1. ОБЩИЕ СВЕДЕНИЯ О ЯЗЫКЕ, ПЕРЕВОДЕ И СТРУКТУРЕ ТРАНСЛЯТОРА

Массовое производство ЭВМ, быстро возрастающая роль малых и особенно персональных ЭВМ привели к необходимости найти новые методы и средства программирования прикладных и системных задач. В этом вопросе достигнут определений прогресс, однако ни новые инструментальные системы, ни новейшие технологии и новейшие стили программирования (логический, концептуальный и другие), не нарушили основополагающего значения алгоритмических языков в разработке как базового программного обеспечения ЭВМ, так и различных программных средств для непрограммистов.

В последнее время все большую популярность завоевывают алгоритмические языки, появившиеся в 80-х годах, такие как Паскаль, Си, Ада и др. Успех в распространении этих языков связан не только с включением в языки новых средств описания и обработки сложных структур данных, но и с разработкой надежных и эффективных трансляторов с этих языков. Последнее обязано в первую очередь практическим и научным результатам, полученным в теории синтаксически-управляемого перевода и методах трансляции.

1.1. Схема работы транслятора

Каждая вычислительная машина имеет свой собственный язык программирования — язык команд или машинный язык — и может исполнять программы, записанные только на этом языке. С помощью машинного языка, в принципе, можно описать любой алгоритм, но затраты на программирование будут чрезвычайно велики. Это обусловлено тем, что машинный язык позволяет описывать и обрабатывать лишь примитивные структуры данных — бит, байт, слово. Программирование в машинных кодах требует чрезмерной детализации программы и доступно лишь программистам, хорошо знающим устройство и функциониро-

вание ЭВМ. Преодолеть эту трудность и позволили языки высокого уровня (Фортран, ПЛ/1, Паскаль, Си, Ада, и др.) с развитыми структурами данных и средствами их обработки, не зависящими от языка конкретной ЭВМ.

Алгоритмические языки высокого уровня обладают достаточно высокой изобразительной силой, они дают возможность программисту достаточно просто и удобно описывать алгоритмы решения многих прикладных задач. Такое описание называют *исходной программой*, а язык высокого уровня — *входным языком*.

Языковым процессором называют программу на машинном языке, позволяющую вычислительной машине понимать и выполнять программы на входном языке. Различают два основных типа языковых процессоров: интерпретаторы и трансляторы [2].

Интерпретатор — это программа, которая в качестве входа допускает программу на входном языке и по мере распознавания конструкций входного языка реализует их, выдавая на выходе результаты вычислений, предписанные исходной программой.

Транслятор — это программа, которая допускает на входе исходную программу и порождает на своем выходе программу, функционально-эквивалентную исходной, называемую *объектной*. Объектная программа записывается на объектном языке. В частном случае, объектным языком может служить машинный язык, и в этом случае полученную на выходе транслятора программу можно сразу же выполнить на ЭВМ (проинтерпретировать). При этом ЭВМ является интерпретатором объектной программы в машинных кодах. В общем случае объектный язык необязательно должен быть машинным или близким к нему (автокодом). В качестве объектного языка может служить некоторый *промежуточный язык* — язык, лежащий между входным и машинным языками.

Если в качестве объектного языка используется промежуточный язык, то возможны два варианта построения транслятора.

Первый вариант — для промежуточного языка имеется (или разрабатывается) другой транслятор — с промежуточного языка на машинный, и он используется в качестве последнего блока проектируемого транслятора.

Второй вариант построения транслятора с использованием промежуточного языка — построить интерпретатор команд промежуточного языка и использовать его в качестве последнего блока транслятора. Преимущество интерпретаторов проявляется в отладочных и диалоговых трансляторах, обеспечивающих работу пользователя в диалоговом режиме, вплоть до внесений изменений в программу без ее повторной полной перетрансляции.

Интерпретаторы используются также и при эмуляции программ — исполнении на технологической машине программ, составленных для другой (объектной) машины. Данный вариант, в частности, используется при отладке на универсальной ЭВМ программ, которые будут выполняться на специализированной ЭВМ.

Транслятор, использующий в качестве входного языка язык, близкий к машинному (автокод или ассемблер), традиционно называют *ассемблером*. Транслятор для языка высокого уровня называют *компилятором*.

В построении компиляторов за последние годы достигнуты значительные успехи. Первые компиляторы использовали так называемые *прямые методы трансляции* — это преимущественно эвристические методы, в которых на основе общей идеи для каждой конструкции языка разрабатывался свой алгоритм перевода в машинный эквивалент [2]. Эти методы были медленные и не носили структурного характера. В основе методики проектирования современных компиляторов лежит композиционный *синтаксически-управляемый метод* обработки языков. Композиционный в том смысле, что процесс перевода исходной программы в объектную реализуется композицией функционально независимых отображений с явно выделенными входными и выходными структурами данных. Отображения эти строятся из рассмотрения исходной программы, как композиции основных аспектов (уровней) описания входного языка: лексики, синтаксиса, семантики и прагматики, и выявления этих аспектов из исходной программы в ходе ее компиляции. Рассмотрим эти аспекты с целью получения упрощенной модели компилятора.

Основой любого естественного или искусственного языка является *алфавит* — набор допустимых в языке элементарных знаков (букв, цифр и служебных знаков). Знаки могут объединяться в *слова* — элементарные конструкции языка, рассматриваемые в тексте (программе) как неделимые символы, имеющие определенный смысл. Словом может быть и одиночный символ. Например, в языке Паскаль словами являются идентификаторы, ключевые слова, константы и разделители, в частности знаки арифметических и логических операций, скобки, запятые и другие символы. Словарный состав языка вместе с описанием способов их представления составляют *лексику* языка.

Слова в языке объединяются в более сложные конструкции — предложения. В языках программирования простейшим предложением является оператор. Предложения строятся из слов и более простых предложений по правилам синтаксиса. *Синтаксис* языка представляет собой описание правильных предложений. Описание смысла предложений, т.е. значений слов и их внутренних связей, составляет *семантику* языка. В дополнение отметим, что конкретная программа не-

сет в себе некоторое воздействие на транслятор — прагматизм. В совокупности синтаксис, семантика и *прагматизм* языка образуют *семиотику* языка.

Перевод программы с одного языка на другой, в общем случае, состоит в изменении алфавита, лексики и синтаксиса языка программы с сохранением ее семантики. Процесс трансляции исходной программы в объектную обычно разбивается на несколько независимых подпроцессов (фаз трансляции), которые реализуются соответствующими блоками транслятора. Удобно считать основными фазами трансляции лексический анализ, синтаксический анализ, семантический анализ и синтез объектной программы. Тем не менее, во многих реальных компиляторах эти фазы разбиваются на несколько подфаз, могут также быть и другие фазы (например, оптимизация объектного кода) [5]. На рис. 1.1. показана упрощенная функциональная модель транслятора.

В соответствии с этой моделью входная программа прежде всего подвергается лексической обработке. Цель лексического анализа — перевод исходной программы на внутренний язык компилятора, в котором ключевые слова, идентификаторы, метки и константы приведены к одному формату и заменены условными кодами: числовыми или символьными, которые называются дескрипторами. Каждый дескриптор состоит из двух частей: класса (типа) лексемы и указателя на адрес в памяти, где хранится информация о конкретной лексеме. Обычно эта информация организуется в виде таблиц. Одновременно с переводом исходной программы на внутренний язык на этапе лексического анализа проводится *лексический контроль* — выявление в программе недопустимых слов.

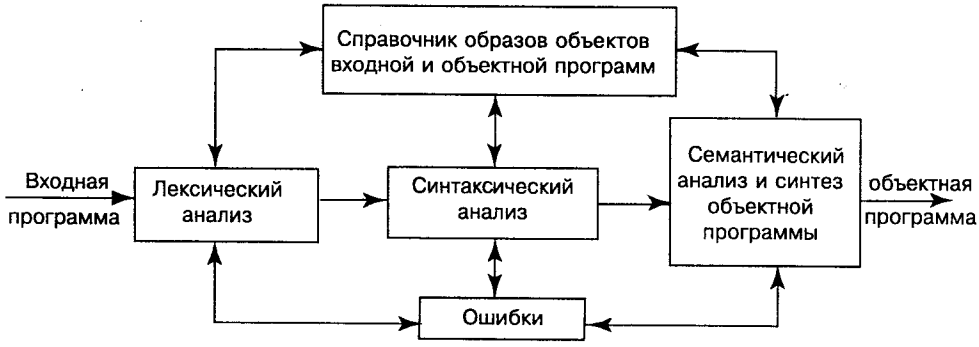

Рис. 1.1. Упрощенная функциональная модель транслятора

Синтаксический анализатор воспринимает выход лексического анализатора и переводит последовательность образов лексем в форму промежуточной программы. Промежуточная программа является, по существу, представлением синтаксического дерева программы. Последнее отражает структуру исходной программы, т.е. порядок и связи между ее операторами. В ходе построения синтаксического дерева выполняется *синтаксический контроль* — выявление синтаксических ошибок в программе.

Фактическим выходом синтаксического анализатора может быть последовательность команд, необходимых для того, чтобы строить промежуточную программу, обращаться к таблицам справочника, выдавать, когда это требуется, диагностические сообщения [5].

Синтез объектной программы начинается, как правило, с распределения и выделения памяти для основных программных объектов. Затем производится исследование каждого предложения исходной программы и генерируются семантически эквивалентные предложения объектного языка. В качестве входной информации здесь используется синтаксическое дерево программы и выходные таблицы лексического анализатора — таблица идентификаторов, таблица констант и другие. Анализ дерева позволяет выявить последовательность генерируемых команд объектной программы, а по таблице идентификаторов определяются типы команд, которые допустимы для значений операндов в генерируемых командах (например, какие требуется породить команды: с фиксированной или плавающей точкой и т.д.).

Непосредственно генерации объектной программы часто предшествует *семантический анализ*, который включает различные виды семантической обработки. Один из видов — проверка семантических соглашений в программе. Примеры таких соглашений: единственность описания каждого идентификатора в программе, определение переменной производится до ее использования и т.д. Семантический анализ может выполняться и на более поздних фазах трансляции, например, на фазе оптимизации программы, которая тоже может включаться в транслятор. Цель оптимизации — сокращение временных ресурсов или ресурсов оперативной памяти, требуемых для выполнения объектной программы.

Таковы основные аспекты процесса трансляции с языков высокого уровня. Подробнее организация различных фаз трансляции и связанные с ними практические способы их математического описания рассматриваются ниже.

1.2. Описание входного языка транслятора

Первое, что отличает один язык программирования от другого — это их синтаксис. Основное назначение синтаксиса — предоставить систему обозначений для обмена информацией между программистом и транслятором. Однако, при разработке деталей синтаксиса чаще исходят из второстепенных критериев, назначение которых: сделать программу удобной для чтения, написания и трансляции, а также сделать ее однозначной. Если удобство чтения и записи программ необходимы для пользователя языка программирования, то простота трансляции и отсутствие разночтений в языке имеют отношение к нуждам транслятора. Эти цели, в общем случае, противоречивы, и нахождение приемлемого компромисса при их решении является одной из центральных задач при разработке языка программирования.

Разработка нового языка программирования начинается с определения его синтаксиса. Для описания синтаксиса языка программирования, в свою очередь, нужен также некоторый класс. Язык, предназначенный для описания другого языка, называют *метаязыком*. Язык, используемый для описания синтаксиса языка, называют *метасинтаксическим* языком. В метасинтаксических языках используется специальная совокупность условных знаков, которая образует нотацию этого языка.

Исторически первым метасинтаксическим языком, который использовался на практике для описания синтаксиса языков программирования (в частности Алгола-60), являются *нормальные формы Бэкуса*, сокращенно обозначают БНФ — бэкусова нормальная форма или бэкусо-науровская форма. Основное назначение форм Бэкуса состоит в представлении в сжатом и компактном виде строго формальных и однозначных правил написания основных конструкций описываемого языка программирования.

Формальное определение синтаксиса языка программирования обычно называется *грамматикой*.

В форме Бэкуса описываются два класса объектов: это, во-первых, основные символы языка программирования и, во-вторых, имена конструкций описываемого языка, или так называемые, *металингвистические переменные*.

Каждая металингвистическая формула (форма) описывает правила построения конструкций языка и состоит из двух частей. В левой находится металингвистическая переменная, обозначающая соответствующую конструкцию. Далее следует *металингвистическая связка* ::=, означающая «определяется как» или «есть». В правой части формулы указывается один или несколько вариантов построения конструкции, определяемой в левой части. Для пост-

роения определяемой формулой конструкции нужно выбрать некоторый вариант ее построения из правой части формулы и подставить вместо каждой металингвистической переменной некоторые цепочки основных символов. Варианты правой части формулы разделяются металингвистичской связкой |, имеющей смысл «или».

Сами металингвистические переменные обозначаются словами, поясняющими смысл описываемой конструкции, и заключаются в угловые скобки < >.

В качестве примера БНФ приведем определение десятичного целого числа:

1. < десятичное целое число > ::= < число без знака > |
   + < число без знака > | — < число без знака >
2. < число без знака > ::= < цифра > | < число без знака > < цифра >
3. <цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.

Например, число +294 выводится по формулам следующим образом:

< десятичное целое число >,
+ < число без знака >,
+ < число без знака > < цифра >,
+ < число без знака > 4,
+ < число без знака > < цифра > 4,
+ < число без знака > 94,
+ < цифра > 94,
+ 294.

Особенностью многих металингвистических формул является наличие в них *рекурсий*, т.е. использование для описания конструкций самих описываемых конструкций. Рекурсия может быть явной и неявной. *Явная рекурсия* имеет место, например, в правиле 2 в приведенном выше списке правил описания десятичного числа. *Неявная рекурсия* присутствует в случае, когда при построении конструкции на некотором шаге используется металингвистическая переменная, обозначающая саму эту конструкцию.

Наличие рекурсий затрудняет чтение и понимание металингвистических формул, однако это едва ли не единственный способ, позволяющий с помощью конечного числа правил, описать язык, который может содержать бесконечное число цепочек основных символов. Языки же программирования бесконечны — на них можно записать бесконечное число правильных программ, и при описании их синтаксиса с помощью БНФ всегда будут присутствовать явные или неявные рекурсии.

На практике для описания синтаксиса языков программирования применяются и другие металингвистические языки. Одна из целей их использования — устранить некоторую неестественность представления в БНФ общих синтаксических конструкций для необязательных, альтернативных и повторяющихся элементов правил. Так, для описания синтаксиса таких языков, как КОБОЛ и ПЛ/1, использовалась следующая нотация, являющаяся расширением БНФ:

- ❑ Необязательный элемент внутри правила заключается в квадратные скобки [. . .].
- ❑ Альтернативные элементы обозначаются вертикальным списком вариантов, заключенным в фигурные скобки {. . .}.
- ❑ Необязательные альтернативные варианты обозначаются вертикальным списком вариантов, заключенным в квадратные скобки [. . .].
- ❑ Повторяющийся элемент обозначается списком из одного элемента (заключенного, если это необходим, в фигурные или квадратные скобки) со следующим за ним обычным многоточием ....
- ❑ Обязательные ключевые слова подчеркиваются, а необязательные шумовые слова — нет.

Приводимое ранее описание БНФ десятичного числа в данной нотации будет иметь вид:

<десятичное целое > ::= [+ / −] <цифра > ...

<цифра> ::= { 0 / 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9 }

Такое описание более компактно и естественно. Однако оба описания эквивалентны, т.е. любое правило, записанное на этом языке, может быть однозначно представлено в виде одной или нескольких форм Бэкуса, и наоборот.

Формы Бэкуса представляют более формальное описание языка и они, по существу, натолкнули исследователей на внедрение математических средств для системного описания и исследования языков программирования, использование

математического аппарата как основы для синтаксического анализа в трансляторах, что позднее получило развитие в разнообразных методах синтаксического анализа, основанных на формальных синтаксических определениях.

Необходимо отметить, что БНФ не позволяет описывать контекстные зависимости в языке программирования. Например, такое ограничение Паскаль программ, как «идентификатор не может быть описан дважды в одном и том же блоке», нельзя описать средствами БНФ. Ограничения такого рода ближе уже к другой характеристике языка — семантике. Поэтому здесь используются другие средства, в общем случае называемые *метасемантическими языками*. Однако, как правило, ядром этих языков является та же БНФ.

1.3. Формальное определение языков программирования

Систематическое использование математических методов для описания языков программирования восходит к 1960 году. Тогда было обнаружено, что формы Бэкуса, которые использовались для описания синтаксиса языка АЛГОЛ-60, имеют строгое формальное обоснование с помощью средств математической лингвистики. С этого времени и началась история развития и применения формального математического аппарата — теории формальных языков и грамматик — для проектирования и конструирования трансляторов.

1.3.1. ФОРМАЛЬНЫЕ ЯЗЫКИ И ГРАММАТИКИ

Первоначально наука о языках — *лингвистика* — сводилась к изучению конкретных естественных языков, их классификации, выяснению сходств и различий между ними. Возникновение и развитие метаматематики, изучающей язык математики, проведение работ по изучению средств коммуникации животных и другие исследования привели в 30-х годах к существенно более широкому представлению о языке, при котором под языком понимается всякое средство общения, состоящее из [1]:

- ❑ знаковой системы, т.е. множества допустимых последовательностей знаков;
- ❑ множества смыслов этой системы;
- ❑ соответствия между последовательностями знаков и смыслами, делающего «осмысленными» допустимые последовательности знаков.

Знаками могут быть буквы алфавита, математические обозначения, звуки и т.д. Математическая лингвистика рассматривает только такие знаковые системы, в которых знаками являются символы некоторого алфавита, а последовательностями знаков — тексты, т.е. языки рассматриваются как произвольные последовательности осмысленных текстов. При этом правила, определяющие множество текстов, образуют *синтаксис языка*, а описание множества смыслов и соответствия между смыслами и текстами — *семантику языка*. Семантика языка зависит от характера объектов, описываемых языком, и средства ее изучения различны для различных типов языков. Синтаксис же языка, как оказалось, в меньшей степени зависит от назначения языка и может изучаться методами, не зависящими от содержания и назначения языка. Математический аппарат для изучения синтаксиса языков получил название *теория формальных грамматик*. С точки зрения синтаксиса, язык здесь понимается уже не как средство общения, а как множество формальных объектов — последовательностей символов алфавита. Термин «формальный» подчеркивает, что объекты и операции над ними рассматриваются чисто формально, без каких-либо содержательных интерпретаций объектов. Воспроизведем основные термины и определения этой теории.

Буква (или *символ*) — это простой неделимый знак; множество букв образует алфавит. Алфавиты являются множествами, и поэтому к ним можно применять теоретико-множественные обозначения. В частности, если A и B такие алфавиты, что $A \subseteq B$, то будем говорить, что A является подалфавитом B.

Цепочка — упорядоченная последовательность букв алфавита. Пусть A — алфавит, тогда цепочки одинаковой длины являются элементами множества: $A_n = A \times A \times ... \times A$ и записываются в виде: $a_1 a_2 ... a_n$, а не $(a_1, a_2, ..., a_n)$, как это принято для обозначения элементов декартового произведения множеств. Буквы также являются цепочками для случая n = 1. Цепочка может и не иметь букв, тогда это — *пустая цепочка*, будем обозначать эту цепочку символом λ. При этом λ не является буквой, т.е. $\lambda \notin A$.

Цепочки будем называть также *словами*. Множество всех возможных цепочек (слов) над алфавитом A называют *замыканием* A и обозначают A*, так что

$$A^* = A^0 \cup A^1 \cup ... = \bigcup_{n=0}^{\infty} A^n \text{ , здесь } A^0 = \{\lambda\},\ A^1 = A.$$

Множество A* называют *итерацией* алфавита A. Множество непустых цепочек (слов) над алфавитом A (*усеченная итерация* алфавита A) определятся как:

$$A^+ = A^* \setminus \{\lambda\} = \bigcup_{n=1}^{\infty} A^n$$

Каждая цепочка $\alpha \in A^*$ имеет конечную длину, которая обозначается через $|\alpha|$ и равна числу букв α, при этом $|\lambda| = 0$.

Цепочки могут образовывать последовательности цепочек. Для этих целей используется бинарная операция над цепочками, которая называется *конкатенацией*. Операция конкатенации обозначается знаком Θ, определяется на множестве A^* следующим образом: если $\alpha, \beta \in A^*$, то $\alpha \Theta \beta \in \alpha\beta$, т.е. результатом выполнения операции является цепочка α и сразу же за ней записанная цепочка β. Операция Θ ассоциативна, но не коммуникативна. При этом $\lambda\Theta\alpha = \alpha\Theta\lambda = \alpha$ для любых цепочек α и $|\alpha\beta| = |\alpha| + |\beta|$.

Если цепочки состоят из повторяющихся букв, то применяются сокращенные обозначения, чтобы показать, что цепочку нужно рассматривать как произведение букв алфавита. Поэтому, например, с помощью буквы $x \in A$ можно образовать цепочки $\lambda = x^0$, $xx = x^2$, $xx^{n-1} = x^n$ и т.д. Это же используется для обозначения повторяющихся цепочек: например цепочку *xyxyx* можно записать как $x(yx)^2$ или $(xy)^2x$, при этом символы $(,) \notin A$.

При преобразовании одних цепочек в другие используется понятие *подцепочки*. Пусть $\alpha, \beta \in A$, A — алфавит. Цепочка β называется *подцепочкой* α, если $\alpha = \gamma\beta\delta$; $\gamma, \delta \in A^*$.

Альтернативным набором терминов для буквы, алфавита или цепочки (слова) является набор: *слово*, *словарь* и *предложение* соответственно. Совокупность цепочек (или предложений) называется *языком*. Формально *язык* L над алфавитом A — это множество цепочек A^*, поэтому $L \subseteq A^*$. Следовательно операции над цепочками индуцируют операции на языках. Отсюда получаем:

$$L^0 = \{\lambda\}; L^n = L^{n-1}L, n \in N, L^+ = \bigcup_{n=1}^{\infty} L^n, \; L^* = \bigcup_{n=0}^{\infty} L^n$$

Таким образом, используя приведенную выше терминологию, язык программирования для заданного алфавита A является таким подмножеством множества A^*, которое включает в себя только те предложения, которые благодаря внешней информации об их семантике считаются осмысленными, т.е. удовлетворяют синтаксису языка программирования.

Приведенное определение формального языка как любого подмножества A^* является чересчур общим: оно не позволяет выделять среди множества языков отдельные их классы, которые используются на практике. Выделять такие классы можно, используя *соотношения Туэ* (названными так в честь норвежского математика, впервые их использовавшего).

Соотношениями Туэ называют правила, согласно которым любой цепочке $\alpha = \gamma\xi_i\delta$ из A^* ставится в соответствие цепочка $\beta = \gamma\eta_i\delta$ из того же множества

A*(i = 1, 2, ..., n) и наоборот. Соотношения Туэ приводят к так называемым *ассоциативным исчислениям* [7]. Однако и соотношения Туэ слишком общи, чтобы быть основой исследования таких свойств, как синтаксис языков программирования. Наложение ограничений на правила туэтовских систем — введение односторонних правил, в этом случае их называют *полусоотношениями* или *продукциями* (обозначают их ($\xi_i \to \eta_i$), привело к созданию формального математического аппарата — *формальным грамматикам*, которые являются, по-существу, полусистемами Туэ, и которые оказались наиболее приемлемым механизмом описания языков программирования.

Теория формальных грамматик занимается описанием, распознаванием и переработкой языков. Она позволяет ответить на ряд прикладных вопросов. Например, могут ли языки из некоторого класса Z распознаваться быстро и просто; принадлежит ли данный язык классу Z; существуют ли алгоритмы, которые давали бы ответ на вопросы типа: «Принадлежит или нет к языку L цепочка α?» и т.д.

В общем случае существуют два основных способа описания отдельных классов языков:

- с помощью порождающей процедуры;
- с помощью распознающей процедуры.

Первая из них задается с помощью конечного множества правил, называемых *грамматикой* и порождающих в точности те цепочки, которые принадлежат языку L. Вторая — с помощью некоторого абстрактного распознающего устройства (автомата). При построении трансляторов используются оба эти способа: грамматика как средство описания *синтаксиса языка программирования, а автомат как модель алгоритма распознавания предложений языка*, который и кладется в основу построения транслятора. При этом методически (и технологически) сначала конструируется грамматика, а затем уже по ней, как по источнику, строится алгоритм распознавания.

Перейдем теперь к формальному изложению рассмотренных выше понятий.

1.3.2. ПОРОЖДАЮЩАЯ ГРАММАТИКА

Определение 1.3.2.1. *Формальной порождающей грамматикой* называется четверка G = <N, T, P, S>, где

T — конечное непустое множество символов, называемое терминальным (основным) словарем грамматики G; элементы множества T называют терминальными символами (*терминалами*);

N — конечное непустое множество символов, называемое нетерминальным (вспомогательным) словарем грамматики G, $T \cap N = \varnothing$, $V = T \cup N$ — объединенный словарь грамматики G; элементы множества N называют нетерминальными символами (или *нетерминалами*);

S — начальный символ (*аксиома*) грамматики G; $S \in N$ и обозначает главный нетерминал (цель) грамматики G;

P — конечное множество правил грамматики, т.е. цепочек вида $\varphi \to \psi$ и называемых также *правилами подстановки* или *продукциями*, при этом φ, ψ — цепочки в словаре $V=T \cup N$ и $\varphi \in (T \cup N)^*N(T \cup N)^*$, $\psi \in (N \cup T)^*$. Конечное двуместное отношение $\to$ интерпретируется как «заменить» φ на ψ «или подставить ψ вместо φ».

Множество правил подстановки P называют также *схемой грамматики*. Цепочка, стоящая в левой части правила грамматики, обязательно содержит хотя бы один нетерминальный символ. В правой же части правила в общем случае может стоять произвольная цепочка из терминальных и нетерминальных символов, включая и пустую цепочку λ.

В дальнейшем элементы из нетерминального словаря N будем обозначать прописными латинскими буквами A, B, C, ..., элементы из T (терминальные символы) — строчными латинскими буквами a, b, c, ..., произвольные цепочки — греческими буквами α, β, γ...

Будем говорить, что цепочка ω' непосредственно выводима из цепочки ω в грамматике G ($\omega \Rightarrow \omega'$), если $\omega = \xi_1\varphi\xi_2$, $\omega' = \xi_1\psi\xi_2$ и в множестве правил подстановки P найдется правило $\varphi \to \psi$.

Будем говорить, что цепочка ω' выводима из цепочки ω в грамматике G ($\omega \overset{*}{\Rightarrow} \omega'$), если найдется последовательность цепочек $\omega = \omega_0, \omega_1, ..., \omega_n = \omega'$ такая, что цепочка ω_{i+1} непосредственно выводима в грамматике G из цепочки ω_i, т.е. ($\omega_i \Rightarrow \omega_{i+1}$) при i = 0, 1, ... n–1, либо $\omega' = \omega$. Всюду ω' — произвольная цепочка, т.е. $\omega_i \in (N \cup T)^*$. Отношение $\overset{*}{\Rightarrow}$ называется *транзитивным замыканием*, а последовательность цепочек $\omega_0, \omega_1, ..., \omega_n$ — *выводом цепочки* ω_n из цепочки ω_0 в грамматике G.

Определение 1.3.2.2. Множество всех цепочек терминальных символов, выводимых из аксиомы грамматики, называется *языком, порождаемым этой грамматикой*, т.е. $L(G) = \{x \mid S \overset{*}{\Rightarrow} x, x \in T^*\}$.

Пример 1.3.2.1. [1].
Рассмотрим грамматику G = <N, T, P, S> у которой:
N = {I},
T = {a, b, c, V, &,¬, (,)},
S = {I},

$$
\begin{aligned}
P = \{\ & I \to (\,I \vee I\,) \\
& I \to (\,I \,\&\, I\,) \\
& I \to \neg I \\
& I \to a \\
& I \to b \\
& I \to c\ \}
\end{aligned}
$$

Эта грамматика описывает язык булевых формул с переменными a, b, c и логическими функциями V, &, ¬. Примером вывода в этой грамматике является вывод:

$$I \Rightarrow (IVI) \Rightarrow ((I\&I)VI) \Rightarrow (a \,\&\, I)VI) \Rightarrow ((a\&b)VI) \Rightarrow ((a \,\&\, b)Vc).$$

Пример 1.3.2.2. Язык $a^n b^n a^n$ порождается следующей грамматикой G = <N, T, P, S>, у которой: N = {I, A, B, C, D}, T = {a, b, c}, S = {I},

$$
\begin{aligned}
P = \{\ & I \to ABA \\
& B \to ABCA \\
& B \to b \\
& bC \to bb \\
& AC \to DC \\
& DC \to DA \\
& DA \to CA \\
& A \to a\ \}
\end{aligned}
$$

Примером вывода в данной грамматике цепочки $a^2b^2a^2$ является вывод:

$$
\begin{aligned}
I \Rightarrow ABA \Rightarrow AABCAA \Rightarrow aABCAA \Rightarrow aaBCAA \Rightarrow \\
aabCAA \Rightarrow aabbAA \Rightarrow aabbaA \Rightarrow aabbaa
\end{aligned}
$$

Необходимо отметить, что хотя порождающая грамматика и описывает процесс порождения цепочек языка L(G), но описание это не является алгоритмическим — в грамматике отсутствует одно из главных свойств алгоритма — *детерминированность*, т.е. не фиксируется конкретный порядок применения правил подстановки грамматики. За счет этого обеспечивается компактность описания языка. Зафиксировать такой перечисляющий алгоритм в общем случае можно различными способами, но для точного определения языка этого не требуется. Таким образом, формальная грамматика G потенциально задает множество алгоритмов порождения языка, при этом мощность этого множества алгоритмов совпадает с мощностью |L(G)|.

Порождающая грамматика в том виде, как она была определена выше, является мощным описательным средством, но все же еще очень общего характера. Практическое применение грамматик связано с решением проблемы *распознавания*. Проблема распознавания разрешима, если существует такой алгоритм, который за конечное число шагов дает ответ на вопрос, принадлежит ли произвольная цепочка над основным словарем грамматики языку, порождаемому этой грамматикой. Если такой алгоритм существует, то язык называется *распознаваемым*. Если к тому же число шагов алгоритма распознавания зависит от длины цепочки и может быть оценено до выполнения алгоритма, язык называется легко распознаваемым [8]. В противном случае не имеет смысла вести речь о построении транслятора для нераспознаваемого языка программирования. Поэтому на практике рассматриваются такие частные классы порождающих грамматик, которые соответствуют распознаваемым, а в большинстве случаев и легко распознаваемым языкам. Наиболее важные классы таких языков могут быть определены в рамках классификации языков, предложенной в 1959 г. американским лингвистом Н.Хомским (классификация по Хомскому). Он предложил классифицировать формальные языки по типу правил порождающих их грамматик.

Класс 0. Правила вывода грамматики имеют форму $\varphi \rightarrow \psi$ без каких либо ограничений на строки φ и ψ. Языки этого класса могут служить моделью естественных языков.

Класс 1. Все элементы P получают из формы $\varphi \rightarrow \psi$, где $\varphi = \xi_1 \alpha \xi_2$, $\psi = \xi_1 \beta \xi_2$; а $\xi_1, \xi_2 \in V^*$, $\alpha \in N$, $\beta \in V^+$. Порождающая грамматика с такими правилами называется грамматикой *непосредственно составляющих* или контекстной грамматикой (НС-грамматика). Языки, порождаемые грамматиками этого класса, называют *контекстно-зависимыми*. В НС-грамматике каждое правило вывода указывает подстановку некоторой непустой цепочки β вместо нетерминала α при условии, что заменяемый нетерминал α находится в окружении ξ_1 и ξ_2, т.е. строки ξ_1 и ξ_2 рассматриваются как контекст, в котором α можно заменить на β. Языки класса 1 могут порождаться также грамматикой, правая часть каждого правила которой не короче его левой части, т.е. длина цепочки при выводе в такой грамматике может только возрастать. Такая грамматика называется *неукорачивающей*. Класс НС-грамматик эквивалентен классу неукорачивающих грамматик [8]. Грамматика из *примера* 1.3.2.2 является НС-грамматикой, у которой $\xi_1 \xi_2 = \lambda$.

Класс 2. Все порождающие правила грамматики имеют вид: $A \to \beta$, где A — нетерминальный символ, а β — непустая цепочка из V, т.е. $\beta \in V^+$. Замена нетерминала A на строку β происходит без учета контекста, поэтому грамматики этого класса называют контекстно-свободными (КС-грамматиками). Если допустить, что $\beta \in V^*$, т.е. возможна пустая подстановка λ вместо нетерминала A, то грамматика называется *укорачивающей* КС-грамматикой (УКС-грамматика). Доказано, что по любой УКС-грамматике можно построить почти эквивалентную КС-грамматику, т.е. порождающую тот же язык, что и исходная грамматика, за исключением пустой цепочки. УКС-грамматики представляют практический интерес потому, что именно они используются для описания языков программирования. Почти эквивалентность УКС и КС-грамматик позволяет свести изучение УКС-грамматик к изучению соответствующих свойств КС-грамматик. Поэтому КС-грамматики играют главную роль при формальном изучении синтаксиса языков программирования и построении блока синтаксического анализа транслятора.

Класс 3. Все порождающие правила имеют вид: $A \to bB$ и $A \to b$, где $A,B \in N$, $b \in T$, т.е. правая часть правила является или единичным терминалом, или единичным терминалом, за которым следует единичный нетерминал. Языки *класса 3* называют языками с конечным числом состояний или *автоматными* (регулярными) языками, а порождающие их грамматики — автоматными грамматиками (А-грамматики). А-грамматики используются в основном на этапе лексического анализа.

Связь между приведенными классами языков и проблемой распознавания языков формулируется следующей теоремой.

Теорема 1.3.2.1. [8]. Язык L(G), порождаемый неукорачивающей грамматикой G, легко распознаваем.

Взаимосвязь (иерархия) языков, в соответствии с классификацией Хомского, определяется нижеследующей теоремой.

Теорема 1.3.2.2. [4]. Если язык L(G) регулярный, то он контекстно-свободный. Если язык L(G) контекстно-свободный, то язык $L(G)\backslash\lambda$ — НС-язык. Если язык L(G) — НС-язык, то он язык класса 0.

С другой стороны, основные классы языков могут быть определены классами абстрактных распознающих устройств (автоматов), которые также образуют соответствующую иерархию. На рис.1.2 [7] приведена иерархия языков и соответствующие ей иерархии грамматик и автоматов как

распознающих устройств, основные из которых будут подробно описаны при рассмотрении вопросов лексического и синтаксического анализа.

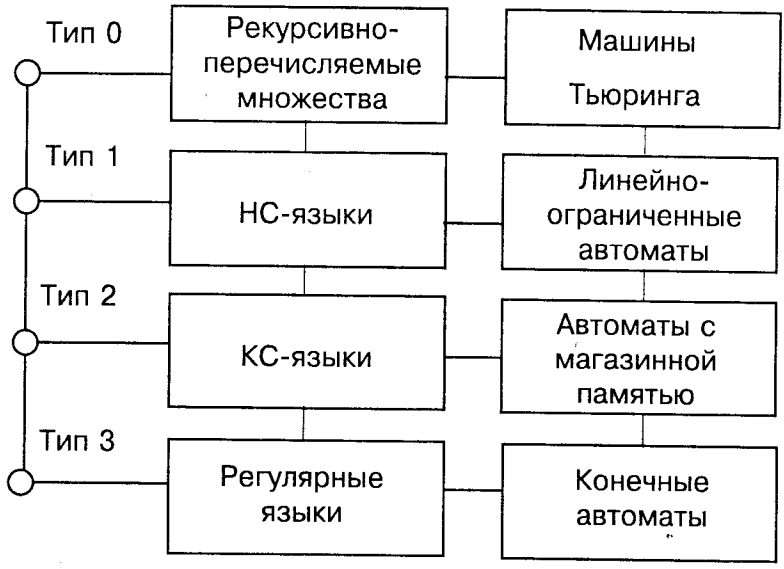

Рис. 1.2. Иерархия языков, грамматик и автоматов

Из четырех классов грамматик контекстно-свободные грамматики наиболее важны в приложении к языкам программирования. С их помощью можно определить большую, хотя и не всю, часть синтаксической структуры языка программирования.

Ранее было показано, как для описания синтаксиса языка используются нормальные формы Бэкуса. Оказывается, между БНФ и КС-грамматиками есть прямая связь — они, по существу, эквивалентны, различия касаются только обозначений. Так связке «::=» из БНФ соответствует в КС-грамматике отношение →, металингвистическим переменным из БНФ соответствуют в КС-грамматике нетерминальные символы, основным символам языка программирования из БНФ соответствуют терминальные символы КС-грамматики. В КС-грамматиках для сокращения записи правила с одинаковыми левыми частями также собирают в одно, используя в качестве разделителя альтернатив знак | (или).