля, подлежащих настройке. Примером подобной формы может служить так называемая таблица настраиваемых элементов.

Наконец, третий обсуждаемый тип объектного модуля – это модуль, содержащий текст на языке ассемблера. В этом случае используется специфическая двухступенчатая схема трансляции (см. раздаточный материал). Полезной особенностью данной схемы является возможность ознакомиться с результатами работы компилятора, изучая ассемблерный текст.

Перейдем к вопросу о том, за сколько проходов компилятор может или должен выполнить свою работу. Будем считать, что "проход транслятора" – это однократный просмотр входного текста от начала до конца с выполнением каких-либо транслирующих действий. Минимальное теоретически достижимое количество проходов компилятора зависит от свойств входного языка (прежде всего, от наличия или отсутствия контекстной зависимости в языке) и для существующих в настоящее время алгоритмических языков лежит в диапазоне от одного до трех.

Фактическое количество проходов в конкретной реализации компилятора зависит от ряда факторов (см. раздаточный материал) и может доходить до девяти.

Выделим только две формы передачи информации от одной фазы компилятора к другой:

- таблицы;
- скомпилированные команды, в том числе, обобщенные (внутрикомпиляторные);

а содержание этих форм указано на диаграмме (см. раздаточный материал).

Дадим краткую характеристику отдельным фазам работы компилятора.

Лексический анализ.

Для справки дадим определение: "лексема" (ударение на второй слог) – это неделимая словарная единица языка. Будем считать, что однотипные лексемы составляют так называемый "синтаксический класс". Применительно

к алгоритмическим языкам можно выделить следующие типовые синтаксические классы:

- идентификаторы (имена);
- литералы (непосредственные данные);
- зарезервированные слова;
- разделители;
- операторы (знаки операций);
- прочие.

Определение: на фазе лексического анализа элементы входной программы разделяются на синтаксические классы, формально унифицируются, а несущественные элементы программы отбрасываются.

Выходной продукт лексического анализатора часто называют "лексической сверткой" или "однородным представлением программы". В частности, одной из форм однородного представления программы может быть так называемая "таблица однородных символов". Получение на фазе лексического анализа однородного представления входной программы позволяет упростить работу последующих фаз работы компилятора. Терминологическое замечание: лексический анализатор часто называют "сканером".

Контрольные вопросы:

1) Является ли лексический анализ обязательной фазой работы компилятора? *(Ответ: нет, не является, но обычно присутствует, так как в противном случае всем последующим фазам компилятора придется оперировать с исходным, т.е. неоднородным представлением программы).*

2) Обязательно ли нужен отдельный проход для осуществления лексического анализа? *(Нет не обязательно, так как лексический анализ может быть выполнен как подпрограмма следующей фазы компилятора – грамматического анализатора).*

Грамматический разбор.

Грамматический разбор является центральной и неотъемлемой фазой работы компилятора. В дальнейшем именно этой фазе будет уделено наибольшее внимание, а сейчас дадим лишь краткую характеристику.

На этой фазе компиляции обычно одновременно осуществляются три трудно разделимых между собой процесса: синтаксический анализ, семантический анализ, генерация внутренней (промежуточной) формы оттранслированной программы. Синтаксический анализ есть процесс выявления синтаксической корректности входного предложения (т.е. соответствия формальным правилам грамматики входного языка).

Семантический анализ есть процесс выявления смысла входного предложения. Во время генерации внутренней промежуточной формы оттранслированной программы создается одна из возможных типовых форм. Наиболее известны следующие формы:

- матрица;
- польская запись;
- список;
- дерево.

В дальнейшем в данном курсе будет уделено внимание каждой из указанных форм.

Оптимизация.

Прежде всего, укажем два типовых и важнейших критерия любой оптимизации программ во время их компиляции:

- минимизация времени выполнения программы;
- минимизация требуемого объема памяти (или, по крайней мере, достижение разумного компромисса между этими иногда противоречащими друг другу критериями).

Возможны различные способы классификации видов оптимизации. Рассмотрим два способа классификации. В первом способе выделим следующие виды оптимизации:

1.1. Машинно-независимая оптимизация.

1.2. Машинно-зависимая оптимизация.

Во втором способе рассмотрим такие виды оптимизации:

2.1. Локальная оптимизация.

2.2. Глобальная оптимизация.

В случае 1.1. речь идет об оптимизации без какого-либо учета аппаратурных особенностей ЭВМ, на которой будет исполняться оттранслированная программа, но с учетом особенностей языка программирования. Подобную оптимизацию называют еще оптимизацией на уровне языка. Наиболее известны следующие приемы машинно-независимой оптимизации:

- приведение общих подвыражений;
- вычисление литеральных подвыражений на стадии компиляции;
- оптимизация логических выражений и подвыражений;
- оптимизация циклов.

В случае 1.2., напротив, речь идет об оптимизации с учетом аппаратурных особенностей ЭВМ, но без учета особенностей языка программирования.

Говоря о локальной и глобальной оптимизации, отметим, что в первом случае имеется в виду, что те или иные приемы оптимизации применяются к отдельным фрагментам транслируемой программы (вплоть до отдельного оператора), а во втором – приемы оптимизации применяются к программе в целом. Конечно, от глобальной оптимизации следует ожидать большего эффекта, чем от локальной, но и алгоритмы глобальной оптимизации существенно сложнее. Сделаем еще два замечания. Во-первых, в зависимости от полноты реализуемой оптимизации только оптимизация может потребовать нескольких проходов компилятора. Во-вторых, объем оптимизирующего компилятора может в несколько раз превышать объем неоптимизирующего.

Распределение памяти и генерация кода.

Кратко отметим, что в зависимости от типа генерируемого объектного модуля имеет место распределение памяти:

а) фактическое;

б) символическое.

Если на выходе компилятора создается перемещаемый модуль, то имеет место так называемое фактическое распределение памяти с точностью до адресов (по крайней мере, относительных). Если же создается модуль, содержащий ассемблерный текст, имеет место символическое распределение памяти, при котором функция распределения памяти сводится к генерации в составе ассемблерного текста символических директив резервирования памяти, а фактическое распределение памяти в дальнейшем выполнит ассемблирующий транслятор.

Что касается генерации кода, то аналогично выше сказанному генерируется код, соответствующий типу создаваемого на выходе компилятора объектного кода.

1.2. Элементы теории формальных языков

Вопросы, подлежащие изучению:

1) Понятие грамматики, как средства для задания языка.
2) Бэкуса-Наура формы (БНФ) для записи правил грамматик.
3) Рекурсия в правилах грамматик. Понятие "эквивалентные грамматики".
4) Классификация грамматик по Хомскому. Необходимые распознающие агрегаты.
5) Классификация языков по Хомскому.

На данном этапе рассмотрения пока не строго определим формальные языки в противоположность естественным языкам (русскому, английскому и т.п.), как языки каким-либо формальным способом определенные и имеющие ограниченную по сравнению с естественными языками сферу применения. Языки программирования являются ярким примером формальных языков.

Возникает вопрос о способе формального задания языка. Если бы количество допустимых предложений какого-либо формального языка было конечным, то язык можно было бы задать перечислением допустимых предложений. Однако для большинства формальных языков, имеющих практическое значение, в т.ч. алгоритмических, предположение о конечности количества допустимых предложений не выполняется. Таким образом, необходим некий специальный формализм для задания (определения) языка. Обычно подобный формализм называют "грамматикой".

Первое определение грамматики. Пусть грамматика есть конечное, непустое множество правил, позволяющих генерировать все допустимые предложения языка. Замечание: при анализе предложений (в том числе, в компиляторах) эти же правила используются, так сказать, "обратным ходом".

Известны и используются на практике различные способы отображения (формы записи) правил грамматик. Исторически первой формой записи правил стала так называемая Бэкуса-Наура форма (БНФ). Общий вид правила в БНФ таков:

левая часть ::= правая часть

Читаются подобные правила следующим образом: "левая часть есть правая часть" или "левая часть выводится как правая часть". Сами правила в БНФ называют или правилами грамматики, или правилами подстановки, или продукциями.

Символы, используемые в БНФ:

- терминальные (неделимые символы языка);

- нетерминальные или **<нетерминалы>** , служащие для отображения промежуточных фаз генерации допустимого предложения (замечания: а) нетерминальные символы будем заключать в угловые скобки, б) допустимые предложения могут состоять только из терминальных символов);

| - символ альтернативной подстановки ("или");

ε - символ пустого множества.

Один из нетерминальных символов грамматики играет особую роль и называется "начальным символом" грамматики. Для него нет строгого формального определения, так как начальный символ определяется составителем грамматики аксиоматически, а потому начальный символ часто называют "аксиомой грамматики". Можно лишь не строго определить, что начальный символ грамматики – это тот символ, ради грамматического описания которого составляется вся грамматика.

(Пример грамматики, записанной в БНФ будет приведен на доске).

Однако при первом же использовании БНФ для записи правил грамматик выясняется, что без использования какого-либо специального приема не удастся с помощью конечного множества правил определить возможность генерации бесконечного множества допустимых предложений. Этим специальным приемом служит введение рекурсии в правила грамматик. Определения для различных типов рекурсивных грамматик приведены в раздаточном материале (см. раздаточный материал).

Там же дается важное для нас определение эквивалентных грамматик. Уместно здесь отметить важное для нас положение, доказанное в теории формальных языков – для любого языка может сконструировано сколь угодно большое количество эквивалентных грамматик.

Теперь можно обратиться к приведенному в раздаточном материале более формальному определению грамматики, когда грамматика **G** определена на основе четырех параметров:

$$\mathbf{G = (V_T , V_N , S , P),}$$

где V_T – алфавит терминальных символов;

V_N – алфавит нетерминальных символов;

S – начальный символ (аксиома) грамматики;

P – множество правил подстановки вида $\alpha ::= \beta$.

Особенности этого определения указаны в раздаточном материале.

Наконец обратимся к приведенной в раздаточном материале четырех типовой классификации грамматик по Хомскому. Данная классификация является общепризнанной. Для нас же важно отметить тот доказанный в теории формальных языков факт о соответствии типа грамматики и типа так называемого распознающего агрегата, необходимого для распознавания заданного грамматикой языка. Ниже приведена таблица соответствия.

Тип грамматики по Хомскому	Тип распознающего агрегата
Тип 0	Машина Тьюринга (с бесконечной лентой)
Тип 1	Машина Тьюринга с конечной лентой
Тип 2	Конечный автомат со стековой (магазинной) памятью
Тип 3	Конечный автомат

Из приведенного соответствия можно сделать важные выводы, касающиеся языков программирования и компиляторов. Легко заметить, что типы грамматик перечислены в нашей таблице в порядке, так сказать, убывания мощности, соответственно, типы распознающих агрегатов – в порядке убывания сложности. А ведь распознающая часть компилятора неизбежно будет программной реализацией того или иного распознающего агрегата, таким образом, чем проще будет грамматика, описывающая язык, тем проще будут распознающие программы компилятора. При конструировании языков программирования сложилась естественная тенденция к тому, чтобы для описания языков преимущественно использовались автоматные и контекстно-независимые грамматики (типы 3 и 2 по Хомскому). В противном случае, например, для языка, описанного грамматикой типа 1, распознающим агрегатом стала бы ЭВМ в целом со всеми ее ресурсами, а для случая грамматики типа 0 нам вообще не удастся найти физическую реализацию распознающего агрегата.

(Примеры автоматных и контекстно-независимых грамматик приводятся на доске по ходу лекции, а примеры грамматик типа 1 предлагаются в раздаточном материале).

Классификация языков по Хомскому.

Для языков используется аналогичная описанной выше четырех типовая классификация, однако здесь имеется особенность. Действительно, как уже отмечено выше, для любого языка существует сколь угодно большое множество эквивалентных грамматик, при этом вполне может так случиться, что отдельные грамматики указанного множества будут принадлежать к разным типам по классификации Хомского. Как же определить истинный тип языка? Истинный тип языка определяется по типу грамматики наименьшей мощности среди всего множество эквивалентных грамматик данного языка.

В заключение данного раздела предлагается ознакомиться с примерами грамматик типа 1 (контекстно-зависимых) и выполнить предлагаемые контрольные задания (см. раздаточный материал).

1.3. Лексический анализ

Вопросы, подлежащие изучению:

1) Граф (диаграмма) состояний и переходов лексического анализатора.
2) Матричное (табличное) представление лексического анализатора.
3) Принципы построения лексического анализатора в целом.

Настоящим разделом открывается вторая итерация рассмотрения фаз работы компилятора. По сравнения с разделом 1.1. все фазы компиляции будут рассмотрены более подробно и с использованием введенного аппарата грамматик.

Исходный тезис данного раздела таков: как правило, алгоритмические языки конструируются таким образом, чтобы элементы всех синтаксических классов (т.е. все лексемы) описывались только автоматными грамматиками. (При этом напомним, что перечисление есть простейший случай автоматной грамматики.) Таким образом, для распознавания лексем достаточно агрегата в виде конечного автомата, а лексический анализатор в таких условиях есть программная реализация конечного автомата. В свою очередь, для представления конечных автоматов чаще всего используются две общепринятые формы: графическая (граф) и табличная (матричная). Применительно к лексическим анализаторам (ЛА) рассмотрим каждую из двух указанных форм.

Граф (диаграмма) состояний и переходов лексического анализатора.

Подобный граф может быть получен на основе автоматной грамматики, описывающей лексемы какого-либо синтаксического класса. Дальнейшее рассмотрение проведем на примере конкретной грамматики. Пусть это будет грамматика десятичных чисел со знаком и без знака (см. раздаточный материал). Таким образом, пока будем рассматривать построение анализаторов для отдельных синтаксических классов.

(Граф состояний и переходов ЛА для грамматики десятичных чисел будет приведен на доске).

К достоинствам графического представления следует отнести наглядность, а к недостаткам – некоторое неудобство составления программы ЛА на основе графа.

Матричное (табличное) представление лексического анализатора.

Используем традиционную табличную форму, в которой каждой строке будет соответствовать текущее состояние автомата (в данном случае лексического анализатора), а столбцам поставим в соответствие входные символы (в данном случае литеры). В ячейке на пересечении какой-либо строки и какого-либо столбца будем указывать номер состояния, в которое перейдет анализатор под воздействием входного символа, если такой переход является разрешенным. Пустые ячейки будут соответствовать не разрешенным переходам, т.е. тем переходам анализатора, которые выводят его из процесса распознавания десятичных чисел. Переходы, поименованные термином "ВЫХОД", соответствуют благополучному выходу из процесса распознавания десятичных чисел.

	Входной символ			
Текущее состояние	Знак + или –	Цифра 0,1,2,…,9	Десятичная точка	Прочие
1. Старт	**2**	**3**		
2. Знак		**3**		
3. Целое		**3**	**4**	
4. Смешанное число		**5**		
5. Десятичное число	**ВЫХОД**	**5**		**ВЫХОД**

Однако построенный анализатор даже в случае благополучного выхода из процесса распознавания лишь констатирует тот факт, что на входе анализатора была последовательность литер, соответствующая синтаксически корректному десятичному числу. Если мы зададимся вопросом, например, каково значение входного числа, то анализатор на этот вопрос не отвечает. В подобных ситуациях имеет смысл дополнить распознающий агрегат так называемыми семантическими программами.

Пусть в данном случае мы ставим цель ввести в состав анализатора семантические программы для вычисления арифметического значения входного числа (в общем случае перед семантическими программами может быть поставлена любая другая цель).

Введем обозначения, которые будем использовать при описании семантических программ:

Ч – значение модуля числа;

Зн – значение знака числа;

цф – значение цифры из входной последовательности;

зн – значение знака из входной последовательности;

Д – делитель.

Наименования и содержание программ представим в таблице:

Имя программы	Содержание программы
Инициация	Ч := 0; Зн := +; Д:= 1.0
Р1	Зн := зн
Р2	Ч := Ч * 10.0 + цф
Р3	Д := Д * 10.0; Ч := Ч + цф / Д

Краткий комментарий. Программа Р1 – это программа присвоения знаку числа значения знака из входной последовательности. Программа Р2 реализует известное рекуррентное соотношение для вычисления значения целой части числа при условии просмотра входной строки слева направо. Программа Р3 служит для вычисления значения числа на той стадии распознавания, когда просматриваются значащие цифры дробной части.

Дополним матрицу состояний и переходов рассматриваемого лексического анализатора, указывая подлежащие исполнению семантические программы в правой части соответствующих ячеек матрицы. Например, если в ячейке матрицы на пересечении строки "1.Старт" и столбца "Цифра" указана программа Р2, то это означает, что при переходе анализатора из состояния "1. Старт" в состояние "3. Целое" под воздействием цифры необходимо кроме собственно перехода в состояние "3. Целое" еще и выполнить программу Р2.

	Входной символ			
Текущее состояние	Знак + или –	Цифра 0,1,2,…,9	Десятичная точка	Прочие
1. Старт	**2, Р1**	**3, Р2**		
2. Знак		**3, Р2**		
3. Целое		**3, Р2**	**4**	
4. Смешанное число		**5, Р3**		
5. Десятичное число	**ВЫХОД**	**5, Р3**		**ВЫХОД**

Наконец, вспомним о том, что в трансляторах обычно выдаются диагностические сообщения, и в порядке решения учебной задачи предусмотрим в нашем лексическом анализаторе возможность формирования диагностических сообщений. Перечень сообщений, который мы приведем, будет лишь одним из возможных, так как очевидно, что в зависимости от объективных или субъективных причин можно сформировать сколько угодно различных перечней. Заметим, что сообщения в предлагаемом перечне можно разделить на два типа: те, которые предназначены для выдачи пользователю (составителю транслируемой программы), и те, которые мы будем называть внутрикомпиляторными, так как они индицируют ситуацию, когда входная цепочка не является лексемой заданного синтаксического класса, но, может

быть, является лексемой какого-либо другого синтаксического класса. Ниже приведем матрицу лексического анализатора, которая по сравнению с предыдущими будет дополнена символическими именами диагностических сообщений, и таблицу диагностических сообщений.

	Входной символ			
Текущее состояние	Знак + или –	Цифра 0,1,2,…,9	Десятичная точка	Прочие
1. Старт	**2, P1**	**3, P2**	**D1**	**D1**
2. Знак	**D2**	**3, P2**	**D3**	**D3**
3. Целое	**D4**	**3, P2**	**4**	**D5**
4. Смешанное число	**D6**	**5, P3**	**D7**	**D6**
5. Десятичное число	**ВЫХОД**	**5, P3**	**D8**	**ВЫХОД**

Таблица диагностических сообщений

Имя сообщения	Содержание сообщения
D1	Не десятичное число, так не начинается со знака или цифры
D2	Ошибка: два знака подряд
D3	Не десятичное число, так как за знаком не следует цифра
D4	Не десятичное число, но целое со значением ЗнЧ
D5	Не десятичное число, так за целой частью не следует точка
D6	Ошибка: за десятичной точкой не следует цифра
D7	Ошибка: две точки подряд
D8	Ошибка: две точки в одном числе

Если в заключение сравнить графическую и матричную форму представления лексического анализатора, то достоинством графической формы является ее наглядность. Основным достоинством матричной формы (особенно если она уже оснащена семантическими программами и диагностическими сообщениями) является то, что данная форма с успехом может быть использована в качестве исходного материала для составления программы лексического анализатора.

До настоящего момента мы рассматривали так называемые "элементарные" лексические анализаторы, которые позволяют распознавать лексемы только одного синтаксического класса. Вопросы построения лексического анализатора в целом, т.е. для случая распознавания лексем всех синтаксических классов какого-либо языка рассматриваются в раздаточном материале (см. раздаточный материал).

1.4. Грамматический разбор

Вопросы, подлежащие изучению:

1) Понятие "синтаксическое дерево".

2) Определение грамматического разбора. Методы грамматического разбора.

3) Грамматический разбор сверху-вниз с возвратами. Линейное представление синтаксического дерева.

4) Синтаксические диаграммы как средство отображения правил грамматик и процесса распознавания.

5) Рекурсивный спуск как безвозвратный метод грамматического разбора сверху-вниз.

6) Грамматики предшествования и грамматический разбор снизу-вверх.

7) Матричное представление синтаксического дерева.

Как уже отмечалось выше, грамматический разбор – это центральная и неотъемлемая фаза работы компилятора. Прежде чем перейти к рассмотрению собственно грамматического разбора введем понятие "синтаксическое дерево".

Синтаксическое дерево – это граф, который поясняет (иллюстрирует) процесс генерации допустимого предложения в рамках заданной грамматики или процесс распознавания предложения (опять же в рамках заданной грамматики). Структура синтаксического дерева может быть описана следующим образом. Корнем дерева служит начальный символ грамматики, в качестве листьев дерева выступают терминальные символы, а все узлы остальные узлы графа, лежащие между корнем и листьями, представлены нетерминальными символами грамматики. Замечание: принято изображать синтаксические деревья корнем вверх, а листьями вниз.

В раздаточном материале приведен пример синтаксического дерева для десятичного числа -123.45 , причем, граф дерева изображен с максимальной детализацией в том смысле, что в узлах графа указаны не только наименования нетерминальных символов, но и условные обозначения правил грамматики, которые используются на каждом шаге генерации (или распознавания) предложения. Приведенное синтаксическое дерево соответствует грамматике десятичных чисел, которая, в свою очередь, приведена в раздаточном материале в разделе "грамматический разбор".

Определим грамматический разбор как процесс восстановления синтаксического дерева для входного предложения в рамках заданной грамматики. Если дерево удалось восстановить, то входное предложение

является синтаксически корректным в рамках заданной грамматики, в противном случае – нет.

Однако ранее мы утверждали, что грамматический разбор есть осуществляемые одновременно процессы синтаксического и семантического анализа, а в приведенном выше определении мы пока сказали только о семантической корректности входного предложения. Определен ли в процессе восстановления синтаксического дерева смысл входного предложения и где он "спрятан"? В рамках теории формальных языков утверждается, что смысл входного предложения отображен в структуре восстановленного синтаксического дерева.

Теперь возникает новый вопрос – об однозначности определенного выше грамматического разбора. Определение: будем называть грамматику неоднозначной, если хотя бы для одного допустимого предложения существует более одного синтаксического дерева. Отсюда следует рекомендация разработчикам алгоритмических языков – грамматики алгоритмических языков должны быть однозначны. Классическим примером неоднозначной грамматики стала грамматика операторов условного перехода, которая допускает и полную, и сокращенную форму оператора условного перехода. В целях практического ознакомления с неоднозначными грамматиками рекомендуется выполнить упражнение.

Упражнение. Пусть дана грамматика операторов условного перехода:

a) <S> ::= if b then <S> else <S>
b) <S> ::= if b then <S>
c) <S> ::= a ,

где b – логическое выражение, принимающее значение «истина» или «ложь»;

a – лексема, представляющая любой оператор кроме оператора условного перехода.

Показать, что для допустимого входного предложения
if b_1 then if b_2 then a_1 else a_2
существуют, по крайней мере, два разных синтаксических дерева.
Показать также, что этим деревьям, в свою очередь, соответствуют существенно отличающиеся структуры программ.

Дадим определения еще нескольким вспомогательным понятиям. Будем считать, что процесс грамматического разбора состоит из последовательности шагов. На каждом шаге разбора осуществляется или сдвиг (shift) к следующему символу, или так называемая свертка, иначе говоря, редукция (от английского reduction) уже просмотренных символов в соответствии с одним из правил заданной грамматики. Забегая вперед, отметим, что главной проблемой разработки любого метода грамматического разбора как раз и является нахождение алгоритма разрешения конфликта сдвиг/свертка (shift/

reduction) для любого шага разбора. Будем называть "основой" символ или группу символов, сворачиваемых на очередном шаге разбора. Будем называть грамматический разбор "каноническим", если символы входного предложения просматриваются слева-направо. Условимся, что в дальнейшем в данном разделе будем рассматривать только канонические способы разбора.

Если говорить о типизации методов грамматического разбора, то принято прежде всего разделять их на методы "сверху-вниз" ("нисходящий разбор") и методы "снизу-вверх" ("восходящий разбор"). В первом случае синтаксическое дерево восстанавливается от корня к листьям, а во втором – от листьев к корню.

Даже после того как мы определили, что есть нисходящий или восходящий разбор, не вполне очевидно каким должен быть порядок сдвигов и сверток в том или ином случае. Однако теория формальных языков дает ответ на этот вопрос. Установлено, что при грамматическом разборе сверху-вниз порядок использования правил должен быть точно таким же, как и при так называемом левом выводе предложения, а при грамматическом разборе снизу-вверх порядок использования правил должен быть обратным по отношению к правому выводу. Напомним, что при левом выводе на каждом шаге вывода правила подстановки используются для самого левого нетерминального символа сентенциальной формы, а при правом – для самого правого. В раздаточном материале приведены диаграммы для левого и правого вывода десятичного числа –123.45, из которых можно для данного примера определить порядок сверток для грамматического разбора сверху-вниз и снизу-вверх, соответственно. Так при грамматическом разборе сверху-вниз порядок сверток будет таким: a, b2, d1, e2, d1, e3 и т.д., а при грамматическом разборе снизу-вверх, соответственно: e2, e3, e4, d2, d1, d1, b2 и т.д.

Перейдем теперь к рассмотрению конкретных методов грамматического разбора. Начнем с грамматического разбора "сверху-вниз с возвратами".

Дадим словесное описание алгоритма разбора. Разбор управляется непосредственно правилами грамматики, представленными в БНФ. Сам разбор состоит из последовательности шагов. На каждом шаге ставится некая цель распознавания. Если целью является распознавание нетерминала, то на очередном шаге разбора ставится соответствующая подцель (на основе соответствующего правила грамматики) и так далее, пока не останутся одни терминалы. Если какая-либо из целей или подцелей оказалась недостижимой, то необходимо вернуться к предыдущим шагам и испытать альтернативные правила, если они имеются. Если и после испытания всех альтернатив цель или

подцель осталась недостижимой, то предложение не является допустимым в рамках заданной грамматики.

(Пример реализации алгоритма сверху-вниз с возвратами в форме таблицы шагов разбора будет приведен на доске).

Отметим, что при реализации рассматриваемого метода разбора в случае благополучного разбора какого-либо входного предложения в результате обычно будет получено так называемое "линейное представление синтаксического дерева". Обсуждение линейного представления синтаксического дерева, а также особенностей грамматического разбора сверху-вниз с возвратами можно найти в раздаточном материале *(см. раздаточный материал)*.

Введем в рассмотрение так называемые "синтаксические диаграммы" и обсудим возможность их двоякого использования:

а) как средства отображения правил грамматик;

б) как средства, иллюстрирующего процесс распознавания.

Сначала остановимся на первой функции синтаксических диаграмм. Синтаксические диаграммы – это графическая форма представления правил грамматик, являющаяся альтернативной по отношению к Бэкуса-Наура формам. Доказано, что синтаксические диаграммы и БНФ являются совершенно эквивалентными формами для представления автоматных и контекстно-зависимых грамматик. Это означает, что любой грамматики, представленной в БНФ, можно найти эквивалентную синтаксическую диаграмму, и наоборот. Естественно, что возможны эквивалентные преобразования синтаксических диаграмм, аналогичные эквивалентным преобразованиям грамматик, представленным в БНФ.

Дальнейшее обсуждение будем вести с использованием иллюстраций, приведенных в раздаточном материале *(см. раздаточный материал).* Символы синтаксических диаграмм (аналогично символам БНФ) разделяются на терминальные и нетерминальные. Первые из них обычно отображаются кругами или овалами, а вторые – квадратами или прямоугольниками. В качестве соединительных линий используются направленные ветви графа синтаксической диаграммы. Принято синтаксические диаграммы изображать таким образом, чтобы они развивались слева направо, а символ, определяемый диаграммой, обычно размещают в левом верхнем углу поля диаграммы. Также в раздаточном материале приведены типовые конструкции (примитивы) синтаксических диаграмм. Их всего четыре: последовательное соединение (конкатенация); параллельное соединение (альтернатива); повторение символа

ноль или один раз; цикл (повторение символа или ноль раз, или один раз, или сколько угодно раз). Указанных примитивов достаточно для конструирования сколь угодно сложных синтаксических диаграмм.

Рассмотрим два типовых примера грамматик, представленных в форме синтаксических диаграмм *(см. раздаточный материал).* Первый пример – это автоматная грамматика языка десятичных чисел, а второй – контекстно-независимая грамматика языка арифметических выражений. Приведенные примеры наглядно иллюстрируют принципиальные отличия синтаксических диаграмм указанных двух типов грамматик: синтаксические диаграммы автоматных грамматик (тип 3 по Хомскому) развиваются линейно и не содержат рекурсивных вызовов распознающих процедур, а вот синтаксические диаграммы контекстно-независимых грамматик (тип 2) неизбежно содержат рекурсивные вызовы распознающих процедур.

Для иллюстрации второй возможной функции синтаксических диаграмм введем понятие "распознающая точка". Представим себе процесс просмотра символов входного (анализируемого) предложения, и пусть эта абстрактная распознающая точка перемещается по синтаксической диаграмме от ее начала и далее под управлением каждого очередного символа. Если распознающей точке удалось пройти путь от начала синтаксической диаграммы до выхода из диаграммы, то входное предложение является допустимым в рамках заданной грамматики, в противном случае – нет. Мы в еще большей мере усилим возможности синтаксических диаграмм по иллюстрации процесса трансляции, если оснастим их семантическими программами. В примерах раздаточного материала синтаксическая диаграмма грамматики десятичных чисел оснащена семантическими программами для вычисления арифметического значения числа. Обозначения и содержание семантических программ соответствуют введенным в разделе "Лексический анализ". Можно сделать вывод о том, что подобные синтаксические диаграммы (с учетом возможности оснащения семантическими программами) являются достаточно наглядной и удобной формой, пригодной для использования в качестве исходной при разработке соответствующих транслирующих программ.

Рекурсивный спуск как безвозвратный метод грамматического разбора сверху-вниз. Данный метод разбора впервые был предложен Н. Виртом в период разработки языка Паскаль. Основное достоинство метода – высокая скорость разбора. Однако условием реализуемости этого метода является удовлетворение грамматики транслируемого языка специальным ограничениям. Наряду с уже введенными ограничениями, характерными для всех нисходящих методов разбора (например, невозможность обработки

прямой левой рекурсии) здесь приходится вводить дополнительные. Эти ограничения можно выразить в разных формах. Выраженный в словесной форме основной принцип рекурсивного спуска таков – на каждом шаге грамматического разбора дальнейшее направление разбора должно определяться только очередным терминальным символом. Если переформулировать данное ограничение на язык синтаксических диаграмм, то оно будет выглядеть следующим образом. Необходимо рассмотреть каждую точку разветвления диаграммы, и каждая ветвь диаграммы, исходящая из любой точки разветвления должна начинаться только с терминального символа, причем, множество этих терминальных символов должно быть не пересекающимся. Сформулированное ограничение должно выполняться с учетом возможной вложенности синтаксических диаграмм друг в друга.

В публикациях можно встретить аббревиатуру LL(1), которая часто используется в качестве краткого условного обозначения, как метода рекурсивного спуска (LL(1) метод), так и грамматик, пригодных для реализации метода рекурсивного спуска (LL(1) грамматики).

(Во время лекции на доске будет приведен пример грамматики арифметических выражений, пригодной для реализации метода рекурсивного спуска).

Грамматики предшествования и грамматический разбор снизу-вверх.

Перейдем теперь к рассмотрению восходящих методов грамматического разбора. В данном разделе ознакомимся с одним из частных методов грамматического разбора снизу-вверх, который использует понятие так называемых "грамматик предшествования".

Пусть **AB** есть два последовательных символа предложения какого-либо языка, тогда между ними возможны следующие типы отношений предшествования:

1) Говорят, что **A** предшествует (старше) **B**, если **A** принадлежит основе, а **B** – нет.
2) Говорят, что **B** предшествует (старше) **A**, если **B** принадлежит основе, а **A** – нет.
3) **A** и **B** принадлежат одной и той же основе.
4) Последовательность **AB** не может встретиться в допустимых предложениях языка.

Доказано, что если в рамках какой-либо грамматики для любой пары символов отношения предшествования определены однозначно, то на этой основе может быть построен грамматический анализатор снизу-вверх.

Будем называть грамматиками операторного предшествования такие, в которых отношения предшествования зависят только от операторов и не зависят от операндов. Оказывается, многие типовые конструкции алгоритмических языков при их грамматическом описании удовлетворяют указанному ограничению. Рассмотрим в качестве подобного примера скобочные арифметические выражения произвольной сложности.

Введем арифметические выражения со следующими операторами:

Символическое обозначение оператора	Смысловое значение оператора
+	Оператор сложения
*	Оператор умножения
(	Открывающая скобка
)	Закрывающая скобка
#	Ограничитель арифметического выражения слева
\n	Ограничитель арифметического выражения справа

Выявленные отношения предшествования принято отображать в виде так называемой таблицы отношений предшествования. Составим подобную таблицу для введенных нами арифметических выражений.

	Следующий оператор					
Текущий оператор	#	+	*	(	)	\n
#		<	<	<		ВЫХОД
+		>	<	<	>	
*		>	>	<	>	
(		<	<	<	= =	
)		>	>		>	
\n						

Обозначения, принятые в таблице:

< - между символами находится начало основы;

> - между символами находится конец основы;

= = - символы принадлежат к одной и той же основе;

- пустая клетка обозначает невозможность данной последовательности.

Алгоритм грамматического разбора на основе выявленных отношений предшествования относительно прост. Дадим его словесное определение.

Входное предложение просматривается слева направо до первого обнаружения конца основы, после чего осуществляется обратный просмотр до первого обнаружения начала основы. Символы между обнаруженными таким образом началом и концом основы сворачиваются, после чего продолжается

просмотр слева направо до первого обнаружения конца основы и т.д. аналогично выше сказанному. Легко заметить, что описанный алгоритм довольно естественно может быть реализован с использованием такого агрегата, как конечный автомат со стековой (магазинной) памятью.

(Во время лекции с использованием аудиторной доски могут быть приведены примеры грамматического разбора. Так, например, для арифметических выражений:

*# i + i * i \n и # X + Y * (Z + X) + A \n*

в результате разбора будут получены свертки:

*для первого выражения T1 = i * i*

T2 = i + T1 ,

а для второго выражения T1 = Z + X

*T2 = Y * T1*

T3 = X + T2

T4 = T3 + A ,

где T1, T2 и т.д. – ячейки временной памяти с соответствующими номерами.

Выводы, которые можно сделать из приведенных примеров:

- во время грамматического разбора контекстно-независимых языков в общем случае приходится генерировать так называемые ячейки временной памяти для хранения промежуточных результатов;

- оба примера подтверждают, что грамматический разбор в данном случае осуществляется именно снизу-вверх.)

Матричное представление синтаксического дерева.

В заключение настоящего раздела введем и кратко обсудим одну из возможных внутренних форм оттранслированной программы. Будем называть матрицей синтаксического дерева внутримашинное представление приведенной ниже табличной структуры.

Ячейка временной памяти (номер строки матрицы)	Оператор	Операнд 1	Операнд 2	Указатель вперед	Указатель назад
T1	+	Z	X	2	0
T2	*	Y	T1	3	1
T3	+	X	T2	4	2
T4	+	T3	A	5	3
...	...	...	...	...	...

Совокупность первых четырех элементов матрицы – ячейка временной памяти, оператор, операнд 1, операнд 2 – часто называют "тетрадой", а

совокупность трех элементов – оператор, операнд 1, операнд 2 – "триадой". Тетрады первых четырех строк приведенной выше матрицы только для примера заполнения подобных матриц приведены в соответствии с результатами разбора арифметического выражения **X + Y * (Z + X) + A** . Поле "указатель вперед" содержит номер логически следующей строки матрицы, а поле "указатель назад" – номер логически предыдущей строки. Поля указателей необходимы только в тех случаях, когда в компиляторе предполагаются действия над матрицей синтаксического дерева (например, на фазе оптимизации). Необязательность наличия в матрице колонок указателей вперед и назад отображена на нашем рисунке тем, что эти колонки выделены пунктирными линиями.

Еще одно замечание. Мы только для наглядности нашего рисунка показали размещение в матрице самих операторов и операндов, обычно там размещаются однородные представления указанных элементов, выработанные лексическим анализатором.

Условимся, что при предстоящем рассмотрении следующих фаз работы компилятора – оптимизации и генерации кода – мы временно ограничимся использованием только одной внутренней формы оттранслированной программы – матрицей синтаксического дерева. Возможность использования других форм рассмотрим несколько позже.

1.5. Машинно-независимая оптимизация.

Вопросы, подлежащие изучению:

1) Критерии машинно-независимой оптимизации.
2) Типовые приемы машинно-независимой оптимизации.
3) Примеры алгоритмов реализации машинно-независимой оптимизации.
4) Краткая характеристика отдельных фаз работы компилятора.

Еще раз напомним два типовых и важнейших критерия оптимизации программ во время их компиляции:

- минимизация времени выполнения программы;
- минимизация требуемого объема памяти (или, по крайней мере, достижение разумного компромисса между этими иногда противоречащими друг другу критериями).

В случае машинно-независимой оптимизации речь идет об оптимизации без какого-либо учета аппаратурных особенностей ЭВМ, на которой будет исполняться оттранслированная программа, но с учетом особенностей языка программирования. Подобную оптимизацию называют еще оптимизацией на уровне языка. Наиболее известны следующие приемы машинно-независимой оптимизации:

- приведение общих подвыражений;
- вычисление литеральных подвыражений на стадии компиляции;
- оптимизация логических выражений и подвыражений;
- оптимизация циклов.

Во время лекции, посвященной данной теме, с использованием аудиторной доски или раздаточного материала будет рассмотрен конкретный алгоритм машинно-независимой оптимизации и проиллюстрировано его применение на каком-либо примере. При этом в качестве исходной будет использована матрица синтаксического дерева, полученная на выходе грамматического анализатора, а в результате будет получена оптимизированная матрица.

1.6. Генерация кода и машинно-зависимая оптимизация.

Вопросы, подлежащие изучению:

1) Генерация неоптимизированного машинного кода.
2) Неразрывность процессов машинно-независимой оптимизации и генерации кода.
3) Примеры алгоритмов реализации машинно-зависимой оптимизации.

При проведении лекции по данной теме используется раздаточный материал *(см. раздаточный материал).*

В качестве промежуточный формы оттранслированной программы по-прежнему используется матрица синтаксического дерева. Показано, что простейшим способом генерации кода является использование так называемых "таблиц порождаемого кода". В этом случае генерация не оптимизированного кода эквивалента безусловным вызовам макросов, где именем макроса служит оператор, а фактическими параметрами – операнды и ячейка временной памяти.

Показано, что машинно-зависимая оптимизация есть не что иное, как генерация машинно-зависимо оптимизированного кода, и эти два процесса неразрывно связаны между собой. Для генерации машинно-зависимо оптимизированного кода могут быть использованы так называемые "оптимизирующие таблицы порождаемого кода". В этом случае генерация кода эквивалентна условным вызовам макросов. Обсуждаются типовые условия вызова макросов.

1.7. Распределение памяти как фаза работы компилятора

Вопросы, подлежащие изучению:

1) Фактическое и символическое распределение памяти.
2) Элементы программы, для которых должно быть осуществлено распределение памяти.
3) Классы памяти (сточки зрения ее распределения).

При проведении лекции по данной теме используется раздаточный материал *(см. раздаточный материал).*

Определяются объекты транслируемой программы, для которых потребуется распределение памяти. Выделяются классы памяти с точки зрения ее распределения: статическая, динамическая, управляемая. Обсуждаются особенности взаимодействия компилятора и операционной системы для всех трех случаев (классов памяти).

1.8. Потоки информации в компиляторе

В качестве заключения к разделу "компиляторы" предлагается рассмотреть и обсудить диаграмму "потоки информации в компиляторе" *(см. раздаточный материал).*

При построении диаграммы выделены постоянные таблицы (объекты) и динамические таблицы (объекты) компилятора, а собственно цель построения диаграммы – это наглядная иллюстрация взаимосвязей фаз работы компилятора с теми или иными постоянными и динамическими таблицами (объектами) компилятора.

2. Интерпретаторы

Интерпретаторы были определены выше как такие системы программирования, в которых совмещены процессы трансляции и выполнения. При этом отмечалось, что обычно имеют место пооператорная трансляция и выполнение, а результаты трансляции не запоминаются.

Очевидно, что, например, циклические программы в режиме интерпретации будут выполняться гораздо медленнее по сравнению с откомпилированными программами. Однако существуют области целесообразного применения интерпретаторов, укажем наиболее характерные из них:

- в учебном процессе при изучении языков программирования;
- при отладке программ;
- в интерфейсах различных программных систем, в т.ч. ОС (интерпретатор командной строки).

Существующие интерпретаторы можно разделить на два подтипа:

а) работающие непосредственно по входному тексту;

б) вырабатывающие некую промежуточную форму, которая затем обрабатывается в режиме интерпретации.

В качестве примера подобной внутренней формы рассмотрим так называемую "польскую запись".

Терминологическое замечание: привычную нам форму записи арифметических, логических и т.п. выражений следует называть "инфиксной" (от англ. infix) формой в том смысле, что операторы здесь размещены между операндами. Теперь укажем свойства польской записи:

1) Польская запись является бесскобочной.
2) Операнды польской записи следуют точно в том же порядке, что и в исходной инфиксной записи.
3) В зависимости от расположения операторов можно выделить два подтипа польских записей:

   3.1) в префиксной польской записи операторы непосредственно предшествуют своим операндам;

   3.2) в постфиксной, которую часто называют обратной польской записью (ОПЗ), операторы следуют непосредственно за своими операндами.

С теоретический точки зрения префиксная и постфиксная записи равносильны в том смысле, что обе могли бы с успехом использоваться в качестве внутренней формы в трансляторах. Однако на практике наибольшее распространение получила постфиксная или, иначе говоря, обратная польская запись (ОПЗ), поэтому для дальнейшего рассмотрения выберем только ОПЗ.

Приведем примеры преобразования инфиксной записи арифметических выражений в ОПЗ.

A + B —> AB+ ; A * B —> AB* ;

(A + B) * (C + D) —> AB+CD+* ;

A + B + C + D —> AB+C+D+

и т. д.

Приведем пример формального определения ОПЗ, пусть это будет грамматика ОПЗ для арифметических выражений:

a) < операнд > ::= < операнд > < операнд > < оператор > | идентификатор

b) < оператор > ::= + | – | * | / ;

для решения проблемы преобразования унарного минуса (если он встретился в исходной инфиксной записи) введем в грамматику ОПЗ дополнительное правило:

c) < операнд > ::= < операнд > @ ,

где @ специальное обозначение унарного минуса в ОПЗ.

В чем же смысл использования ОПЗ в качестве внутренней формы в интерпретаторах? Главное – это возможность вычисления выражения, представленного в ОПЗ, за один просмотр слева-направо с помощью конечного автомата со стековой памятью. Следует отметить простоту алгоритма вычисления.

Дадим описание алгоритма. Выражение, представленное в ОПЗ просматривается слева-направо, если встретился идентификатор, то он заносится на вершину стека, а если встретился оператор, то необходимо немедленно выполнить соответствующую операцию, при этом операнды (в частном случае, один операнд) "снимаются" с верхних уровней стека, а результат операции размещается на вершину стека. *(Во время лекции будет приведен пример вычисления выражения с использованием описанного алгоритма).*

Остается вопрос о том, насколько сложно получить ОПЗ из исходной инфиксной записи? Оказывается, сравнительно давно известен так называемый "алгоритм сортировочной станции", пригодный для осуществления необходимого преобразования. Причем, ОПЗ будет получена путем однократного просмотра инфиксного выражения слева-направо, а для реализации алгоритма опять потребуется лишь конечный автомат со стековой памятью. Таким образом, при использовании описанных алгоритмов работа интерпретатора в целом может быть представлена, как совместное функционирование двух конечных автоматов со стековой памятью.

3. Синтаксически управляемые процессы трансляции

3.1. Транслирующие грамматики

Раздел "Транслирующие грамматики", а также следующий за ним раздел "Атрибутные транслирующие грамматики" служат иллюстрацией и обоснованием возможности построения автоматизированных систем для разработки компиляторов. Подобные автоматизированные системы для краткости иногда называют "компиляторами компиляторов".

Введем понятия "символ действия", а затем "транслирующая грамматика".

Начнем рассмотрение с примера. Пусть поставлена задача символического описания процесса перевода арифметических многочленов, представленных в инфиксной форме, в обратную польскую запись (ОПЗ). Пусть входным является многочлен: ***A + B * D*** , тогда на выходе преобразователя должна появиться ОПЗ вида ***A B D * +*** .

Пусть символическим обозначением так называемого символа действия служат фигурные скобки: ***{...}*** . В свою очередь, содержанием символа действия могут быть любые действия, в принципе выполнимые с помощью ресурсов компьютера. В каждом конкретном случае содержание символов действия определяется в зависимости от целей перевода (трансляции). В нашем примере целью трансляции является формирование на некотором носителе обратной польской записи входного многочлена, поэтому разумно ввести символ действия, содержанием которого будет вывод на носитель того или иного символа. Например, ***{вывод A}*** или ***{вывод + }*** (в дальнейших выкладках, касающихся данного примера, термин "вывод" в символах действия будет отброшен). Тогда процесс перевода входного многочлена в ОПЗ можно представить в виде следующей цепочки, которую будем называть последовательностью актов:

A {A} + B {B} * D {D} {*} {+} .

Приведенную последовательность можно прочитать следующим образом: читать A, вывести A, читать B, вывести B, читать *, читать D, вывести D, вывести *, вывести + .

Ставится задача: найти такую контекстно-свободную грамматику, которая описывала бы указанный перевод. Предпримем попытку построить так называемую транслирующую грамматику для перевода инфиксных многочленов в ОПЗ на основе грамматики входных арифметических многочленов:

Исходная грамматика	Транслирующая грамматика
a) <М> ::= <М> + <ОД> b) <М> ::= <ОД> c) <ОД> ::= <ОД> * <С> d) <ОД> ::= <С> e) <С> ::= (<М>) f) <С> ::= A g) <С> ::= B h) <С> ::= D	a) <М> ::= <М> + <ОД> { + } b) <М> ::= <ОД> c) <ОД> ::= <ОД> * <С> { * } d) <ОД> ::= <С> e) <С> ::= (<М>) f) <С> ::= A {A} g) <С> ::= B {B} h) <С> ::= D {D}

где символ действия по-прежнему означает вывод соответствующего входного символа на некоторый носитель.

Определение. Будем называть транслирующими такие контекстно-свободные грамматики, в которых все терминальные символы разделены на два типа - входные и символы действия.

Цепочки, которые описываются транслирующими грамматиками, есть последовательности актов.

Если из транслирующей грамматики исключить символы действия, то оставшаяся грамматика будет называться входной, поскольку она является грамматикой входных выражений.

Процесс трансляции, осуществляемый под формальным управлением транслирующей грамматики будем называть синтаксически управляемым.

Замечание. Следует упомянуть о распространенном способе чтения правил (продукций) транслирующих грамматик. Так, например, продукцию а) из нашей транслирующей грамматики следует читать: обработка многочлена сводится к обработке многочлена, чтению оператора +, обработке одночлена и выводу оператора (символа) + .

Поскольку для входных выражений, подлежащих трансляции, существует множество эквивалентных грамматик, то при выборе в качестве входной каждой новой эквивалентной грамматики необходимо заново конструировать и транслирующую грамматику. Приведем еще один пример грамматики инфиксных арифметических выражений и соответствующий пример транслирующей грамматики для перевода входных выражений в ОПЗ:

Исходная грамматика	Транслирующая грамматика
a) <М> ::= <ОД> <М-список> b) <М-список> ::= + <ОД> <М-список> c) <М-список> ::= ε d) <ОД> ::= <С> <ОД-список> e) < ОД-список > ::= * <С> <ОД-список> f) < ОД -список> ::= ε g) <С> ::= (<М>) h) <С> ::= A i) <С> ::= B j) <С> ::= D	a) <М> ::= <ОД> <М-список> b) <М-список> ::= + <ОД> {+} <М-список> c) <М-список> ::= ε d) <ОД> ::= <С> <ОД-список> e) < ОД-список > ::= * <С> {*} <ОД-список> f) < ОД -список> ::= ε g) <С> ::= (<М>) h) <С> ::= A {A} i) <С> ::= B {B} j) <С> ::= D {D}

3.2. Атрибутные транслирующие грамматики

Во всех рассмотренные выше (в нашем курсе) грамматиках мы фактически оперировали только с синтаксическими классами символов грамматик. Однако для полной характеристики процесса трансляции необходимо учитывать так называемые "значения" символов. В свою очередь, в зависимости от целей трансляции значение какого-либо символа может быть представлено значениями одного или нескольких так называемых "атрибутов". Приведем примеры возможных атрибутов для символов некоторых типовых синтаксических классов алгоритмических языков:

Синтаксический класс	Примеры возможных атрибутов
1. Идентификаторы (имена)	1.1. Номер строки в таблице имен. 1.2. Тип. 1.3. Длина. 1.4. Первая литера идентификатора. 1.5. Количество литер в идентификаторе и т.п.
2. Литералы (непосредственные данные)	2.1. Номер строки в таблице литералов. 2.2. Тип. 2.3. Длина. 2.4. Исходное значение из входного текста. 2.5. Двоичное значение. и т.п.
3. Операторы	2.1. Номер строки в таблице операторов. 2.2. Тип оператора. 2.3. Ранг оператора. и т.п.

Таким образом, атрибутные транслирующие грамматики (АТГ) - это такие транслирующие грамматики, в которые введены атрибуты символов и операции над атрибутами.

Правила вычисления атрибутов в АТГ вводятся таким образом, чтобы значения атрибутов можно было вычислять по принципу: от символа синтаксического дерева к соседнему символу синтаксического дерева. При выполнении указанного условия атрибуты в АТГ могут быть разделены на два типа: а) "синтезируемые" атрибуты, значения которых вычисляются при движении по дереву от листьев к корню; б) "наследуемые" атрибуты, значения которых вычисляются при движении по дереву от корня к листьям. *(Во время лекции будут рассмотрены примеры АТГ с синтезируемыми и наследуемыми атрибутами с использованием аудиторной доски или раздаточного материала).*

Для нашего рассмотрения важен следующий факт: в теории формальных языков доказано, что в атрибутных транслирующих грамматиках содержится

достаточно информации для автоматизированного построения трансляторов, а это собственно и является теоретической предпосылкой и обоснованием возможности построения автоматизированных систем для разработки компиляторов.

4. Машинно-ориентированные системы программирования (ассемблеры)

Материал данного раздела относительно более прост по сравнению с материалом раздела "Компиляторы", а потому здесь в большей мере будет использована самостоятельная работа студентов, учебные пособия и раздаточный материал. В настоящем конспекте будут перечислены подразделы настоящего раздела и даны краткие комментарии.

4.1. Области применения и этапы развития машинно-ориентированных систем программирования

Несмотря на кажущуюся архаичность, ассемблеры имеют область целесообразного применения. Полезно также ознакомиться с этапами развития подобных систем программирования *(см. раздаточный материал)*.

4.2. Общая схема работы ассемблера

В данном подразделе рассматриваются типы команд и типы адресаций, обрабатываемых ассемблером *(см. раздаточный материал)*, обосновывается двухпроходная схема работы ассемблера.

4.3. Алгоритмы работы двухпроходного ассемблера

Констатируется, что двухпроходная схема работы ассемблера стала наиболее распространенной. Обсуждаются функции и алгоритмы первого и второго проходов *(при рассмотрении используется аудиторная доска и раздаточный материал).* Приводятся алгоритмы первого и второго проходов гипотетического ассемблера.

4.4. Однопроходные и трехпроходные ассемблеры

Обсуждается возможность построения однопроходных ассемблеров. Показывается, что достижения однопроходности обычно приходится или ужесточать правила программирования, и/или в какой-то степени жертвовать качеством выходного объектного модуля *(при рассмотрении используется аудиторная доска и раздаточный материал).*

В трехпроходном ассемблере обычно присутствует по сравнению с двухпроходным ассемблером дополнительный начальный проход, выполняющий функцию текстового препроцессора.

Литература

Литература по курсу "Транслирующие системы" (расширенный список)

Основная

1. Карпов Ю.Г. Теория и технология программирования: Основы построения трансляторов. Учебное пособие для вузов. – СПб: БХВ-Петербург, 2005. – 270 с. : ил.
2. Ахо Альфред, Лам Моника, Сети Рави, Ульман Джеффри. Компиляторы: принципы, технологии, инструментарий, 2-е изд.: Пер. с англ. – М.: ООО "Издательский дом Вильямс", 2011. – 1184 с.
3. Карпов, Ю.Г. Основы построения трансляторов [Электронный ресурс] : Учеб. пособие / Ю.Г. Карпов ; СПбГПУ .— Электрон. текстовые дан. (1 файл : 2,97 Мб) .— Загл. с титул. экрана .— Доступ из локальной сети ФБ .— Учебное пособие .— Adobe Acrobat Reader 4.0 .— <URL:http://www.unilib.neva.ru/dl/local/113.pdf>.

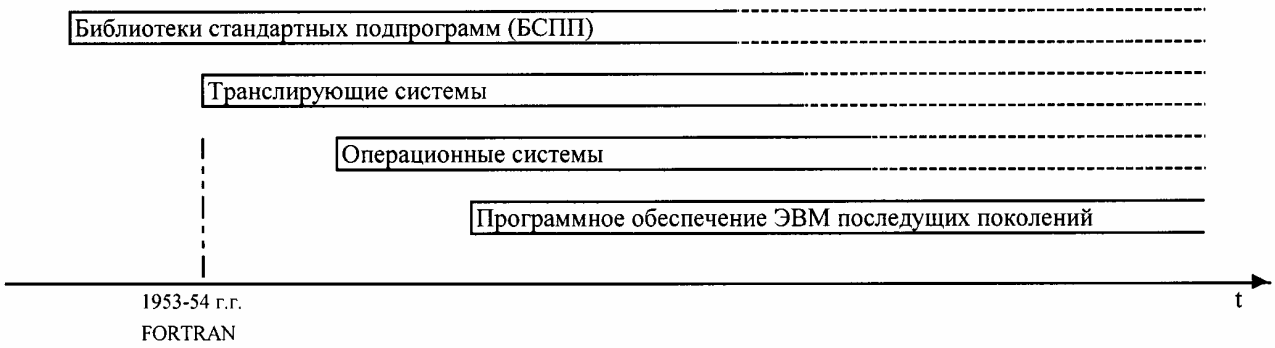
Дополнительная

1. . Свердлов, С. З. Языки программирования и методы трансляции : учебное пособие для вузов по направлению "Прикладная математика и информатика" / С. З. Свердлов .— СПб. [и др.] : Питер, 2007 .— 637 с. : ил ; 24 см + 1 электрон. опт. диск (CD-Rom) .— (Учебное пособие) .— Издательская программа "300 лучших учебников для высшей школы" .— Библиогр.: с.632-637 Электронная версия доступна по адресу: http://chursinvb.ucoz.ru/load/sverdlov_sz_jazyki_programmirovanija_i_metody_transljacii/1-1-0-46
2. Ахо Альфред, Сети Рави, Ульман Джеффри. Компиляторы: принципы,технологии, инструменты.: Пер. с англ. – М.: Издательский дом «Вильямс», 2001. – 768 с.
3. Опалева Э.А., Самойленко В.П., Языки программирования и методы трансляции. – СПб: БХВ-Петербург, 2005. –480 с. : ил.
4. Кузьмин А.А., Никонов В.В., Цыган В.Н. Программное обеспечение ЭВМ: Учебное пособие – Л.: ЛПИ, 1986.
5. Карпов Ю.Г. Основы построения компиляторов: Учебное пособие – Л.: ЛПИ, 1982.
6. Хантер Р. Проектирование и конструирование компиляторов: Пер. с англ. – М.: Финансы и статистика, 1984.
7. Льюис Ф., Розенкранц Д., Стирнз Р. Теоретические основы проектирования компиляторов: Пер. с англ. – М.: Мир, 1979.
8. Грис Д. Конструирование компиляторов для цифровых вычислительных машин: Пер. с англ. – М.: Мир, 1985.

Приложение
Раздаточный материал к лекциям по курсу "Транслирующие системы"

Будем называть "системой программного обеспечения" совокупность программ, которые должны или могут функционировать совместно.

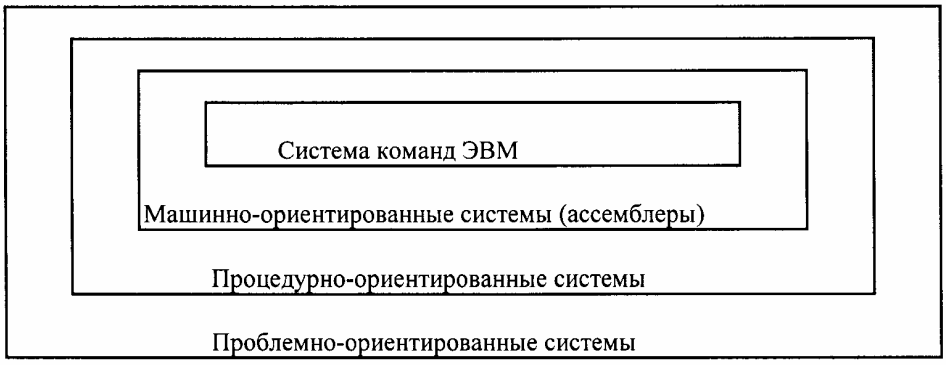
Этапы развития систем программного обеспечения



Введем понятие "система программирования". Будем называть системой программирования совокупность языка программирования и транслятора.

Классификация систем программирования (три способа)

1) По степени "удаленности" от системы команд



2) По характеру использования

- Для программирования
  - Системного
    - Ассемблеры
    - "Си"
    - Форт
    - Паскаль (частично)
    - другие
  - Пользовательского
- Вспомогательные для:
  - загрузчиков
  - редакторов
  - компоновщиков
  - отладчиков
  - интерфейсов ОС
  - и т.п.

3) По способу использования

- Алгоритмические
- Неалгоритмические (например, системы логического программирования)

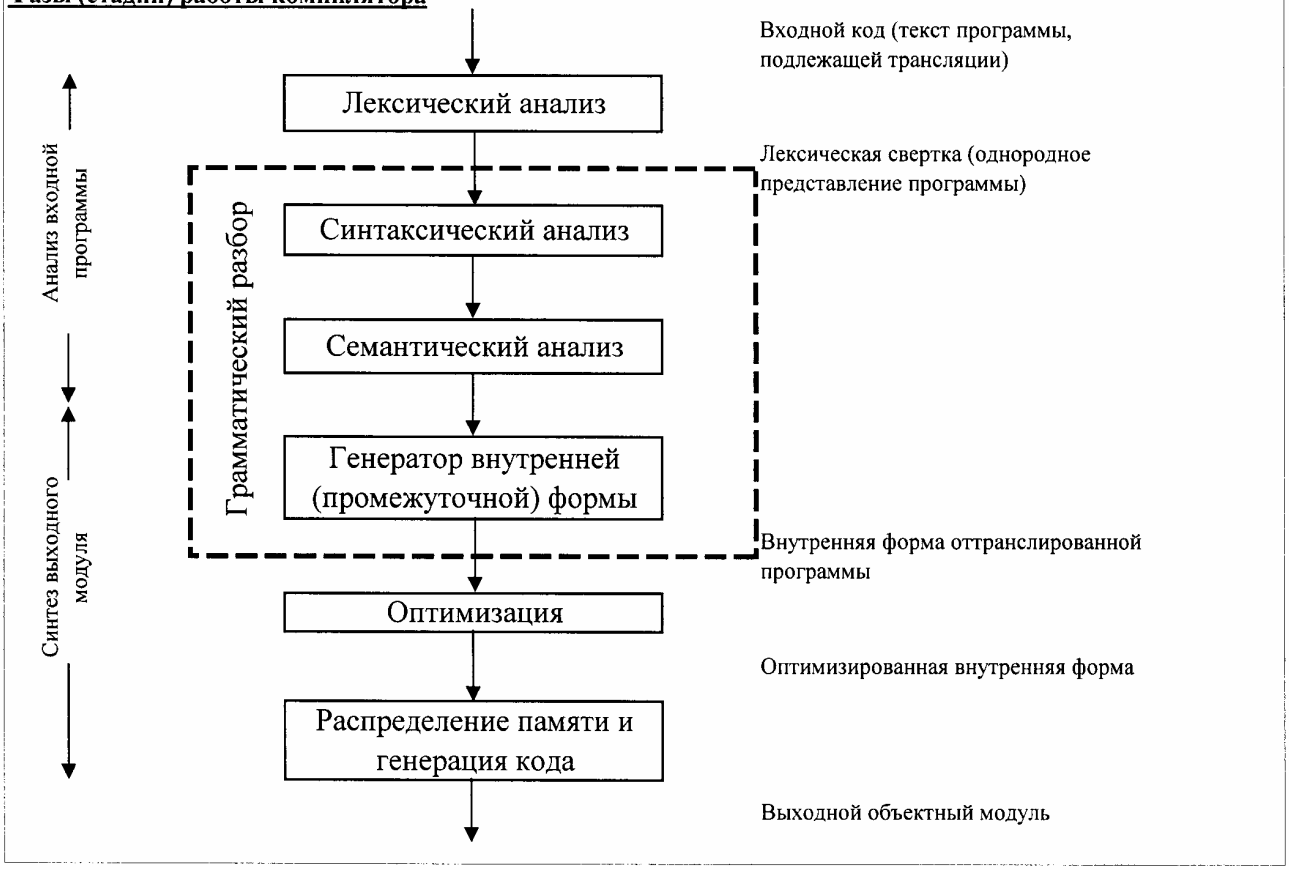
Подлежащие изучению типы трансляторов:

1) Ассемблеры.
2) Интерпретаторы.
3) Компиляторы.

Достоинства языков высокого уровня:

1) ЯВУ более естественны (по сравнению с ассемблерами), а потому легче изучаются.
2) Программы, написанные на ЯВУ, легче отлаживаются.
3) Программы на ЯВУ во многом самодокументированы.
4) Повышается производительность труда программистов, выраженная в количестве команд выходного объектного кода.
5) С учетом оптимизирующих возможностей современных компиляторов возможно получение высоко эффективных выходных кодов.

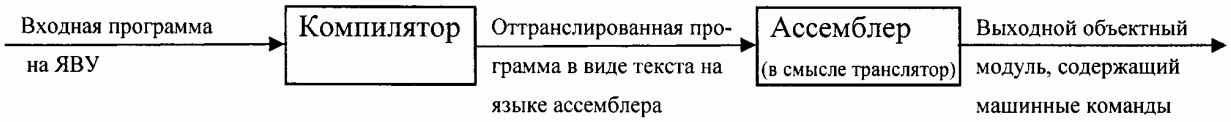
Фазы (стадии) работы компилятора



Виды объектных модулей:

а) абсолютные; б) перемещаемые; в) в виде текста на языке ассемблера.

В случае в) может быть использована следующая схема трансляции:



Количество проходов компилятора:

Минимальное теоретически достижимое количество проходов компилятора лежит в диапазоне от 1 до 3 (для существующих в настоящее время языков) и зависит от свойств конкретного языка.

В практических реализациях компиляторов количество проходов может превышать минимальное теоретически достижимое количество, так как зависит от ряда факторов (требований к качеству трансляции, субъективных желаний разработчика компилятора и т.п.). Втречаются компиляторы с количеством проходов до 9.

Формы передачи информации от одной фазы (стадии) работы компилятора к другой:

а) таблицы ;

б) скомпилированные команды (в том числе, обобщенные).

Элементы теории формальных языков

Вопросы, подлежащие изучению:

1) Понятие грамматики, как средства для задания (описания, определения) языка.
2) Бэкуса-Наура формы (БНФ) для записи правил грамматик.
3) Рекурсия в правилах грамматик. Понятие "эквивалентные грамматики".
4) Классификация грамматик по Хомскому. Необходимые распознающие агрегаты.
5) Классификация языков по Хомскому.

На данном этапе рассмотрения пока не строго определим формальные языки в противоположность естественным языкам (русскому, английскому и т.п.), как языки каким-либо формальным способом определенные и имеющие ограниченную по сравнению с естественными языками сферу применения. Опубликованы различные определения формального языка, например в Википедии дано следующее определение:

*"В математической логике и информатике **формальный язык** — это множество конечных слов (строк, цепочек) над конечным алфавитом. Понятие языка чаще всего используется в теории автоматов, теории вычислимости и теории алгоритмов. Научная теория, которая имеет дело с этим объектом, называется теорией формальных языков."*

Языки программирования являются ярким примером формальных языков.

Возникает вопрос о способе формального задания языка. Если бы количество допустимых предложений какого-либо формального языка было конечным, то язык можно было бы задать перечислением допустимых предложений. Однако для большинства формальных языков, имеющих практическое значение, в т.ч. алгоритмических, предположение о конечности количества допустимых предложений не выполняется. Таким образом, необходим некий специальный формализм для задания (определения, описания) языка. Обычно подобный формализм называют "грамматикой". (В дальнейшем мы увидим, что, в первую очередь, речь идет об описании синтаксиса языка.)

Первое определение грамматики. Пусть грамматика есть конечное, непустое множество правил, позволяющих генерировать все допустимые предложения языка. Замечание: при анализе предложений (в том числе, в компиляторах) эти же правила используются, так сказать, "обратным ходом".

Известны и используются на практике различные способы отображения (формы записи) правил грамматик. Исторически первой формой записи правил стала так называемая Бэкуса-Наура форма (БНФ). Общий вид правила в БНФ таков:

левая часть ::= правая часть

Читаются подобные правила следующим образом: "левая часть есть правая часть" или "левая часть выводится как правая часть". Сами правила в БНФ называют или правилами грамматики, или правилами подстановки, или продукциями.

Символы, используемые в БНФ:

- терминальные (неделимые символы языка);
- нетерминальные или **<нетерминалы>** , служащие для отображения промежуточных фаз генерации допустимого предложения (замечания: а) нетерминальные символы будем заключать в угловые скобки, б) допустимые предложения могут состоять только из терминальных символов);
- | - символ альтернативной подстановки ("или");
- ε - символ пустого множества.

Один из нетерминальных символов грамматики играет особую роль и называется "начальным символом" грамматики. Для него нет строгого формального определения, так как начальный символ определяется составителем грамматики аксиоматически, а потому начальный символ часто называют "аксиомой грамматики". Можно лишь не строго определить, что начальный символ грамматики – это тот символ, ради грамматического описания которого составляется вся грамматика.

Однако при первом же использовании БНФ для записи правил грамматик выясняется, что без использования какого-либо специального приема не удастся с помощью конечного множества правил определить возможность генерации бесконечного множества допустимых предложений. Этим специальным приемом служит введение рекурсии в правила грамматик.

Для подраздела «Элементы теории формальных языков»

Рекурсия в правилах грамматик

Типы рекурсивных грамматик

1) будем называть грамматику непосредственно леворекурсивной, если она содержит в себе правила вида:

<нетерминал A> ::= < нетерминал A > B ;

2) будем называть грамматику непосредственно праворекурсивной, если она содержит в себе правила вида:

<нетерминал A> ::= B < нетерминал A >;

3) будем называть грамматику грамматикой с непосредственным самовосстановлением, если она содержит в себе правила вида:

<нетерминал A> ::= B < нетерминал A > C ,

где **B** и **C** – терминалы, или нетерминалы, или любые их сочетания.

Определение. Две грамматики называются эквивалентными, если они описывают в точности один и тот же язык.

Еще одно определение грамматики

Пусть грамматика G определена как:

G = (V_T , V_N , S , P),

где V_T – алфавит терминальных символов;

V_N – алфавит нетерминальных символов;

S – начальный символ (аксиома) грамматики;

P – множество правил подстановки вида $\alpha ::= \beta$.

Отметим некоторые свойства элементов данного определения.

Очевидно, что пересечение $V_T \cap V_N = \varepsilon$ (ε – обозначение пустого множества),

S принадлежит множеству V_N .

Введем понятие объединенного алфавита V ,

т.е. $V = V_T \cup V_N$.

Обозначим через

V^* множество всех предложений по алфавиту V, включая ε ,

V^+ множество всех предложений по алфавиту V, исключая ε ,

тогда очевидно $\beta \in V^*$, $\alpha \in V^+$.

Классификация грамматик по Хомскому

1) Тип 0. Нет ограничений на вид правил подстановки $\alpha ::= \beta$.

2) Тип 1. Контексно-зависимые (они же – неукорачивающие) грамматики.

Здесь имеет место контекстная зависимость в правилах подстановки, которую символически можно отобразить как B <A> C ::= B a C ,

кроме того $|\alpha| \leq |\beta|$, где $|\alpha|$ и $|\beta|$ обозначают количество символов в α и β , соответственно.

3) Тип 2. Контекстно-независимые (контекстно-свободные) грамматики.

Здесь α - всегда единственный нетерминал.

4) Тип 3. Автоматные (регулярные) грамматики.

Здесь кроме ограничений из пункта 3 есть еще ограничения и на вид правой части всех правил, а именно – β может иметь только вид:

терминал <нетерминал> | <нетерминал> терминал | терминал | ε .

О соответствии между типом грамматики и типом распознающего агрегата.

Наконец обратимся к приведенной в раздаточном материале четырех типовой классификации грамматик по Хомскому. Данная классификация является общепризнанной. Для нас же важно отметить тот доказанный в теории формальных языков факт о соответствии типа грамматики и типа так называемого распознающего агрегата, необходимого для распознавания заданного грамматикой языка. Ниже приведена таблица соответствия.

Тип грамматики по Хомскому	Тип распознающего агрегата
Тип 0	Машина Тьюринга (с бесконечной лентой)
Тип 1	Машина Тьюринга с конечной лентой
Тип 2	Конечный автомат со стековой (магазинной) памятью
Тип 3	Конечный автомат

Из приведенного соответствия можно сделать важные выводы, касающиеся языков программирования и компиляторов. Легко заметить, что типы грамматик перечислены в нашей таблице в порядке, так сказать, убывания мощности, соответственно, типы распознающих агрегатов – в порядке убывания сложности. А ведь распознающая часть компилятора неизбежно будет программной реализацией того или иного распознающего агрегата, таким образом, чем проще будет грамматика, описывающая язык, тем проще будут распознающие программы компилятора. При конструировании языков программирования сложилась естественная тенденция к тому, чтобы для описания языков преимущественно использовались автоматные и контекстно-независимые грамматики (типы 3 и 2 по Хомскому). В противном случае, например, для языка, описанного грамматикой типа 1, распознающим агрегатом стала бы ЭВМ в целом со всеми ее ресурсами, а для случая грамматики типа 0 нам вообще не удастся найти физическую реализацию распознающего агрегата.

Классификация языков по Хомскому.

Для языков используется аналогичная описанной выше четырех типовая классификация, однако здесь имеется особенность. Действительно, как уже отмечено выше, для любого языка существует сколь угодно большое множество эквивалентных грамматик, при этом вполне может так случиться, что отдельные грамматики указанного множества будут принадлежать к разным типам по классификации Хомского. Как же определить истинный тип языка? Истинный тип языка определяется по типу грамматики наименьшей мощности среди всего множество эквивалентных грамматик данного языка.

В заключение данного раздела предлагается ознакомиться с примерами грамматик типа 1 (контекстно-зависимых) и выполнить предлагаемые контрольные задания (см. раздаточный материал).

Для подраздела «Элементы теории формальных языков»

Примеры языков и грамматик типа 1 по Хомскому (контекстно-зависимых)

Пусть задан язык L_3

$$L_3 = \{ a^{(2 \text{ в степени } n)} \mid n => 0 \}$$

Соответственно допустимыми цепочками языка являются следующие:

a, aa, aaaa, aaaaaaaa, ...

и т.д. все цепочки, в которых символ **a** повторяется (2 в степени n) раз.

Грамматикой данного языка является грамматика G_3

$$G_3 = (\{ a \}, \{ S, N, Q, R \}, S , P),$$

где **P** (множество правил подстановки) :

a) <S> ::= <Q> <N> <Q>
b) <Q> <N> ::= <Q> <R>
c) <R> <N> ::= <N> <N> <R>
d) <R> <Q> ::= <N> <N> <Q>
e) <N> ::= a
f) <Q> ::= ε

Задание для самостоятельного выполнения: вывести цепочку **aaaa** с использованием правил грамматики G_3 .

–

Еще один пример грамматики и языка типа 1.

Пусть задан язык L_4

$$L_4 = \{ a^n \ b^n \ c^n \mid n > 0 \}$$

Грамматикой данного языка является грамматика G_4

$$G_4 = (\{ a, b, c \}, \{ S, B, C \}, S , P),$$

где **P** (множество правил подстановки) :

a) <S> ::= a <S> <B> <C>
b) <C><B> ::= <B> <C>
c) b <B> ::= bb
d) c <C> ::= cc

Задание для самостоятельного выполнения: вывести цепочку **aaabbbccc** с использованием правил грамматики G_4 .

Для раздела «Лексический анализ»

Грамматика десятичных чисел

a) <десятичное число> ::= <десятичное число> <цифра> |<смешанное число> <цифра>
b) < смешанное число > ::= <знак> <целое> <десятичная точка> | <целое> <дес. точка>
c) <целое> ::= <целое> <цифра> | <цифра>
d) <знак> ::= + | –
e) <цифра> ::= 0 | 1 | 2 | 3 | . . . | 9
f) < десятичная точка > ::= .

Для раздела «Грамматический разбор»

Грамматика десятичных чисел

a) < десятичное число > ::= <целая часть> <дробная часть>
b) < целая часть > ::= + <целое без знака> | – <целое без знака> | <целое без знака>
c) <дробная часть> ::= .<целое без знака>
d) <целое без знака> ::= <цифра> <целое без знака> | <цифра>
e) <цифра> ::= 0 | 1 | 2 | 3 | . . . | 9

Грамматика арифметических выражений

Обозначения в грамматике.
Нетерминальные символы:
<АВ> – арифметическое выражение;
<М> – многочлен;
<ОД> – одночлен;
<С> – сомножитель.

Терминальные символы:
i – любой идентификатор, выявленный лексическим анализатором;
\n – символ ограничителя строки;
(– открывающая скобка;
) – закрывающая скобка.

Правила грамматики:
a) <АВ> ::= <М> \n
b) <М> ::= <ОД> + <М> | <ОД>
c) <ОД> ::= <С> * <ОД> | <С>
d) <С> ::= (<М>) | i

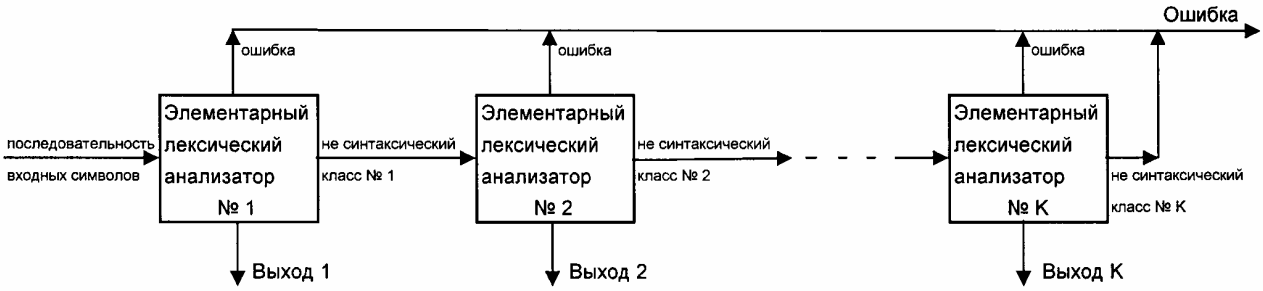
Заключительные замечания по разделу "Лексический анализ"

1. Лексический анализ может быть выполнен или как отдельный проход компилятора, или как подпрограмма вызываемая по мере надобности во время выполнения грамматического разбора.

2. Лексический анализатор для принятия правильного решения каждый раз должен просматривать последовательность входных символов максимально возможной длины.

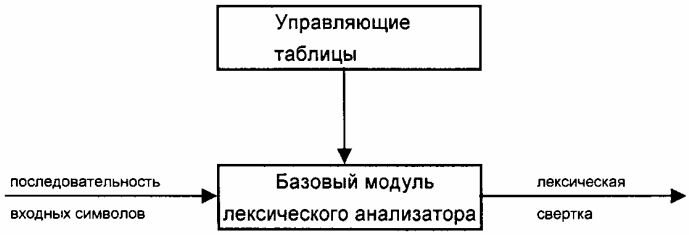
3. Лексический анализатор, способный распознавать лексемы только одного синтаксического класса, будем называть элементарным лексическим анализатором. Тогда встает вопрос о том, как построить лексический анализатор в целом, то есть такой анализатор, который будет распознавать лексемы для случая произвольного (но конечного) количества синтаксических классов.
Приведем в качестве примеров два типовых способа построения подобных лексических анализаторов.

3.1. Процедурный принцип построения ЛА.
Пусть подлежащие распознаванию лексемы разделены на K синтаксических классов. Тогда элементарные лексические анализаторы могут быть объединены по следующей схеме.



Благополучные выходы из процесса распознавания лексем

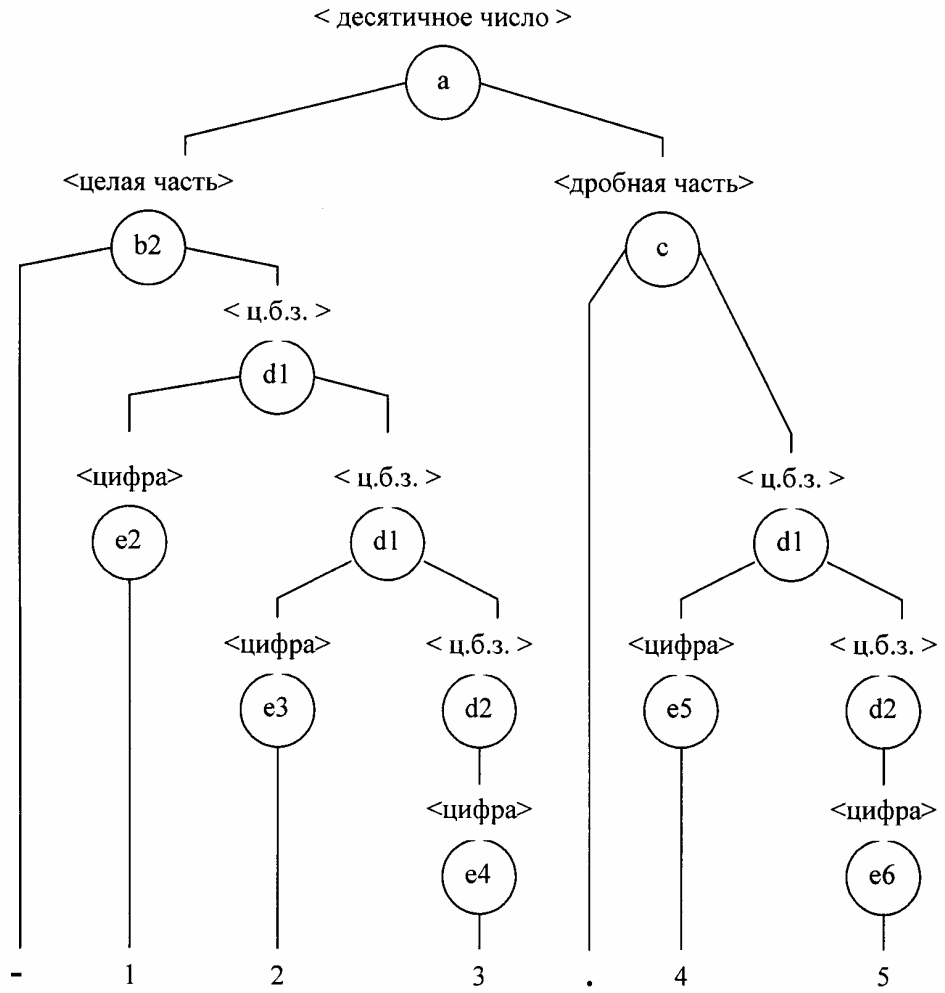
3.2. Таблично-управляемый лексический анализатор.
Возможность построения подобного анализатора базируется на доказанном в теории формальных языков факте, состоящем в том, что независимо от количества синтаксических классов и их свойств в программе лексического анализатора может быть выделена инвариантная по отношению к синтаксическим классам функциональная часть. Будем называть ее базовым модулем лексического анализатора.
В качестве управляющей структуры данных по отношению к базовому модулю лексического анализатора используются так называемые управляющие таблицы. Для настройки лексического анализатора в целом на распознавание конкретных лексем требуется генерация соответствующих управляющих таблиц.
Структура таблично-управляемого лексического анализатора может быть представлена следующим образом.



Каждый из двух приведенных способов построения лексических анализаторов (3.1. и 3.2.) имеет свои достоинства и недостатки, а также области целесообразного применения.

Раздаточный материал для раздела "Грамматический разбор"

Синтаксическое дерево для числа –123.45



«Левый вывод» числа –123.45

<десятичное число> $\xrightarrow{a}$ <целая часть> <дробная часть> $\xrightarrow{b2}$ - <ц.б.з.> <дробная часть> $\xrightarrow{d1}$
$\xrightarrow{d1}$ – <цифра> <ц.б.з.> <дробная часть> $\xrightarrow{e2}$ –1 <ц.б.з.> <дробная часть> $\xrightarrow{d1}$
$\xrightarrow{d1}$ –1 <цифра> <ц.б.з.> <дробная часть> $\xrightarrow{e3}$ –1 2 <ц.б.з.> <дробная часть> $\xrightarrow{d2}$
$\xrightarrow{d2}$ –1 2 <цифра> <дробная часть> $\xrightarrow{e4}$ –1 2 3<дробная часть> $\xrightarrow{c}$ –1 2 3 . <ц.б.з.> $\xrightarrow{d1}$
$\xrightarrow{d1}$ –1 2 3 . <цифра> <ц.б.з.> $\xrightarrow{e5}$ –1 2 3 . 4 <ц.б.з.> $\xrightarrow{d2}$ –1 2 3 . 4 <цифра> $\xrightarrow{e6}$ –123.45

«Правый вывод» числа –123.45

<десятичное число> $\xrightarrow{a}$ <целая часть> <дробная часть> $\xrightarrow{c}$ <целая часть> . < ц.б.з > $\xrightarrow{d1}$
$\xrightarrow{d1}$ <целая часть> . <цифра> <ц.б.з.> $\xrightarrow{d2}$ <целая часть> . <цифра> <цифра> $\xrightarrow{e6}$
$\xrightarrow{e6}$ <целая часть> . <цифра> 5 $\xrightarrow{e5}$ <целая часть> . 4 5 $\xrightarrow{b2}$ – <ц.б.з.> . 4 5 $\xrightarrow{d1}$
$\xrightarrow{d1}$ – <цифра> <ц.б.з.> . 4 5 $\xrightarrow{d1}$ – <цифра> <цифра> <ц.б.з.> . 4 5 $\xrightarrow{d2}$
$\xrightarrow{d2}$ – <цифра> <цифра> <цифра> . 4 5 $\xrightarrow{e4}$ – <цифра> <цифра> 3 . 4 5 $\xrightarrow{e3}$
$\xrightarrow{e3}$ – <цифра> 2 3 . 4 5 $\xrightarrow{e2}$ –123.45

Грамматический разбор сверху-вниз с возвратами

Пусть дана грамматика десятичных чисел:

a) < десятичное число > ::= <целая часть> <дробная часть>
b) < целая часть > ::= + <целое без знака> | – <целое без знака> | <целое без знака>
c) <дробная часть> ::= .<целое без знака>
d) <целое без знака> ::= <цифра> <целое без знака> | <цифра>
e) <цифра> ::= 0 | 1 | 2 | 3 | . . . | 9

Пусть на вход грамматического анализатора было подано число **–123.45**
Диаграмма (таблица), иллюстрирующая процесс разбора *(в приведенной ниже диаграмме заполнены только две первые строки, следует самостоятельно продолжить заполнение диаграммы)* :

№ шага	Строка цели или подцели		Цель	Правило	Справедливо	После возврата	
	Уже идентифицированная	Текущая литера				Правило	Справедливо
1		–	Дес. Число	a			
2		–	Целая Часть	b1			
3		–					
4		–					
5							
6							
7							
8							
9							
10							

Линейное представление синтаксического дерева.

Пусть дана грамматика десятичных чисел:

a) < десятичное число > ::= <целая часть> <дробная часть>
b) < целая часть > ::= + <целое без знака> | – <целое без знака> | <целое без знака>
c) <дробная часть> ::= .<целое без знака>
d) <целое без знака> ::= <цифра> <целое без знака> | <цифра>
e) <цифра> ::= 0 | 1 | 2 | 3 | . . . | 9

Пусть на вход грамматического анализатора было подано число **-123.45**
Тогда так называемое линейное представление синтаксического дерева (дерева разбора), записанное в скобочной форме, будет иметь вид:

Дес.число (Целая часть(-ц.б.з.(цифра(1) ц.б.з.(цифра(2) ц.б.з.(цифра(3)))))
Дробная часть(. ц.б.з.(цифра(4) ц.б.з.(цифра(5)))))

Принцип расстановки скобок следующий: при входе в какое-либо распознающее правило в линейную форму синтаксического дерева вставляется открывающая скобка, а при каждом выходе из распознающего правила - закрывающая скобка. Убедитесь, что именно приведенная линейная форма синтаксического дерева (конечно, без скобок) соответствует последовательности целей (см. поле "цель" диаграммы разбора), формируемых распознающим автоматом в процессе грамматического разбора методом "сверху-вниз".

Особенности грамматического разбора "сверху-вниз с возвратами"

1) Небезразличен порядок проверки альтернативных правил: он должен быть таким, чтобы проверка кокого-либо правила не заблокировала проверку другого правила.
2) Невозможность обработки прямой левой рекурсии, так как она (левая рекурсия) находится в логическом противоречии с каноническим разбором "сверху-вниз".
3) Исключительно малая скорость работы, обусловленная возвратами.
   Возможные способы ускорения разбора:
   3.1. "Заглядывание вперед", т.е. такая организация разбора, при которой на каждом шаге разбора анализируется не единственный очередной символ, а 2 символа, или 3, или 4 … Такой способ разбора привлекателен по своей идее, но на практике применяется редко из-за существенно возрастающей разветвленности алгоритма разбора. Количество ветвей алгоритма на каждом шаге разбора будет определяться как количество сочетаний $C(n,k) = (n!) / ((n-k)! * k!)$, где n - количество символов алфавита, k - количество символов, анализируемых на очередном шаге разбора. Например, $C(13,2) = 78$.
   3.2. Эквивалентные преобразования грамматики, управляющей разбором, по критерию минимизации количества альтернатив в продукциях грамматики.
   3.3. Организация безвозвратного грамматического разбора (так называемый рекурсивный спуск), но для его реализации придется наложить на грамматику дополнительные ограничения, о которых будет сказано позже.
   3.4. Главное достоинство рассматриваемого метода грамматического разбора состоит в том, что сам разбор управляется непосредственно правилами грамматики.

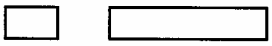
Раздаточный материал к подразделу "Синтаксические диаграммы"

Вводные тезисы

Синтаксическая диаграмма для каждого нетерминала задает такой помеченный граф с одним входом и одним выходом, что последовательность пометок при движении по любому пути от входа к выходу графа определяет возможную конечную подстроку, которой может быть заменен этот нетерминал при выводе цепочки языка [Карпов Ю.Г.]. Таким образом синтаксические диаграммы могут служить средством отображения: а) порождающих правил грамматик (наряду с Бэкуса-Наура формами); б) процесса распознавания цепочек (т.е. процесса трансляции).

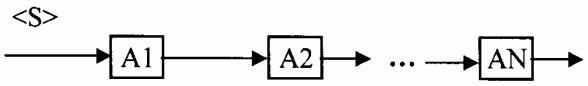
Символы синтаксических диаграмм

кругами или овалами отображают терминальные символы

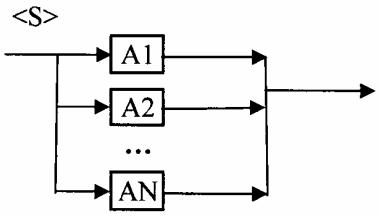


квадратами или прямоугольниками отображают нетерминальные символы

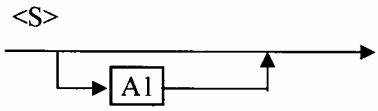
Типовые конструкции синтаксических диаграмм — Соответствующие Бэкуса-Наура формы



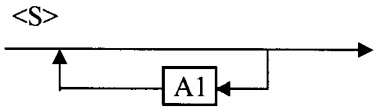
<S> ::= <A1> <A2> ... <AN>



<S> ::= <A1> | <A2> | ... | <AN>



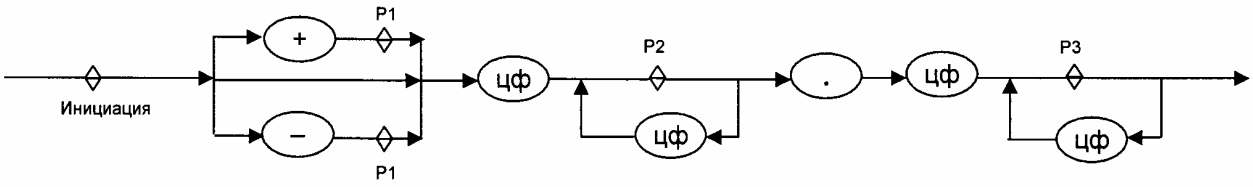
<S> ::= [<A1>]
(РБНФ)



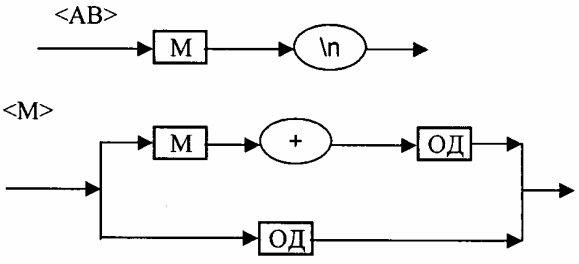
<S> ::= {<A1>}
(РБНФ)

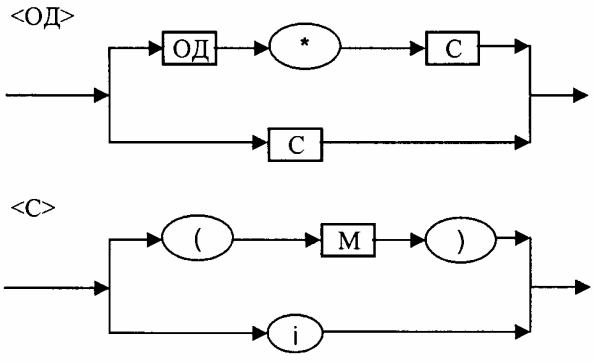
Пример автоматной грамматики (грамматики десятичных чисел), представленной в виде синтаксической диаграммы

<десятичное число>



Пример контекстно-независимой грамматики (в данном случае грамматики арифметических выражений), представленной в виде совокупности синтаксических диаграмм





Генерация кода и машинно-зависимая оптимизация

Простейший способ генерации кода состоит в использовании так называемых таблиц попрождаемого кода.

Пример возможного содержания таблиц порождамого кода в зависимости от "адресности" ЭВМ.

Строка матрицы синтаксического дерева	Генерируемый код		
	Для 3-х адресной ЭВМ	Для 2-х адресной ЭВМ	Для одно-адресной ЭВМ
TN + ОП1 ОП2 (сложение)	СЛЖ ОП1, ОП2, TN	ПЕР ОП1, R1 СЛЖ ОП2, R1 ПЕР R1, TN	ЧТЕ ОП1 СЛЖ ОП2 ЗАП TN
TN * ОП1 ОП2 (умножение)	УМН ОП1, ОП2, TN	ПЕР ОП1, R1 УМН ОП2, R1 ПЕР R1, TN	ЧТЕ ОП1 УМН ОП2 ЗАП TN
. . . .	. . .	. . .	. . .

Пример генерации неоптимизированного и оптимизированного кода для арифметического выражения X+Y*(Z+X)+A для 2-х адресной ЭВМ (код приведен с использованием языка ассемблера гипотетической ЭВМ).

Тетрады матрицы синтаксического дерева	Порождаемый код		
	№ п/п	Неоптимизированный	Машинно-зависимо оптимизированный с использованием единственного РОНа
T1 + Z X	1 2 3	ПЕР Z, R1 СЛЖ X, R1 ПЕР R1, T1	ПЕР Z, R1 СЛЖ X, R1
T2 * Y T1	4 5 6	ПЕР Y, R1 УМН T1, R1 ПЕР R1, T2	УМН Y, R1
T3 + X T2	7 8 9	ПЕР X, R1 СЛЖ T2, R1 ПЕР R1, T3	СЛЖ X, R1
T4 + T3 A	10 11 12	ПЕР T3, R1 СЛЖ A, R1 ПЕР R1, T4	СЛЖ A, R1 ПЕР R1, T4
	12 команд 4 ячейки временной памяти		6 команд 1 ячейка временной памяти

Генерация неоптимизированного кода эквивалентна безусловным вызовам макросов, где именем макроса служит оператор, а фактическими параметрами - операнды и ячейка временной памяти.

Генерация машинно-зависимо оптимизировнного кода эквивалентна условным вызовам макросов. При этом используется оптимизирующая таблица порождаемого кода, содержащая для каждого оператора несколько вариантов порождаемого кода.
Примеры типовых условий:
1) Находится ли уже один из операндов в РОНе.
2) Нужно ли сохранять результат операции.
3) Имеются ли еще свободные РОНы.
4) Имеют ли операнды одни и те же тип и длину, а если нет, то в выходной объектный модуль должны быть вставлены вызовы подпрограмм, приводящих операнды к одинаковым типу и длине.
5) Учет при генерации кода относительного быстродействия выполнения альтернативных команд.

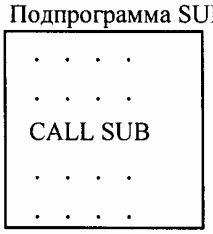
Распределение памяти как фаза работы компилятора (тезисы).

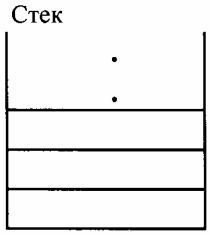
Глубина осуществляемого компилятором распределения памяти зависит от вида выходного объектного модуля. Если выходной объектный модуль таков, что содержит машинный код, то компилятор должен провести полное фактическое распределение памяти (хотя бы с точностью до относительных адресов). Если же выходной модуль является программой на языке ассемблера, то компилятор проводит распределение памяти "на символическом уровне" (генерирует директивы резервирования памяти и т.п.), а фактическое распределение памяти осуществляется ассемблирующим транслятором (ассемблером).
Однако в любом случае распределение памяти должно быть осуществлено для следующих элементов оттранслированной программы:
1) все команды, оставшиеся после всех оптимизаций;
2) все литералы, оставшиеся после всех оптимизаций;
3) все ячейки временной памяти, оставшиеся после всех оптимизаций;
4) все переменные и массивы.

Классы памяти (с точки зрения ее распределения):
1. Статическая (внутренняя и внешняя).
2. Динамическая (всегда внутренняя).
3. Управляемая.

Статическая память распределяется во время компиляции и резервируется на все время выполнения главной программы и ее подпрограмм и процедур. Будем называть внутренней ту память, которая используется внутри процедур и подпрограмм. Соответственно, внешней будем называть ту память, которая используется в главной программе (в том числе, для глобальных объектов).
Статическое распределение памяти - это наиболее простой для реализации способ, но приводящий к наибольшим затратам памяти во время исполнения программы.

Динамическая память используется внутри подпрограмм и процедур и резервируется только на время их исполнения. Два характерных примера (и одновременно - признака) динамического распределения памяти - это
а) возможность использования "динамических" массивов в процедурах и подпрограммах;
б) возможность рекурсивных вызовов внутри подпрограмм (процедур).
Две приведенные ниже диаграммы схематично иллюстрируют, что есть рекурсивный вызов подпрограммы, и как механизм стековой памяти используется для обеспечения корректности рекурсивных вызовов.





Здесь имеется ввиду, что каждый очередной рекурсивный вызов п/п SUB порождает создание очередной области инициализации.

Управляемая память означает наличие в транслируемом языке таких исполняемых операторов, которые позволяют программисту управлять распределением памяти по ходу исполнения программы. Подобные операторы могут, например, иметь имена ALLOCATE … и FREE … ("отвести память" и "освободить память", соответственно).

Раздаточный материал к подразделу "**Потоки информации в компиляторе**"

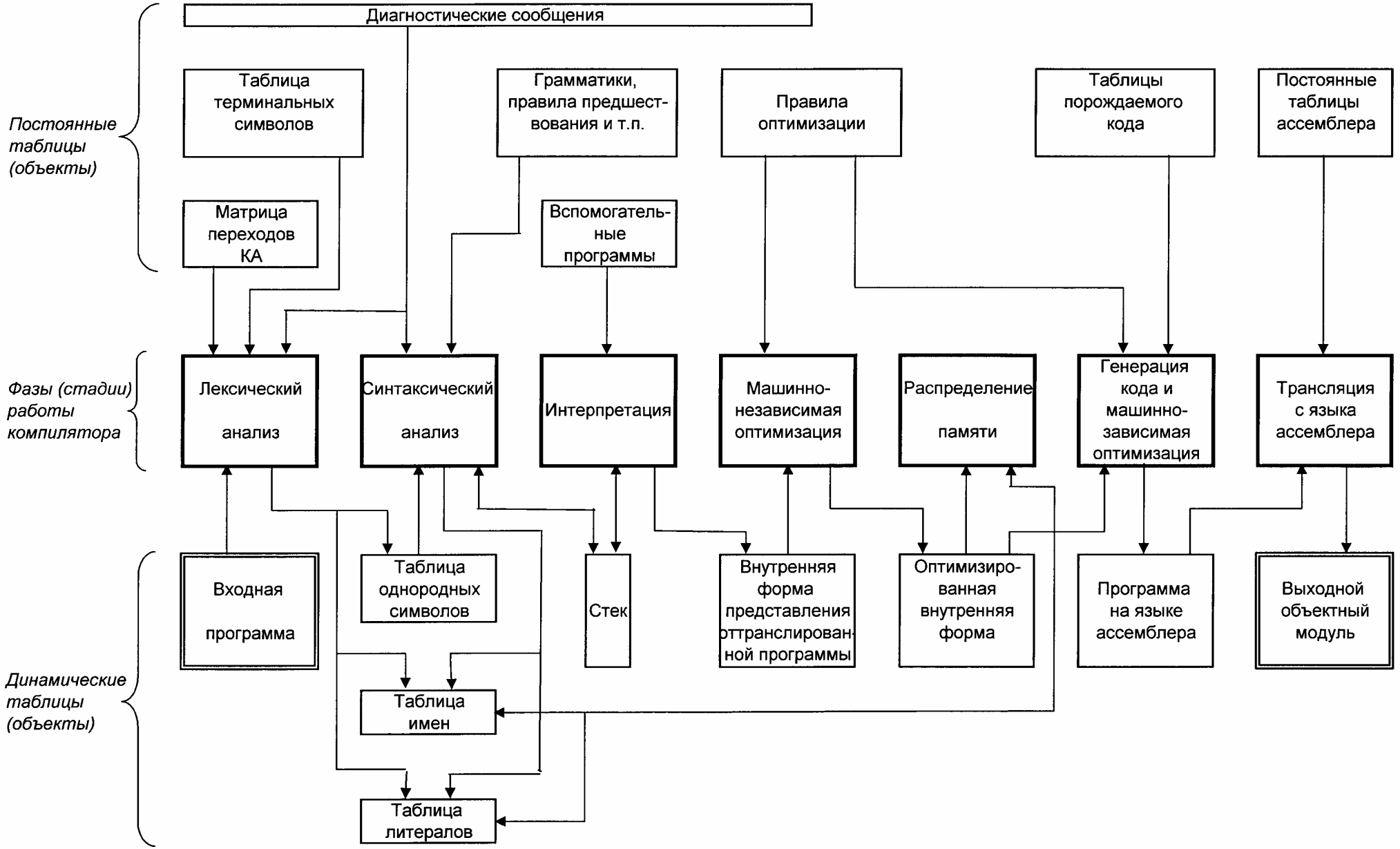


Рис. Потоки информации в компиляторе

Раздаточный материал по теме:

"Использование списковых структур в качестве внутренней формы представления оттранслированной программы в компиляторе"

Будем считать, что связанные списки - такие внутрикомпьютерные структуры данных, где каждый элемент структуры обычно называют записью. В свою очередь, каждая запись состоит из двух частей: информационной и поля указателей. В простейшем случае (в так называемых односвязных списках) поле указателей содержит единственный указатель - указатель на логически следующую запись списка. В общем случае (в так называемых многосвязных списках) поле указателей может содержать несколько указателей.

Ниже приведен пример двусвязного списка, состоящего из четырех записей с логическими номерами от 1 до 4. На рисунке схематично показано, что поле указателей каждой записи (кроме первой и последней) содержит два указателя: указатель на логически следующую запись и указатель на логически предыдущую запись.

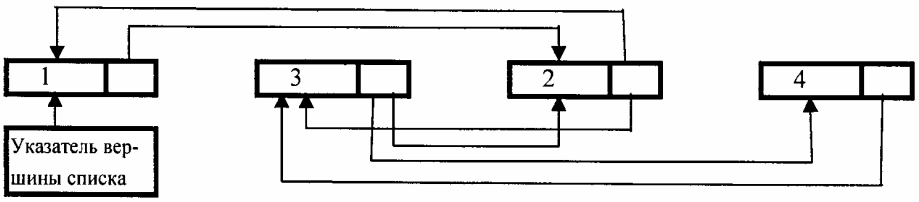
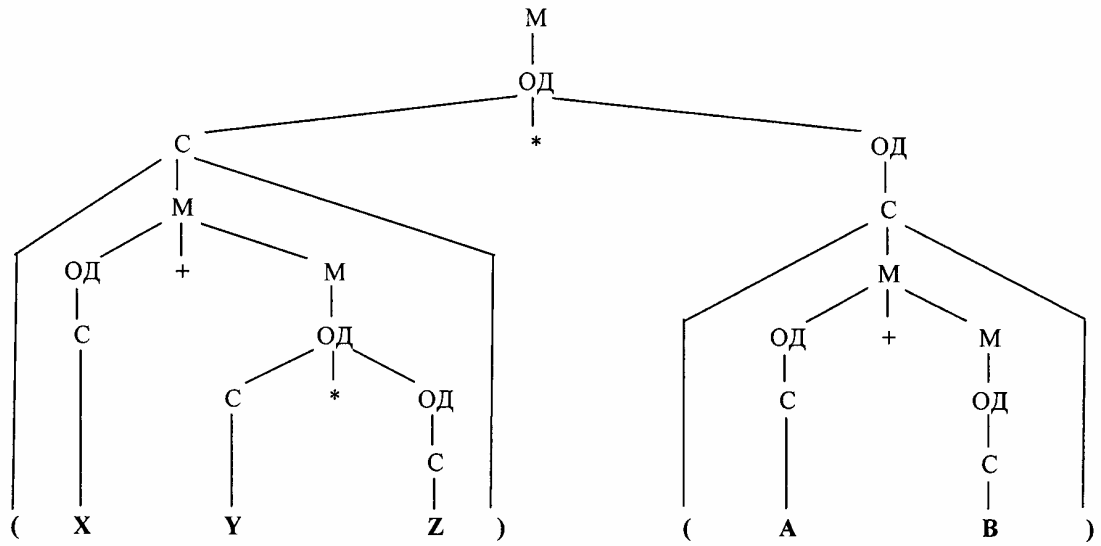


Рис. Пример двусвязного списка

В контексте рассматриваемой темы обсудим два вопроса: 1) как в процессе грамматического разбора построить списковое представление оттранслированной программы; 2) как на основе спискового представления осуществить генерацию кода.

Для большей наглядности привлечем к нашему обсуждению пример. Пусть подлежит трансляции многочлен вида: **(X + Y * Z) * (A + B)** , где X, Y, Z, A, B - идентификаторы, обозначающие переменные.

Построим граф синтаксического дерева для указанного многочлена. Пусть при построении графа используется упоминавшаяся выше праворекурсивная грамматика многочленов.



При изображении графа дерева применены уже использовавшиеся ранее обозначения: М - многочлен; ОД - одночлен; С - сомножитель.

Линейное представление данного синтаксического дерева будет иметь вид:

M [ОД [C [(M [ОД [C [**X**]] + M [ОД [C [**Y**] * OD [C [**Z**]]]]])] *

* ОД [C [(M [ОД [C [**A**]] + M [[ОД [C [**B**]]]])]]]] ,

где: открывающая квадратная скобка используется при каждом входе в распознающее правило, а закрывающая квадратная скобка - при каждом выходе из какого-либо распознающего правила, кроме того, жирным шрифтом выделены терминальные символы.

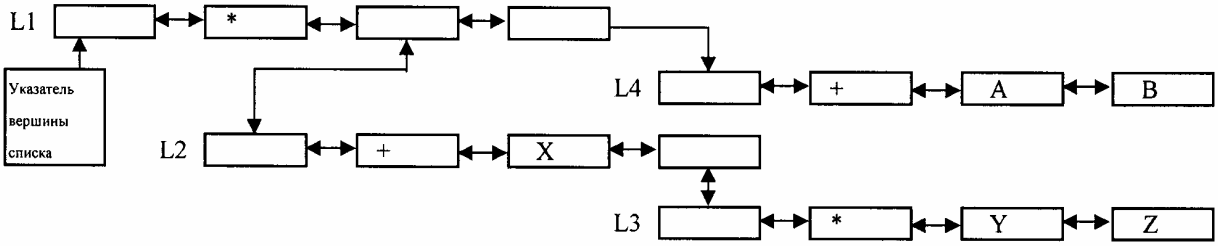
Раздаточный материал по теме:
"Использование списковых структур в качестве внутренней формы представления оттранслированной программы в компиляторе" (продолжение)

Дадим словесное описание алгоритма получения спискового представления синтаксического дерева.

Для получения спискового представления совершается обход синтаксического дерева сверху-вниз. Символы, принадлежащие одной основе, просматриваются слева направо. Подобный обход синтаксического дерева возможен, например, после того как линейное представление дерева образовалось в стеке в процессе синтаксического анализа. Если встретившееся при обходе дерева выражение является простым, т.е. не содержит операций, то необходимо запомнить в ячейке списка значение выражения и выдать адрес ячейки. Если же встретившееся выражение не является простым, то необходимо создать новый (очередной) подсписок Li (где i - номер подсписка) и в этом подсписке выполнить следующие операции:

а) Занести в ячейку подсписка операцию.
б) Проанализировать выражение слева от операции и, если оно простое, то запомнить его значение в ячейке подсписка, а если оно не является простым, то создать новый подсписок.
в) Аналогично проанализировать выражение справа от операции.
г) Продолжать обход дерева и анализ выражений, пока не останутся только простые выражения.
д) Выдать адрес сформированного списка.

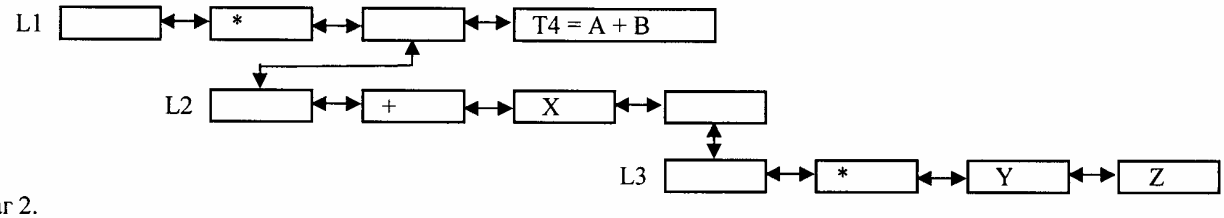
Применительно к нашему примеру дадим графическую интерпретацию спискового представления синтаксического дерева, полученного в соотвествии с описанным алгоритмом.



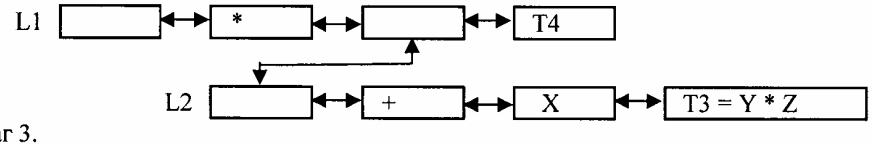
Теперь дадим словесное описание алгоритма генерации объектного кода на основе спискового представления оттранслированной программы. На каждом шаге генерации осуществляется обход полученного списка с целью нахождения подсписка с наибольшим номером, после чего найденный подсписок преобразуется очередную машинную команду и, таким образом, исключается из общего списка, и так далее до подсписка с начальным номером.

Проиллюстрируем работу описанного алгоритма генерации кода на нашем примере. Пусть в данном случае генерируются обобщенные команды, в которых ячейки временной памяти обозначаются как Ti , где i - номер ячейки временной памяти.

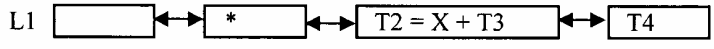
Шаг 1.



Шаг 2.



Шаг 3.



Шаг 4.

Таким образом в результате работы генератора кода получены 4 команды (в данном случае - обобщенные):

T4 = A + B
T3 = Y * Z
T2 = X + T3
T1 = T2 * T4

Отметим одну особенность: полученные команды должны исполняться в порядке убывания номеров ячеек временной памяти.

Раздаточный материал к разделу Синтаксически управляемые процессы трансляции

Пример атрибутной транслирующей грамматики (АТГ) с синтезируемыми атрибутами

Пусть лексический блок некоего компилятора способен вырабатывать и подавать на вход грамматического блока лексемы, принадлежащие следующему множеству:

{ + , * , (,) , k , \n } ,

где **k** – лексема, представляющая константу; ее значением является значение, построенное лексическим блоком во время анализа входного текста; **\n** – лексема, обозначающая ограничитель выражения справа.

Пусть стоит задача спроектировать грамматический блок, который будет распознавать синтаксически корректные арифметические выражения, составленные из символов приведенного выше множества, и выдавать в качестве результата трансляции арифметические значения входных выражений. Попытаемся получить описание проектируемого грамматического блока в форме атрибутной транслирующей грамматики (АТГ).

Обычно конструирование АТГ состоит из трех этапов: 1) составление или выбор входной грамматики; 2) составление транслирующей (но еще не атрибутной) грамматики; 3) составление АТГ. В нашем примере в качестве входной следует выбрать грамматику скобочных арифметических выражений. Что касается транслирующей грамматики, то в нашем простом примере она будет получена из входной путем добавления в первую продукцию символа действия {ОТВЕТ}, который обозначает выдачу на выход арифметического значения выражения. Две обсужденные грамматики приведены в таблице.

Входная грамматика	Транслирующая грамматика
a) <AB> ::= <M> \n	a) <AB> ::= <M> \n {ОТВЕТ}
b) <M> ::= <M> + <ОД>	b) <M> ::= <M> + <ОД>
c) <M> ::= <ОД>	c) <M> ::= <ОД>
d) <ОД> ::= <ОД> * <C>	d) <ОД> ::= <ОД> * <C>
e) <ОД> ::= <C>	e) <ОД> ::= <C>
f) <C> ::= (<M>)	f) <C> ::= (<M>)
g) <C> ::= k	g) <C> ::= k

Теперь приступим к составлению искомой АТГ. Атрибутная транслирующая грамматика должна состоять из продукций транслирующей грамматики, но символы грамматики должны быть оснащены необходимыми атрибутами, и должны быть определены правила вычисления атрибутов.

Обычно описание АТГ начинается с перечисления нетерминальных символов грамматики, оснащенных атрибутами, при этом указывается тип атрибута (синтезируемый или наследуемый):

$<M>_x$, x – синтезируемый;

$<ОД>_x$, x – синтезируемый;

$<C>_x$, x – синтезируемый.

Кроме того, как было сказано выше, у нас один терминальный символ (k) будет оснащен атрибутом: k_x , x – из входной последовательности.

Далее приведем собственно продукции АТГ, при этом ниже каждой продукции будут указаны соответствующие правила вычисления атрибутов *(замечание: в правилах принято использовать символ стрелочки справа налево вместо символов равенства или присвоения)*:

a) $<AB> ::= <M>_p$ \n $\{ОТВЕТ\}_q$ $q \leftarrow p$	e) $<ОД>_r ::= <C>_p$ $r \leftarrow p$
b) $<M>_r ::= <M>_p + <ОД>_q$ $r \leftarrow p + q$	f) $<C>_r ::= (<M>_p)$ $r \leftarrow p$
c) $<M>_r ::= <ОД>_p$ $r \leftarrow p$	g) $<C>_r ::= k_p$ $r \leftarrow p$
d) $<ОД>_r ::= <ОД>_p * <C>_q$ $r \leftarrow p * q$	

В качестве упражнения необходимо построить атрибутное синтаксическое дерево для какого-либо арифметического выражения, например, для (10 + 12) * (7 + 8) \n .

7.6. Пример: наследуемые атрибуты

Теперь мы приведем пример, показывающий, каким образом можно определить атрибутную информацию, так чтобы она распространялась вниз по дереву вывода. Рассмотрим следующую грамматику с начальным символом ⟨описание⟩:

1. ⟨описание⟩ → ТИП V ⟨список переменных⟩
2. ⟨список переменных⟩ →, V ⟨список переменных⟩
3. ⟨список переменных⟩ → ε

Эта грамматика порождает описания, подобные тем, что встречаются во многих языках программирования.

Предположим, что существует лексический блок, задающий три лексемы:

$$V \text{ ТИП},$$

где лексема V обозначает переменную и ее значение является указателем на соответствующий этой переменной элемент таблицы имен; ТИП — лексема со значением, определяющим, какой из типов, ВЕЩЕСТВЕННЫЙ, ЦЕЛЫЙ или ЛОГИЧЕСКИЙ, должен быть поставлен в соответствие переменным из данного списка.

При обработке описания каждой переменной синтаксический блок вызывает процедуру УСТАНОВИТЬ-ТИП, которая помещает один из типов, ВЕЩЕСТВЕННЫЙ, ЦЕЛЫЙ или ЛОГИЧЕСКИЙ, в надлежащее поле элемента таблицы имен, соответствующего данной переменной. Вызов УСТАНОВИТЬ-ТИП лучше всего осуществить сразу после того, как переменная поступила на вход синтаксического блока. Такая синхронизация описывается следующей грамматикой, транслирующей в цепочки, где для обозначения вызова процедуры УСТАНОВИТЬ-ТИП используется символ действия {УСТАНОВИТЬ-ТИП}:

1. ⟨описание⟩ → ТИП V {УСТАНОВИТЬ-ТИП} ⟨список переменных⟩
2. ⟨список переменных⟩ →, V {УСТАНОВИТЬ-ТИП} ⟨список переменных⟩
3. ⟨список переменных⟩ → ε

Предположим, что процедура УСТАНОВИТЬ-ТИП имеет два аргумента: указатель на элемент таблицы имен, соответствующий переменной, и тип переменной. Тогда вызов процедуры УСТАНОВИТЬ-ТИП можно записать так:

УСТАНОВИТЬ-ТИП (указатель, тип)

Мы хотим ввести в вышеприведенную грамматику атрибуты и правила их вычисления, чтобы в последовательностях актов входные символы были представлены вместе с их значениями, играющими роль атрибутов, а вхождения символов действия {УСТАНОВИТЬ-ТИП} имели по два атрибута, представляющих аргументы соответствующего вызова процедуры УСТАНОВИТЬ-ТИП. Тогда вхождения УСТАНОВИТЬ-ТИП будут иметь такой вид:

$\{\text{УСТАНОВИТЬ-ТИП}\}_{\text{указатель, тип}}$

Символ действия {УСТАНОВИТЬ-ТИП} с его атрибутами хорошо иллюстрирует разницу между атрибутом и значением. У этого символа имеется два атрибута — указатель и тип, но только одно значение, которым является пара (указатель, тип). Таким образом, понятие атрибута — это уточнение понятия значения. Так как мы переходим к особенностям модели атрибутной грамматики, теперь мы имеем дело исключительно с атрибутами.

Возвращаясь к задаче порождения требуемой атрибутной последовательности актов, рассмотрим как вхождения {УСТАНОВИТЬ-ТИП} получают свои атрибуты. В правиле 1 это делается просто, так как атрибуты символа действия {УСТАНОВИТЬ-ТИП} можно получить, используя входные символы ТИП и V, входящие в это правило. В правиле 2 ТИП не доступен, и его нужно как-то передать, используя атрибуты нетерминала. Для этой цели мы снабдим нетерминал ⟨список переменных⟩ атрибутом, который будет представлять тип. Требуемая последовательность актов будет порождаться следующими правилами с атрибутами:

1. ⟨описание⟩ $\longrightarrow$ ТИП$_t V_p$ $\{\text{УСТАНОВИТЬ ТИП}\}_{p1,\ t1}$ $\langle\text{список переменных}\rangle_{t2}$

$$(t2,\ t1) \longleftarrow t \qquad p1 \longleftarrow p$$

2. $\langle\text{список переменных}\rangle_t \longrightarrow$, V_p $\{\text{УСТАНОВИТЬ-ТИП}\}_{p1,\ t1}$ $\langle\text{список переменных}\rangle_{t2}$

$$(t2,\ t1) \longleftarrow t \qquad p1 \longleftarrow p$$

3. $\langle\text{список переменных}\rangle_t \longrightarrow \varepsilon$

Запись $(t2,\ t1) \longleftarrow t$ означает, что t присваивается одновременно $t1$ и $t2$. На рис. 7.4 показано атрибутное дерево вывода последовательности

$$\text{ТИП}_{\text{ВЕЩЕСТВЕННЫЙ}}\ V_1,\ V_2,\ V_3$$

определяемое данной грамматикой.

Заметим, что на рис. 7.4, чтобы получить значения атрибутов, соответствующие вхождениям нетерминала ⟨список переменных⟩, используются символы, расположенные выше в дереве вывода, или символы, входящие в ту же правую часть правила. Входной символ ТИП определяет значение ВЕЩЕСТВЕННЫЙ, которое передается самому верхнему вхождению нетерминала ⟨список переменных⟩, а затем передается вниз по дереву другим вхождениям. Такой нисходящий характер вычисления значений атрибутов отражается в том, что каждое правило вычисления атрибутов нетерминалов, сопоставленное продукциям, указывает, как вычислять атрибуты нетерминала, входящего в правую часть продукции. Атрибуты, значения которых задаются таким нисходящим способом, принято называть «наследуемыми» атрибутами.

Сравнивая примеры этого и предыдущего разделов, мы видим, что информация о синтезируемых атрибутах распространяется вверх по дереву, тогда как информация о наследуемых атрибутах — вниз

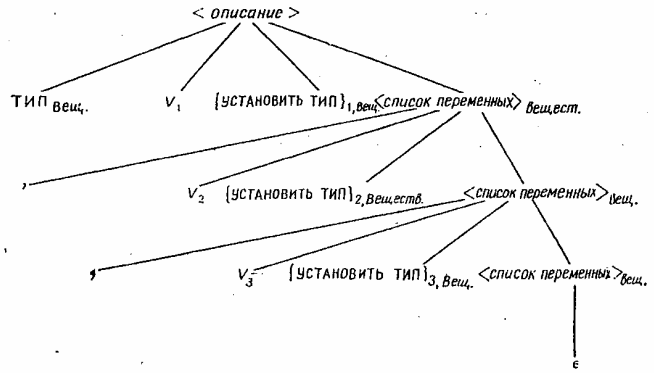


Рис. 7.4.

по дереву. Синтезируемые атрибуты вычисляются по правилам, связанным с нетерминалами, входящими в левую часть продукции, тогда как наследуемые атрибуты — по правилам, связанным с нетерминалами, входящими в правую часть.

К разделу машинно-ориентированные системы программирования (ассемблеры).

Области применения МОС (машинно-ориентированных систем программирования):
1. Системное программирование.
2. Как промежуточный язык в компиляторах.
3. Для программирования микроконтроллеров и т.п. устройств, в тех случаях, когда нецелесообразно разрабатывать транслятор с языка высокого уровня.
4. В учебном процессе.

Структура оператора языка ассемблера:

Поле метки	Поле символического кода оператора	Поле операндов	Поле комментария

Этапы развития МОС.
1. Системы символического кодирования (ССК). ССК по сути были лишь системами символического кодирования операторов, а за распределение памяти отвечал автор программы, т.е. распределял память "вручную".
2. Мнемокоды. В этих системах уже присутствовали элементы автоматического распределения памяти, так как были разрешены к использованию непосредственные данные (литералы), для которых память распределялась транслятором (ассемблером).
3. Автокоды. Это уже развитые ассемблеры, в которых введено символическое именование переменных и массивов, введены так называемые псевдокоманды (директивы), разрешены к применению простейшие арифметические выражения.
4. Макроассемблеры. В макроассемблерах программисту разрешено создавать новые свои собственнные коды операций, которые принято называть макрокомандами.
Практически во всех макроассемблерах принят следующий порядок (механизм) создания макрокоманды: перед первым употреблением макрокоманды (м/к) программист должен в тексте программы дать так называемое макроопределение (описание м/к, прототип м/к).
Приведем структуру макрооопределения, используя символику ассемблера некоторой гипотетической ЭВМ:

Поле метки	Поле КОП	Поле операндов
Имя макрокоманды	МОП	Список формальных аргументов (параметров) м/к
	.	тело макроопределения (макрокоманды)
	.	
	.	
[Имя макрокоманды]	МКОНЕЦ	

Соответственно, вызов макрокоманды будет иметь следующий символический вид:

[Метка]	Имя макрокоманды	Список фактических аргументов (параметров) м/к

На пракике можно встретить два принципа использования макрокоманд:
а) подстановки, когда в каждый момент времени в оттранслирванной и исполняемой программе существует единственный экземпляр тела макрокоманды, а фактические параметры тем или иным способом каждый раз передаются в это единственное тело;
б) генерации, когда транслятор, встречая очередной вызов макрокоманды, каждый раз вставляет в выходной объектный модуль последовательность команд тела макроопределения, но с учетом фактических аргументов (параметров), указанных в строке вызова м/к.
Очевидно, что в случае а) достигается минимизация памяти по сравнению со случаем б), но замедляется выполнение программы; напротив, в случае б) достигается максимальное быстродействие, но объем оттранслированной программы становится трудно предсказуемым, особенно при наличии вложенных макрокоманд.

К подразделу "Общая схема работы ассемблера".

Типы команд, обрабатываемых ассемблером:

1. Мнемокоманды. Транслируются по принципу "один в один" (1--> 1), так как обычно один оператор соотвествует одной исполняемой команде процессора.
2. Псевдокоманды. Транслируются обычно по принципу "один в ноль" (1--> 0), так как псевдокоманды (директивы) несут в себе служебную информацию, используемую во время трансляции.
3. Макрокоманды. Транслируются по принципу "один в N" (1--> N), где обычно N>1. Здесь и далее будем рассматривать только макроассемблеры, работающие по принципу генерации, т.е. макрогенераторы.

Типы адресаций, обрабатываемых ассемблером:

1. Абсолютная. Абсолютная адресация означает, что в поле операндов указан абсолютный ("физический") адрес операнда (естественно, что подобная адресация имеет смысл и может использоваться только в ассеблерах таких компьютеров,где имеет место единое пространсво главной памяти).
2. Символическая. Символическая адресация означает использование в поле операндов символического имени оператора, например, имени метки, переменной , массива.
3. Относительная. Выделим два подтипа относительной адресации:
   3.1. Относительно символической метки.
   3.2. Текущая относительная (т.е. относительно текущей исполняемой команды)
   Пояснения и примеры к п. 3. будут даны ниже.
4. Непосредственная. Непосредственная адресация означает, что в поле операндов приведено значение операнда (арифметическое, текстовое и т.п.).
5. Регистровая. С некоторой условностью можно выделить данный тип адресации. Казалось бы, что регистровая адресация является частным случаем символической, однако многие процесоры имеют так называемые "нерегулярные системы команд", в которых использование регистровой адресации может существенно влиять на код машинной команды. Этот факт и является основанием для выделения данного типа адресации.

В порядке пояснения пунктов 3.1., 3.2. и 4. приведем примеры использования соответствующих типов адресаций:

Поле метки	Поле КОП	Поле операндов
[Имя метки]	БПУ	MET99 + D`10`
	(безусловная передача управления по адресу, вычисленному как адрес метки MET99 плюс десять адресных единиц)	
[Имя метки]	БПУ	* - B`77`
	(безусловная передача управления по адресу, вычисленному как текущее значение счетчика адресов команд (*) минус 77 адресных единиц, причем, вычитаемое количество указано в восьмеричной системе счисления)	
[Имя метки]	ПЕР	Ш`ABC`, R2
	(пересылка значения шестнадцатеричной константы ABC в регистр общего назначения R2)	

Обратимся к вопросу о целесообразном количестве проходов ассемблирующего транслятора.
Обычно выделяют два типовых препятствия для однопроходной трансляции:
1) Ссылки на метки, котоые еще не определены (т.е. вниз по тексту).
2) Определение имен переменных и массивов в произвольном месте программы.
Хотя указанные препятствия преодолимы и однопроходные ассемблеры существуют, однако типовой и наиболее распространенной стала двухпроходная схема.
Именно двухпроходные ассемблеры и рассмотрим наиболее подробно.

Задачи первого и второго проходов в двух-проходном ассемблере

Задачи прохода I

I.1. Распределение памяти под все элементы программы (команды, переменные, массивы, непосредственные данные, описанные данные).

I.2. Выявление наиболее грубых ошибок, в частности, препятствующих второму проходу.

Задачи прохода II

II.1. Собственно трансляция команд с учетом сделанного на I-м проходе распределения памяти.

II.2. Формирование выходного объектного модуля и листинга трансляции.

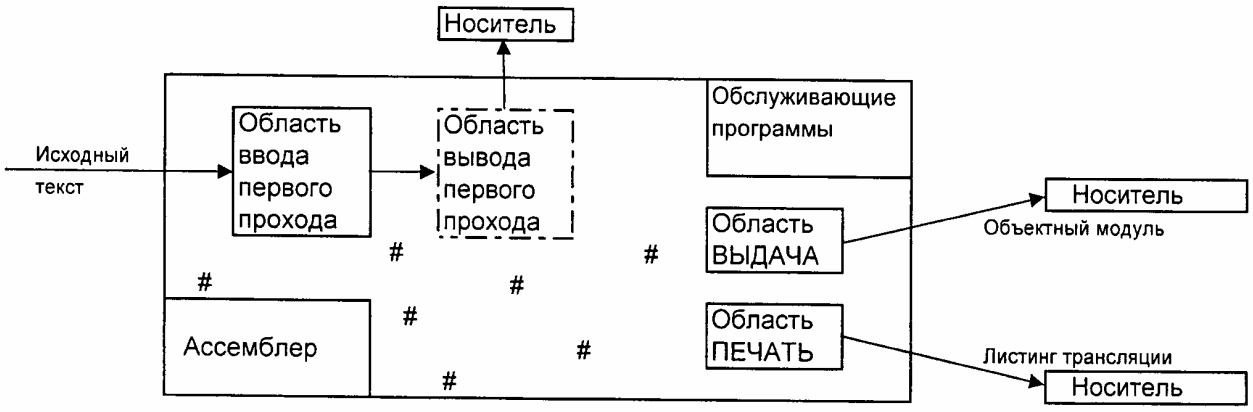
Введем две важные внутренние переменные ассемблирующего транслятора:

1) Общий Счетчик (Общ. Сч.) - подсчитывает количество адресных единиц памяти, зарезервированных в сумме под все элементы программы (команды, данные);
2) Счетчик Команд (Сч. К.) - подсчитывает количество адресных единиц памяти, зарезервированных только под команды.

Правила (проблемы) распределения памяти, осуществляемого ассемблером

1) Недопустимо, чтобы разным элементам программы была отведена (зарезервирована) одна и та же ячейка памяти.
2) Последовательные команды должны быть размещены в последовательных ячейках памяти.
3) Последовательные элементы массивов также должны быть размещены в последовательных ячейках памяти.
4) Всегда имеет место предельный объем памяти для команд программы и ее данных.

Распределение памяти ЭВМ, осуществляющей трансляцию.



- условное обозначение формирующих таблиц ассемблера

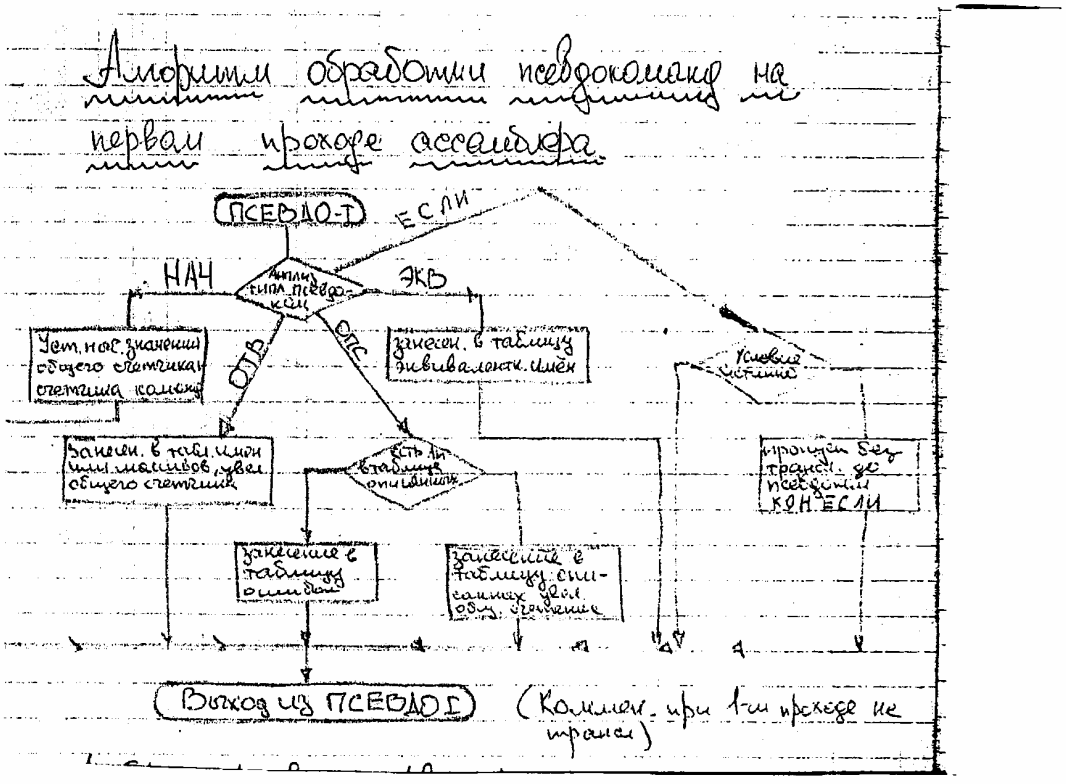
Формирующие таблицы ассемблера могут быть постоянными и динамическими.
Примеры постоянных таблиц: таблица команд; таблица диагностических сообщений; таблица зарезервированных имен и т.п.

Способы организации таблицы команд:

а) алфавитное упорядочение;
б) упорядочение в соответствии с относительными частотами использования команд;
в) упорядочение в соответствии с двоичными кодами мнемокодов команд в сочетании с поиском методом половинного деления;
г) индексирование (т.е. преобразование единственной таблицы в иерархическую совокупность подтаблиц).

Псевдокоманды (директивы) гипотетического ассемблера

Поле метки	Поле КОП	Поле операндов
	Установка ячейки начала прграммы	
Имя программы	**НАЧ**	**Константа**
ПРОГ1	НАЧ	B'1000'
	Отвести память	
ИМЯ	**ОТВ**	**Количество адресных единиц**
Вектор	ОТВ	D'100'
	Эквивалентирования	
ИМЯ	**ЭКВ**	**ИМЯ-эквивалент**
A	ЭКВ	B
C	ЭКВ	Вектор [99]
	Описания данных	
ИМЯ	**ОПС**	**Числовое или текстовое значение**
Пи	ОПС	3.14159
	Комментарий	
[ИМЯ]	**КОМ**	**Текст комментария**
	Конец программы	
[ИМЯ]	**КОН**	**[Константа или имя метки]**
	Псевдокоманда условной трансляции	
[ИМЯ]	**ЕСЛИ**	**Условие**
	. . . Фрагмент программы, подлежащий или не подлежащий трансляции . . .	
	КОНЕЦ ЕСЛИ	



Раздаточный материал. К вопросу об универсальном генераторе макрокоманд.

Пример возможных дополнительных структур данных (в данном случае таблиц), используемых в универсальном генераторе макрокоманд

Имя таблицы	Структура строки таблицы
1. МакроИмя (таблица-каталог)	1.1. Имя м/к. 1.2. Длина таблицы команд м/к. 1.3. Начальный адрес таблицы команд м/к. 1.4. Длина таблицы аргументов м/к.
2. Таблица команд (для каждой макрокоманды)	2.1. Тип команды 2.2. Мнемокод команды. 2.3. Машинный код. 2.4. Тип адресации. 2.5. Адресная ссылка. 2.6. Указатель на индекс-регистр (если он используется).
3. Таблица аргументов (может быть как для каждой м/к, так и общей)	3.1. Имя аргумента. 3.2. Адрес аргумента.

Раздаточный материал. К вопросу о трехпроходных ассемблерах.

Трехпроходные ассемблеры, тезисы

1) Классической считается двухпроходная схема работы ассемблирующего транслятора (ассемблера) с известным распределением функций между первым и вторым проходом (см. лекции).

2) Тем не менее, на практике можно встретить как однопроходные ассемблеры (см. лекции), так и трехпроходные.

3) Наиболее типичным способом построения трехпроходного ассемблера является такой, когда при условии сохранения функций первого и второго проходов организуется еще один проход, предшествующий первому. С некоторой условностью назовем этот новый проход «нулевым» проходом ассемблера.

4) При такой организации трехпроходного ассемблера обычно на «нулевой» проход возлагаются функции текстового препроцессора. То есть на вход «нулевого» прохода поступает текст программы, подлежащей трансляции, а на выходе образуется некий новый текст, являющийся преобразованной версией входного текста.

5) Смысл подобной организации трехпроходного ассемблера состоит в том, чтобы на «текстовом уровне» обработать все команды (операторы) входной программы, допускающие такую обработку и, тем самым, упростить работу первого и второго проходов.

Типичным примером, иллюстрирующим сказанное, является обработка макрокоманд текстовым препроцессором. Здесь имеется ввиду, что реакцией текстового препроцессора на поступление на его вход макрокоманды будет формирование так называемого макрорасширения (как и ранее создаваемого на основе макроопределения и списка фактических аргументов), но на текстовом уровне. Таким образом, макрокоманды «исчезают» из текста программы, а значит исчезает и необходимость обработки макрокоманд на тех проходах, которые мы называем первым и вторым. Алгоритмы этих проходов упрощаются за счет изъятия соответствующих ветвей (ранее эти ветви мы называли «макро», см. раздаточный материал к лекциям).

Кроме макрокоманд в конкретных реализациях ассемблеров могут встретиться и некоторые другие операторы, допускающие обработку текстовым препроцессором на том проходе ассемблера, который мы назвали «нулевым».

— абсолютная;
— символическая;
— относительная;
— непосредственная.

Абсолютная адресация означает употребление в адресной части команд абсолютных (физических) адресов операндов. В этом случае сам программист ответствен за распределение памяти, или, по крайней мере, ему должна быть досконально известна схема распределения памяти. Если в ассемблере реализована та или иная система символического именования данных, то использование соответствующих имен в адресной части команд соответствует символической адресации. Распределение памяти для символически поименованных объектов обычно осуществляется автоматически ассемблером. Говоря об относительной адресации, выделим две ее разновидности: а) адресация относительно символической метки; б) текущая относительная, т. е. адресация относительно исполняемой команды. Непосредственной адресацией будем для краткости называть использование в поле адреса литералов (непосредственных данных).

Из-за последовательного характера ввода и обработки транслируемой программы большинство ассемблеров делаются двухпроходными, т. е. они осуществляют свою работу за два просмотра (прохода) исходной информации. На первом проходе выполняется распределение памяти под имеющиеся в программе команды, переменные, массивы, непосредственные (буквальные) и описанные данные, а также выявляются правильность и согласованность входной программы, возможность дальнейшей трансляции.

На втором проходе осуществляются собственно трансляция мнемонических изображений команд и размещение данных, для которых в первом проходе было произведено распределение памяти, т. е. формируется выходная объектная программа. Кроме того, выполняется немаловажная операция по выводу листинга команд входного и машинного языков, таблиц распределения памяти, сообщений об ошибках и, иногда, о возможной оптимизации программы.

Требования к распределению памяти достаточно очевидны, но все же следует их перечислить: а) недопустимо, чтобы двум различным объектам программы была приписана одна и та же ячейка памяти; б) последовательные команды должны находиться в последовательных ячейках памяти; в) последовательные элементы массивов также должны быть расположены в последовательных ячейках; г) всегда существует предельный объем памяти на программу и ее подпрограммы.

§ 4. Ассемблеры

В нашем рассмотрении будем считать, что ассемблер должен обрабатывать следующие типы команд:
— мнемокоманды;
— псевдокоманды;
— макрокоманды.

При этом могут использоваться такие типы адресации, как:

Работа ассемблера строится на основе специальных так называемых формирующих таблиц. Вводимые поочередно операторы транслируемой программы анализируются ассемблером и дают информацию для пополнения одной или нескольких таблиц. В то же время уже накопленная в таблицах информация активно используется в процессе трансляции. Подробное рассмотрение возможных структур и способов употребления формирующих таблиц содержится, например, в [23], поэтому ограничимся лишь их перечислением в табл. 1.

Таблица 1

Наименование таблицы	Содержимое строки таблицы	Кто составляет
Таблица команд *	Мнемоническое изображение команды, соответствующий машинный код операции, другие характеристики команды	Системный программист — разработчик транслятора
Таблица имен	Имя, адрес, признаки: определено ли**, определено дважды, использовалось ли, есть ли имя-эквивалент	Ассемблер
Таблица массивов	Имя, начальный адрес, количество слов	Ассемблер, может быть совмещена с таблицей имен
Таблица описанных имен	Имя, адрес, двоичное значение	» То же
Таблица буквальных (непосредственных данных)	Исходное значение, адрес, двоичное значение	Ассемблер
Таблица эквивалентных имен	Имя, имя-эквивалент, признак определения	»
Таблица ошибок	Характеристика ошибки	»
Таблица распределения памяти	Таблица в целом содержит схему распределения памяти	

* Таблица является встроенной в ассемблер.

** Под определением имени будем понимать распределение памяти для него или, по крайней мере, резервирование памяти путем увеличения значения общего счетчика (счетчика размещения).

Укрупненная структурная схема алгоритма первого прохода ассемблера представлена на рис. 3. В процессе трансляции организуются и используются два счетчика: 1) общий счетчик или счетчик размещения, служащий для подсчета суммарного количества ячеек памяти, необходимых для программы и ее данных; 2) счетчик команд, подсчитывающий ячейки, зарезервированные для размещения только команд. Использование этих счетчиков как раз и позволяет провести корректное распределение памяти.

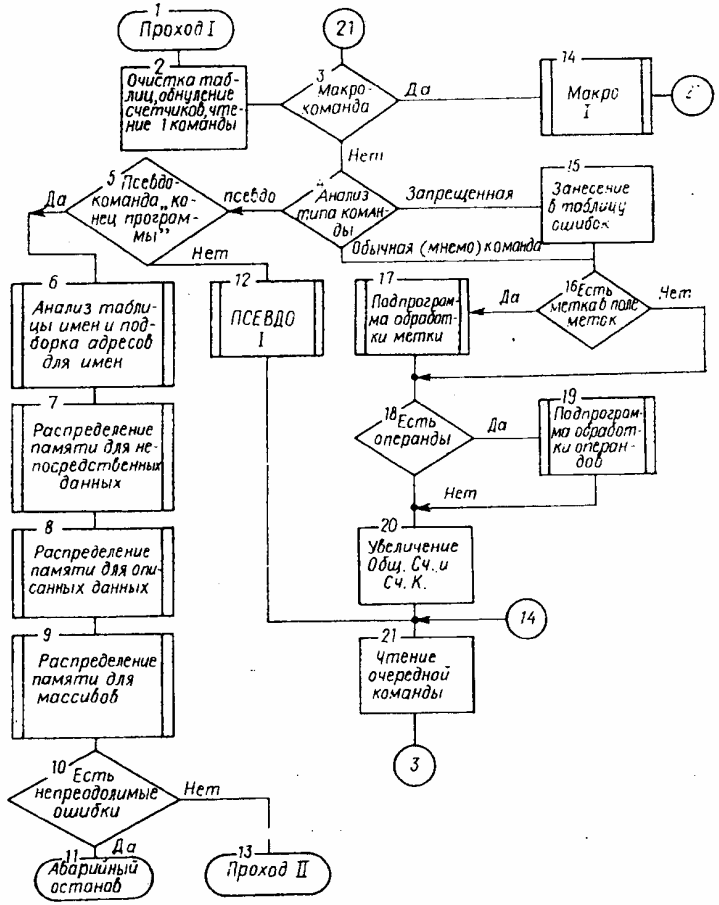


Рис. 3

Если общая схема первого прохода практически любого ассемблера соответствует рис. 3, то внутреннее содержание подпрограмм этого алгоритма существенно зависит от кон-

кретных свойств каждого машинно-ориентированного языка особенностей каждой ЭВМ, возможны и различные способы реализации подпрограмм.

Так, например, на рис. 4 приведен алгоритм обработки метки для случая, когда меткой может быть любое имя (ограниченное, конечно, лишь грамматикой языка), размещенное в общей таблице имен. Предполагается также, что текущее значение счетчика команд соответствует адресу, который будет отведен для первого слова команды, т. е. первому «свободному» на данный момент адресу. Еще один пример — алгоритм обработки операнда (рис. 5), приведенный для случая, когда допускается неявное объявление переменных путем употребления их имен в адресной части команды. Кроме того, предполагается, что буквальные операнды не будут входить в тело машинной команды, а обращения к ним будут происходить с помощью адресных ссылок. В других условиях подпрограмма обработки операндов может принять заметно другой вид. Так, наиболее существенно эта подпрограмма зависит от следующих факторов: допустимого количества операндов в команде, типов операндов и их возможных сочетаний, используемых способов объявления объектов программы, размещения непосредственных операндов, объема памяти, резервируемого под различные операнды.

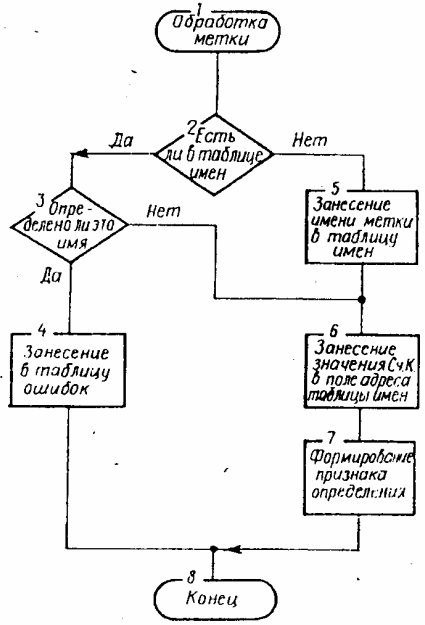


Рис. 4

В приведенных алгоритмах присутствуют блоки, соответствующие увеличению содержимого счетчиков. Приращения значений счетчиков тоже зависят от конкретной реализации. В частности, здесь имеют значение адресная структура памяти ЭВМ, структура машинной команды (одно-словная, двухсловная, трехсловная, всегда ли одна и та же), объем памяти, необходимый для каждого типа операндов.

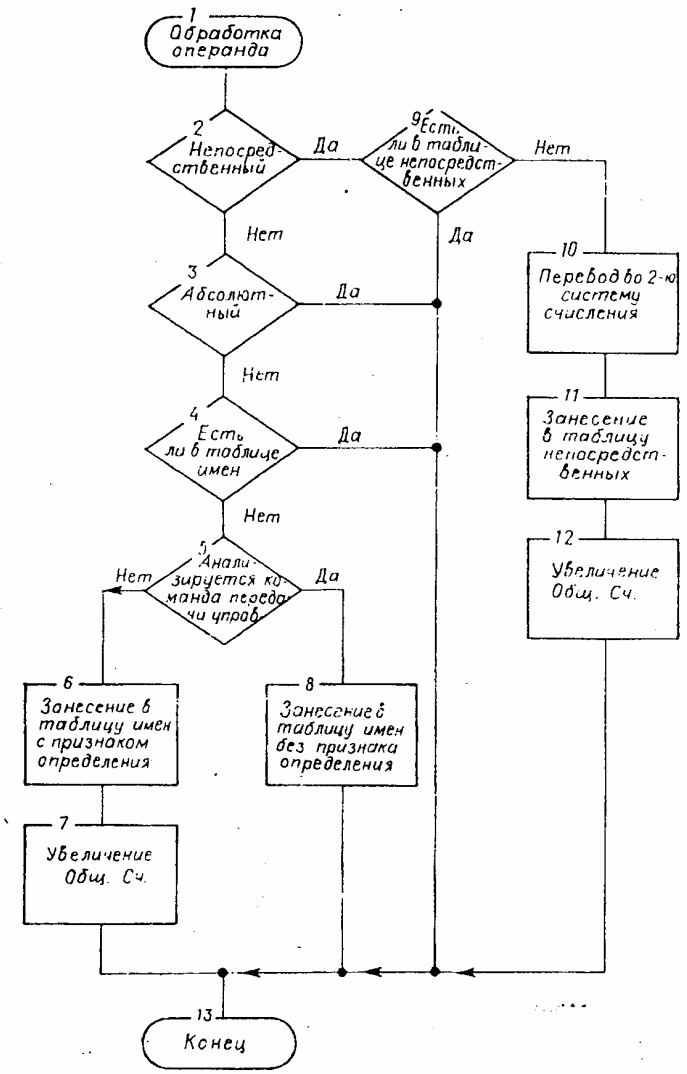


Рис. 5

Подпрограмма обработки псевдокоманд ПСЕВДО I главным образом выполняет операции по заполнению тех или иных формирующих таблиц в зависимости от типа псевдокоманд. Так, после чтения псевдокоманды объявления переменной, объявления массива, эквивалентирования, описания имени формируется строка таблицы соответственно имен, массивов, эквивалентности, описанных имен. Обработка подобных «служебных» псевдокоманд не порождает каких-либо исполняемых команд в тексте объектной программы.

Подпрограммы, соответствующие блокам *6—9* рис. 3, осуществляют распределение памяти для тех объектов транслируемой программы, память под которые была зарезерви-

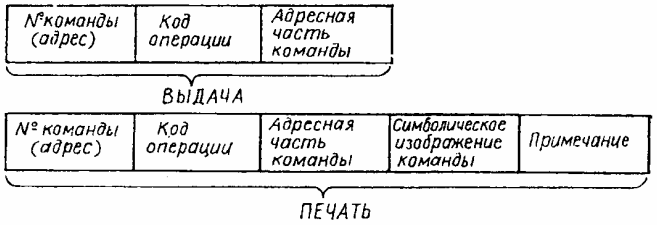


Рис. 6

рована увеличением значения общего счетчика, но конкретные значения адресов еще не были определены. Простейший способ распределения — размещение данных непосредственно вслед за последней командой программы, в этом случае первая свободная ячейка памяти всегда может быть определена по текущему значению счетчика команд. Эти же подпрограммы обычно выявляют неопределенные, неиспользованные, дважды определенные объекты и составляют соответствующие таблицы, поступающие затем в печать. Распределение памяти завершает первый проход, и при отсутствии непреодолимых ошибок может быть начат второй.

При рассмотрении второго прохода будем считать, что ассемблер формирует команды объектной программы в так называемой области ВЫДАЧА, а строки листинга — в области ПЕЧАТЬ. Структура этих областей показана на рис. 6, а укрупненный алгоритм второго прохода — на рис. 7. Принцип «ассемблирования», или иначе «сборки», здесь проявляется в том, что транслятор поочередно формирует поля указанных областей и, таким образом, «собирает» строки объектной программы и выходного листинга из отдельных частей.

Основной цикл второго прохода (см. рис. 7, блоки *4—19*) не требует особых пояснений. Подпрограмма обработки операндов (блок *11*) в зависимости от их типа формирует ад-

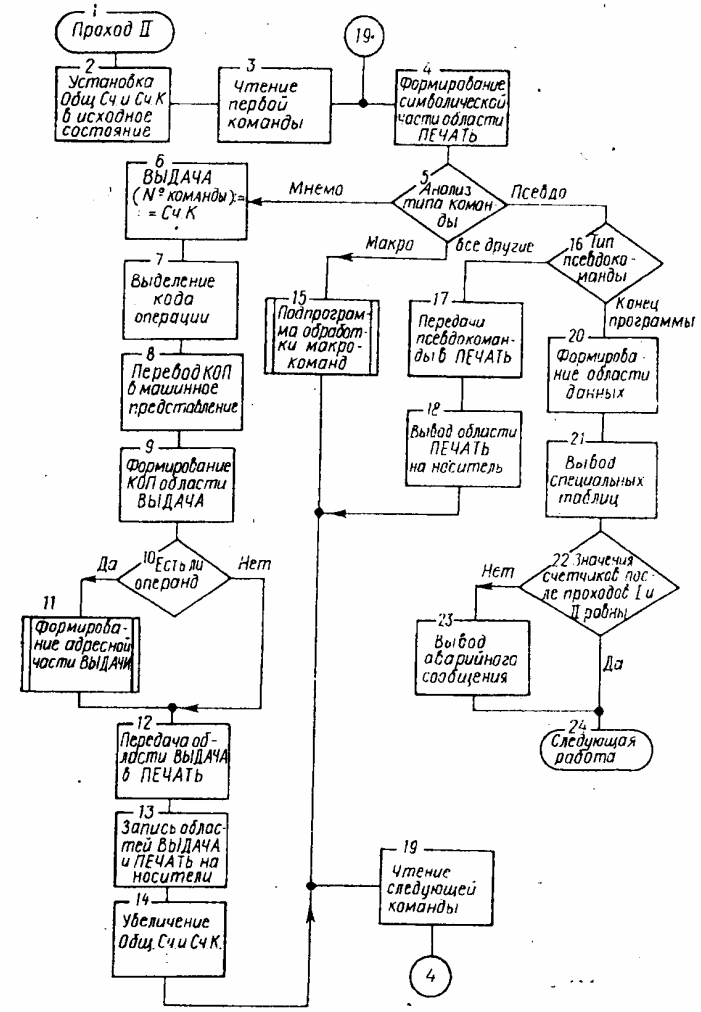


Рис. 7

ресную часть команды или непосредственно из исходных значений операндов, или с использованием таблиц имен, массивов, буквальных, описанных данных. Выход из основ-

ного цикла осуществляется после обнаружения в транслируемом тексте псевдокоманды «Конец программы». При завершении второго прохода выполняется важная функция по формированию области данных. Иначе говоря, здесь должна быть сформирована та часть объектного модуля, которая обеспечит загрузку значений буквальных и описанных данных в отведенные им ячейки памяти перед выполнением программы. В простейшем случае критерием благополучного завершения трансляции является равенство значений, накопленных после первого и второго проходов в общем счетчике и счетчике команд.

Кратко остановимся на обработке макрокоманд, описанных программистом. В зависимости от конкретной реализации ассемблера макрокоманды могут быть подключены к основной программе в различное время: во время сборки, между сборкой и загрузкой; во время загрузки; во время исполнения. Приведенные выше алгоритмы (см. рис. 3 и 7) соответствуют первому из перечисленных вариантов. Однако по способу использования макрокоманды можно разделить на закрытые и открытые. Текст закрытой макрокоманды всегда существует в единственном экземпляре, и используется она по принципу подстановки параметров, фактически аналогично подпрограмме. Дубликат открытой макрокоманды вставляется (генерируется) ассемблером в текст основной программы после обнаружения всякого нового обращения к макрокоманде. Каждый из двух способов имеет достоинства и недостатки: первый из них экономит память, второй — время исполнения программы. Говоря собственно о механизме использования макрокоманд, заметим, что подстановка параметров — довольно стандартная процедура, а вот генерация открытых макрокоманд требует специальных приемов и организации. В последнем случае обычно при трансляции организуются дополнительные формирующие таблицы: таблица имен макрокоманд, таблицы команд каждой макрокоманды и таблица аргументов [23], — которые позволяют выполнить операции по генерации текста макрокоманды на основе прототипа в пределах рассмотренного принципа сборки.

Ассемблирование за два прохода является наиболее универсальным и распространенным, но возможно построение и однопроходных ассемблеров. Основными препятствиями для трансляции за один просмотр обычно служат определение имен в произвольном месте программы и ссылки на метки, которые еще не определены. Первое затруднение может быть преодолено обязательным описанием всех имен в начале программы или ограничением списка возможных имен, а второе — например путем принудительного помечания каждого оператора. Таким образом, возможность трансляции за один проход может быть получена за счет некоторого ужесточения правил программирования. Добавим к этому, что использование памяти после однопроходной трансляции, как правило, будет неоптимальным. Примером языка с ужесточенными правилами может служить автокод «ап», где зафиксирован список имен и обязательно помечание каждого оператора (подобные ограничения имеют место и в языке Бейсик, в котором также возможна однопроходная трансляция).

Задачи для самостоятельного выполнения по курсу "Транслирующие системы"

Тема: автоматные грамматики и лексический анализ

Задача 1 (содержит три подзадачи). Постройте три конечных автомата лексического анализатора а), б) и в) с входным алфавитом {0,1}, которые допускают, соответственно, одну из ниже-перечисленных входных цепочек (или цепочки, обладающие указанными свойствами):

а) Каждый третий символ – единица.
б) Между вхождениями единиц четное число нулей.
в) Допустимая цепочка содержит четное количество нулей и нечетное количество единиц.

Задача 2. Постройте два конечных автомата лексического анализатора а) и б) с входным алфавитом {буквы, цифры}, которые допускают, соответственно, одну из ниже перечисленных входных цепочек:

а) Идентификаторы, состоящие из неограниченного количества литер.
б) Идентификаторы, состоящие из ограниченного количества литер (например, содержащие от одной до пяти литер).
Как и обычно, идентификатор должен обязательно начинаться с буквы.
Постройте регулярные грамматики этих языков.

Тема: контекстно-зависимые грамматики.

Задача 3. Для контекстно-зависимой грамматики **G**:

G = ({a}, {S, N, Q, R}, S, P) ,

где P:
a) <S> ::= <Q> <N> <Q>
b) <Q> <N> ::= <Q> <R>
c) <R> <N> ::= <N> <N> <R>
d) <R> <Q> ::= <N> <N> <Q>
e) <N> ::= a
f) <Q> ::= ε (ε – обозначение пустого множества) ,

показать, что цепочка **aaaa** выводится в соответствии с правилами грамматики.

Тема: грамматический разбор и неоднозначные грамматики.

Задача 4. Пусть дана грамматика операторов условного перехода:

a) <S> ::= if b then <S> else <S>
b) <S> ::= if b then <S>
c) <S> ::= a ,
где b – логическое выражение, принимающее значение «истина» или «ложь»;
a – лексема, представляющая любой оператор кроме оператора условного перехода.

Показать, что для допустимого входного предложения
if b_1 then if b_2 then a_1 else a_2
существуют, по крайней мере, два разных синтаксических дерева.
Показать также, что этим деревьям, в свою очередь, соответствуют существенно отличающиеся структуры программ.

Тема: грамматический разбор.

Задача 5. Грамматический анализатор "снизу-вверх" для следующей грамматики:

1). <S> ::= <S><S>a
2). <S> ::= <S>b
3). <S> ::= <S><S><S>c
4). <S> ::= d

при обработке некоторой входной цепочки распознает правила в следующем порядке:

4,4,4,1,4,4,1,3,2.

Определите:

а) Какова входная цепочка?

б) В каком порядке будут распознаваться правила при обработке этой цепочки распознавателем "сверху-вниз".

Тема: грамматики предшествования.

Задача 6. Определить отношения предшествования, на основе которых может быть разобран оператор условного перехода:

левый ограничитель if ... then ... else ... правый ограничитель

(Предполагается, что Вы построите таблицу отношений предшествования и приведете пример разбора на основе построенной таблицы).

Тема: транслирующие грамматики и атрибутные транслирующие грамматики.

Задача 7. Постройте две транслирующие грамматики а) и б), допускающие в качестве входа произвольную цепочку, состоящую из нулей и единиц, и порождающие в качестве выхода, соответственно:

а) обращение (инверсию) входной цепочки;

б) цепочку вида 0 0 ... 0 1 1 ... 1 ,
где "0" повторяется n раз, "1" повторяется m раз,
а n - число нулей, m - число единиц во входной цепочке.

Задача 8. Постройте транслирующую грамматику, допускающую в качестве входных цепочки, допускаемые конечным автоматом, таблица переходов которого показана на рисунке. В качестве выхода грамматика должна порождать последовательность состояний автомата, которые он проходит в процессе обработки входной цепочки. Например, входная цепочка 1010 должна переводиться в ABBCD.

Состояние	Входной символ	
	0	1
A	A	B
B	B	C
C	D	A
D	A	D

Рис. Таблица переходов распознающего автомата.

A - считается начальным состоянием (Старт).

Цепочки нулей и единиц считаются допустимыми, если их анализ заканчивается состоянием B или D автомата.

Задача 9. Определите атрибуты для следующей грамматики:

1). <целое> ::= цифра<цифры>
2). <цифры> ::= цифра<цифры>
3). <цифры> ::= ε (ε – *обозначение пустого множества*)

так, чтобы нетерминал <целое> имел один синтезируемый атрибут, равный значению порождаемого им числа. Считайте, что входной символ "цифра" имеет один атрибут - число между 0 и 9.

Заключительная задача.

Задача 10. В обычной алгебре переменные обозначаются одной буквой, и поэтому знак операции умножения часто опускается (так, 2z означает удвоение z). Найдите контекстно-свободную грамматику для многочленов, составленных из целых чисел, однобуквенных переменных и операций +, -, ! (возведение в степень), причем, операции умножения подразумеваются между соседними переменными, а также между стоящими рядом переменной и целым числом.