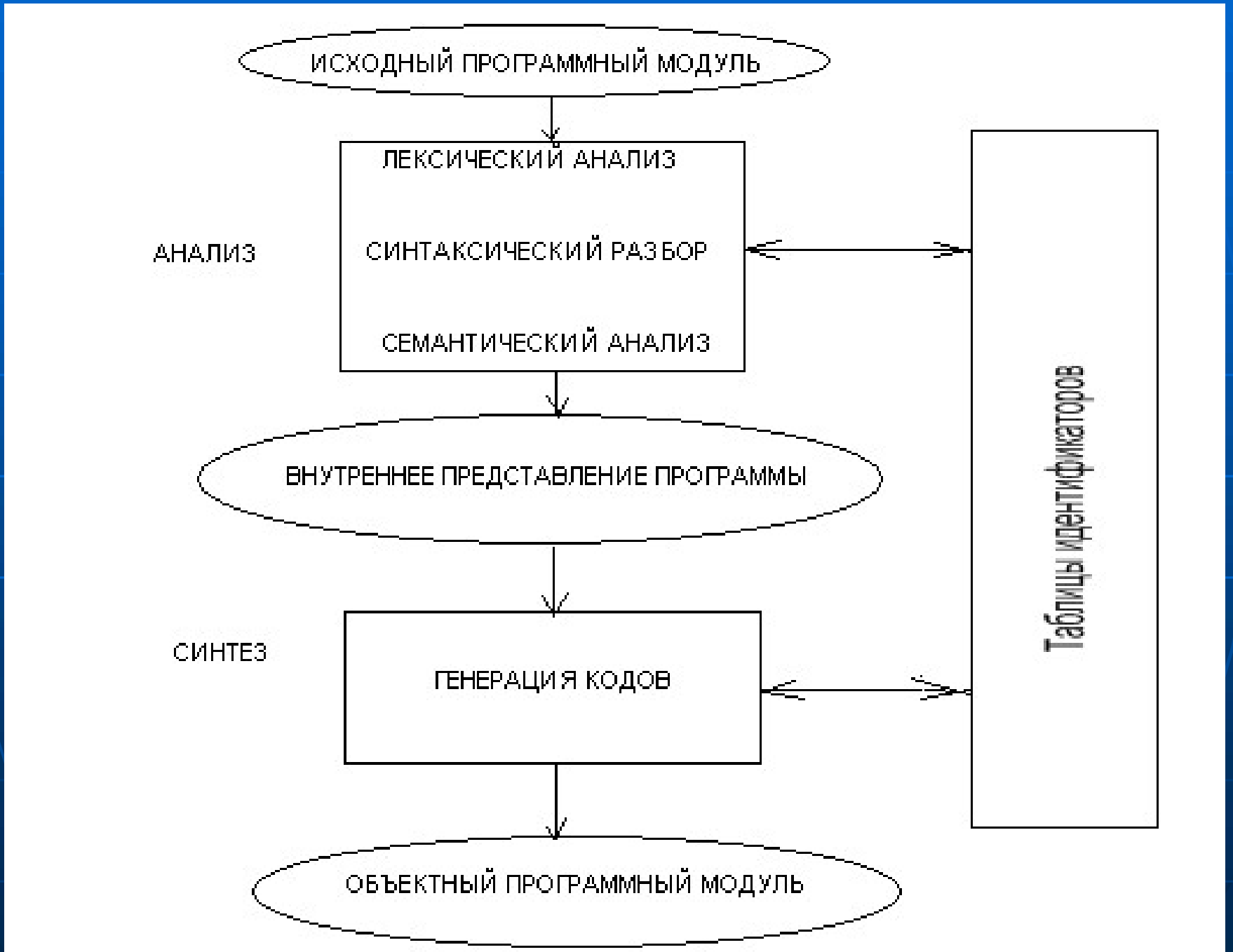
Формальные языки и грамматики

Этапы работы компиляторов



Этапы работы компи-ляторов

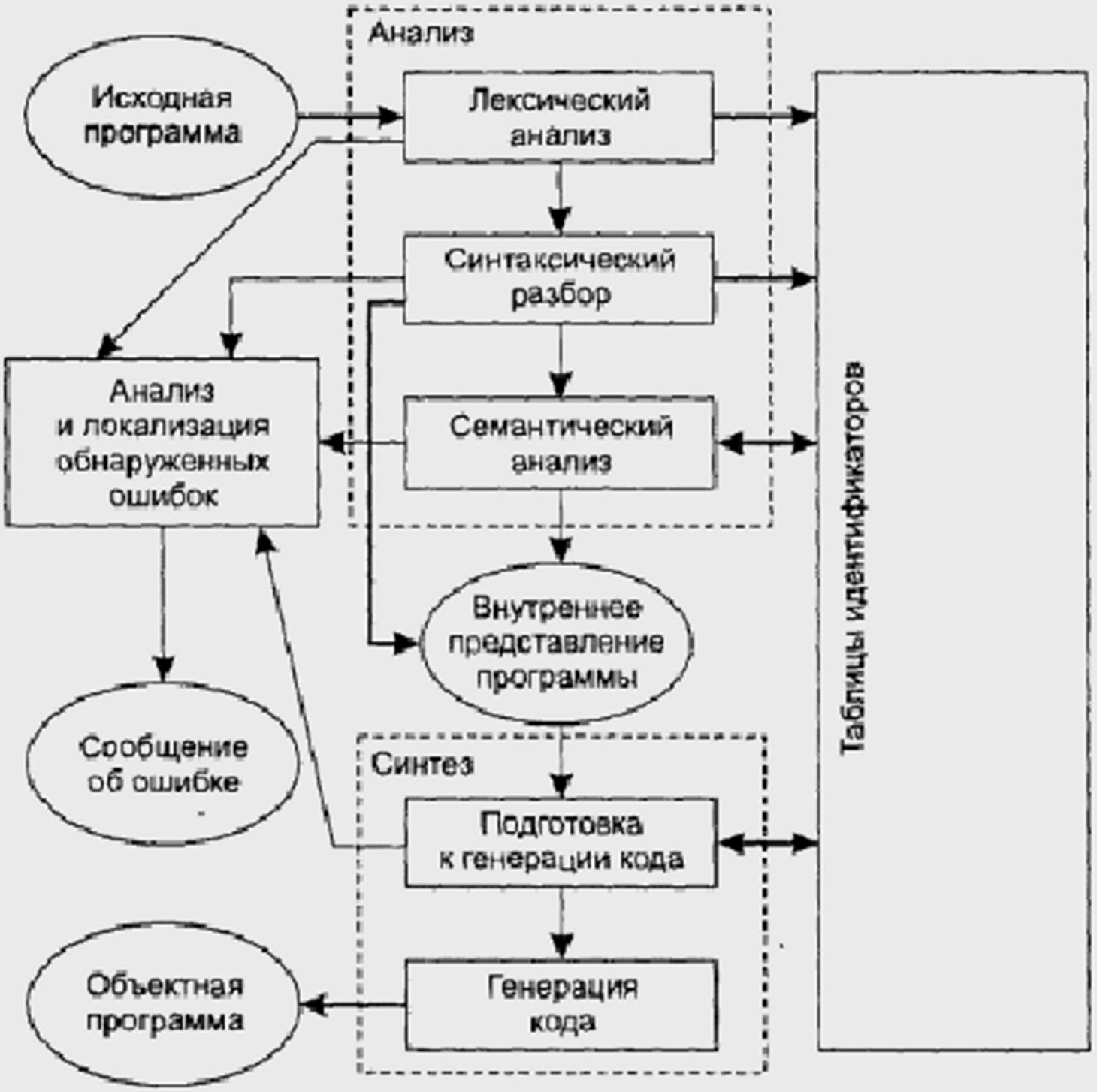


Рис. 2.1. Общая схема работы компилятора

Грамматики

Формально порождающая грамматика G это G(VT,VN,P,S)

VT - множество терминальных символов

VN - множество нетерминальных символов

P - множество правил грамматики вида $\alpha \rightarrow \beta$, где $\alpha \in V^{+}$, $\beta \in V^{*}$

S - целевой (начальный) символ грамматики $S \in VN$

Классификация Хомского

Согласно Хомскому, формальные грамматики делятся на четыре типа.

Тип 0 **неограниченные**

Для грамматики G(VT,VN,P,S), V=VT U VN все правила имеют вид:

$$\alpha \to \beta \text{ где } \alpha \in V^{+}, \beta \in V^{*}.$$

Тип 1
контекстно-зависимые

Грамматику типа 1 можно определить как контекстно зависимую, либо как неукорачивающую

Для грамматики G(VT,VN,P,S), V=VT U VN все правила имеют вид:

- $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$, где α_1, $a_2 \in V^*$, $\beta \in V^+$, $A \in VN$,
  Для контекстно-зависимых грамматик

- $\alpha \rightarrow \beta$, где α, $\beta \in V^+$, $|\alpha| \leq |\beta|$, для неукорачивающих грамматик

Тип 2 контекстно-свободные

Грамматику типа 2 можно определить как контекстно свободную либо как укорачивающую контекстно свободную

Для грамматики G(VT,VN,P,S), V=VT U VN все правила имеют вид:

- $A \rightarrow \beta$, где $A \in VN$, $\beta \in V^+$ для неукорачивающих грамматик

- $A \rightarrow \beta$, где $A \in VN$, $\beta \in V^*$ для укорачивающих грамматик

Тип 3
регулярные

Грамматику типа 3, G(VT,VN,P,S), V=VT U VN можно определить либо как праволинейную либо как леволинейную.

- Грамматика называется праволинейной, если любое правило из P имеет вид:

  $A->\gamma B$ $A->\gamma$ $A,B \in VN, \gamma \in VT^*$

- Грамматика G называется леволинейной, если каждое из правил имеет вид:

  $A->B\gamma$ $A->\gamma$ $A,B \in VN, \gamma \in VT^*$

Формальные языки

Формальные языки

- Язык L(G) является языком типа K если его можно описать грамматикой типа K. Языки классифицируются в соответствии с типами грамматик, с помощью которых они созданы.

- Один и тот же язык может быть задан разными грамматиками, относящимися к разным типам. В таком случае, считается, что язык относится к наиболее простому из них.

Тип 0

Это языки с фразовой структурой (самые сложные языки, разговорные). Сюда можно отнести естественные языки.

- Время на распознавание предложений языка экспоненциально зависит от длины исходной цепочки символов

Тип 1

Контекстно-зависимые языки

- Применяются в анализе и переводе текстов на естественных языках.
- Распознаватели, построенные на их основе, позволяют анализировать тексты с учетом контекстной зависимости в предложениях входного языка.
- Время на распознавание предложений языка экспоненциально зависит от длины исходной цепочки символов.

Тип 2
Контекстно-свободные языки.

- Лежат в основе синтаксических конструкций большинства современных языком программирования.
- Языки данного класса распространены.
- Время на распознавание предложений языка полиномиально зависит от исходной цепочки символов.

Тип 3
Регулярные языки.

- Данные языки лежат в основе простейших конструкций языков программирования, на их основе строятся мнемокоды команд.
- Для работы с регулярными языками можно использовать регулярные множества и выражения, а так же конечные автоматы.
- Время на распознавание предложений всегда линейно зависит от длины исходной цепочки символов.

Нормальная форма Бэкуса-Наура

Нормальная форма Бэкуса-Наура

Терминальные символы записываются как обычные символы алфавита, а нетерминальные – как имена в угловых скобках.

<число>:=<цифра>|<цифра><число>
<цифра>->0|1|2|3|4|5|6|7|8|9

Нормальная форма Бэкуса-Наура

Грамматика БНФ состоит из подмножества правил вывода, каждая из которых определяет синтаксис некоторой конструкции языка программирования.

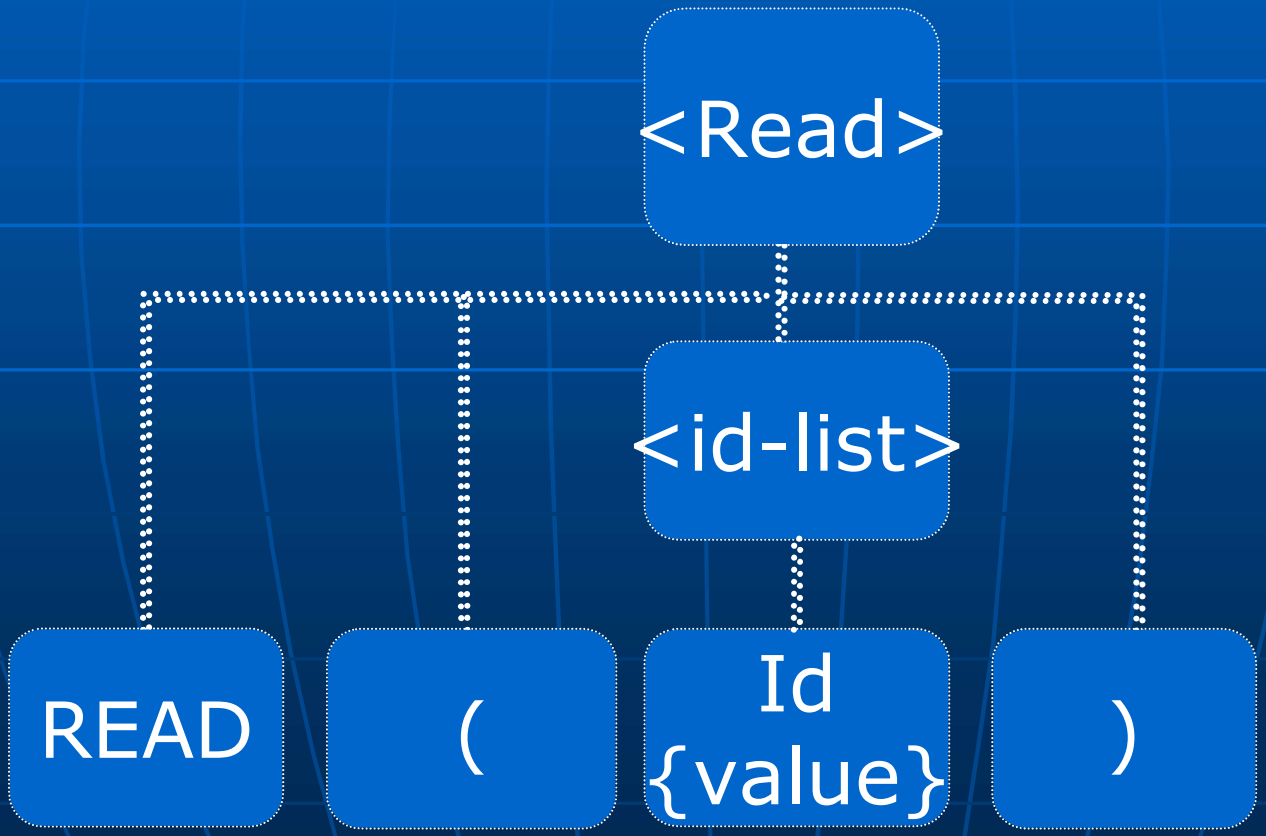
<read>->READ(<id_list>)
<id_list>->id|<id_list>,id

Нормальная форма Бэкуса-Наура

Результат анализа исходного предложения, в терминах грамматических конструкций удобно представить в виде дерева, которое называется деревом грамматического разбора или синтаксическим деревом.

Нормальная форма Бэкуса-Наура

Для предложения READ(VALUE) дерево должно выглядеть следующим образом :



Нормальная форма Бэкуса-Наура

Характер распознаваемых строк может намного упростить процесс лексического анализа, например любые вещественные числа могут сгенерировать посредством регулярного выражения (+/-) цифра*.цифра*. Реальное числосоставление:

1. Возможно знак
2. Последовательность из 0 или > цифр
3. (.)
4. Последовательность из 0 или > цифр

Распознаватели

Распознаватели

Состоят из следующих компонентов :

- Считывающее устройство
- Устройство управления
- Внешняя память

Распознаватели классификация

По видам считывающего устройства распознаватели могут быть:

- односторонними
- двусторонними.

Распознаватели классификация

По видам УУ распознаватели могут быть:

- детерминированные
- недетерминированные.

Распознаватели классификация

По видам внешней памяти распознаватели бывают следующих типов:

- Распознаватели без внешней памяти;
- Распознаватели с ограниченной внешней памятью;
- Распознаватели с неограниченной внешней памятью.

Распознаватели

- Классификация распознавателей определяет сложность алгоритма работы распознавателя.

- Сложность распознавателя напрямую связана с типом языка, входные цепочки которого может принимать распознаватель.

Распознаватели

Для каждого из 4-х основных типов языков существует свой тип распознавателя:

- Для языков с фразовой структурой нужен недетерминированный двусторонний автомат, имеющий неограниченную внешнюю память.
- Для контекстно-зависимых языков распознавателями являются двусторонние недетерминированные автоматы с линейно ограниченной внешней памятью.

Распознаватели

- Для контекстно-свободных языков распознавателями являются односторонние недетерминированные автоматы с магазинной (стековой) внешней памятью (МП-автоматы).
- Для регулярных языков распознавателями являются односторонние недетерминированные автоматы без внешней памяти (конечные автоматы).

Распознаватели

- В компиляторах распознаватели на основе регулярных языков используются для лексического анализа текста исходной программы.
- Задача разбора в общем виде заключается в том, что на основе имеющейся грамматики некоторого языка нужно построить распознаватель для этого языка.
- Заданная грамматика и распознаватель должны быть эквивалентны, т.е. определять один и тот же язык.

Лексический анализ

Лексический анализ. Преобразования грамматик

- алгоритм определения того, принадлежит ли анализируемая цепочка языку, порождаемому этой грамматикой (для леволинейного типа)
  - 1-й символ исходной цепочки $a_1a_2a_3a_4….a_n$# заменяется нетерминальным символом A, для которого в грамматике есть правило вывода
  - A->a_i
  - Полученный нетерминал A и расположенный непосредственно справа это него очередной терминал a_i исходной цепочки, заменяется нетерминалом B, для которого в грамматике есть правило вывода вида B->Aa_i.

Лексический анализ. Преобразования грамматик

1.Прочитана вся цепочка, на каждом шаге находилась единственная нужная свертка, на последнем шаге свертка произошла к символу S. Это значит, что цепочка принадлежит языку. $a_1a_2a_3...a_n\#\in L(G)$.

2. Прочитана вся цепочка. На каждом шаге находилась единственная нужная свертка, на последнем шаге – свертка произошла к символу, отличному от S. Это означает, что исходная цепочка не принадлежит языку а.

Лексический анализ. Преобразования грамматик

3. На некотором шаге не нашлось нужной свертки, т.е. для полученного на предыдущем шаге нетерминала A и расположенного непосредственно справа от него очередного терминала a_i исходной цепочки, не нашлось нетерминала B, для которого в грамматике было бы правило вывода $B->Aa_i$. Это означает, что исходная цепочка не принадлежит языку.

4. На некотором шаге работы алгоритма оказалось, что есть более одной подходящей свертки, т.е. в грамматике разные нетерминалы имеют правила вывода с одинаковыми правыми частями и поэтому не понятно к которому из них производить свертку.

Характеристики конечного автомата:

1) Конечным множеством состояний K.
2) Конечным множеством входным алфавитом.
3) Множеством переходом.
4) Начальным состоянием.
5) Множеством последних сотояний.

Цепочки вывода

Цепочки вывода

Выводом называется процесс порождения предложения языка на основе правил, определяющих язык грамматики.

Цепочки вывода

- Цепочка $\beta=\delta_1\gamma\delta 2$ называется непосредственно выводимой из цепочки $\alpha=\delta_1\omega\delta_2$ в грамматике $G(VT,VN,P,S)$ $V=VT \cup VN$; $\delta_1, \gamma, \delta_2 \in V^*$; $\omega \in V^+$, если в грамматике G существует правило $\omega->\gamma \in P$

- Непосредственная выводимость цепочки β из цепочки α обозначается как $\alpha=>\beta$.

- Цепочка β выводима из цепочки α в том случае, если можно взять несколько символов цепочки α, заменить их на другие символы согласно некоторому правилу грамматики можно получить цепочку β.

Цепочки вывода

Цепочка β называется выводимой из цепочки α в том случае, если выполняется одной из 2-х условий:

1. α=>β
2. существует γ такая, что α=>* γ и γ=> β

Такое определение выводимости цепочки является **рекурсивным**.

Т.е цепочка β, выводима из α, если может построить последовательность непосредственно выводимых цепочек от α к β следующего вида:

- α=>γ1=> γ2=>….=> γn=>β n>=1

Цепочки вывода

- Последовательность непосредственно выводимых цепочек называется **выводом** или **цепочкой вывода**.
- Каждый переход от одной непосредственно выводимой цепочки к следующей цепочке вывода называется **шагом вывода**.

Цепочки вывода

Пример 1:

G({0,1,2,3,4,5,6,7,8,9, -, +,},{S,T,F},P,S)

P: S->T |+T |-T T->F|TF F->0|1|2|3|4|5|6|7|8|9

Построим несколько произвольных цепочек вывода.

1. S=>-T=>-TF=>-TFF=>-FFF=>-4FF=>-47F=>-479
2. S =>T =>TF=>T8=>F8=>18
3. T=>TF=>T0=>TF0=>T50=>F50=>350
4. TFT=>TFFT=>TFFF=>FFFF=>1FFF=>1FF4=>10F4=>1004
5. F=>5

1.S=>*-479 или S=>+-479 или $S=>^{7}-479$

2. S=>*18 или S=>+18 или $S=>^{5}18$
3. T=>*350 или T=>+350 или $T=>^{6}350$
4. TFT=>*1004 или TFT=>+1004 или $TFT=>^{7}1004$
5. F=>*5 или F=>15(утверждение F=>+5 неверно)

Цепочки вывода

- Вывод называется законченным, если на основе цепочки β, полученной в результате вывода нельзя больше сделать ни одного шага вывода ;
- Цепочка β, получается в результате законченного вывода, называется конечной цепочкой вывода.

Цепочки вывода

- Цепочка символов αЄV* называется **сентенциальной формой** грамматики, если она выводима из целевого символа грамматики S=>*α.

- Если цепочка αЄVT получена в результате законченного вывода, то она называется **конечной сентенциальной формой**.

Цепочки вывода

- Дерево называется деревом вывода(деревом разбора) в контекстно свободной грамматике G, если:
  - Каждая вершина дерева помечена символом из множества VNUVTUε,при этом корень дерева помечен символами VTU ε;
  - Если вершина дерева помечена символом AЄVN, а ее непосредственные потомки $a_1,a_2...a_n$,где каждая aЄ(VNUVT),то $A\rightarrow a_1,a_2...a_n$ – правило вывода в данной грамматике
  - Если вершина дерева помечена символом AЄVN,а ее непосредственный потомок символом ε, то $A\rightarrow \varepsilon$ является правилом этой грамматики

Дерево вывода

- Нисходящий способ посторения
  - Формируется от корня к листьям
  - На каждом шаге для вершины помеченной нетерминальным символом пытаются найти такое правило вывода, чтобы имеющиеся в нем терминальные символы проектировались на символы исходной цепочки.

Дерево вывода

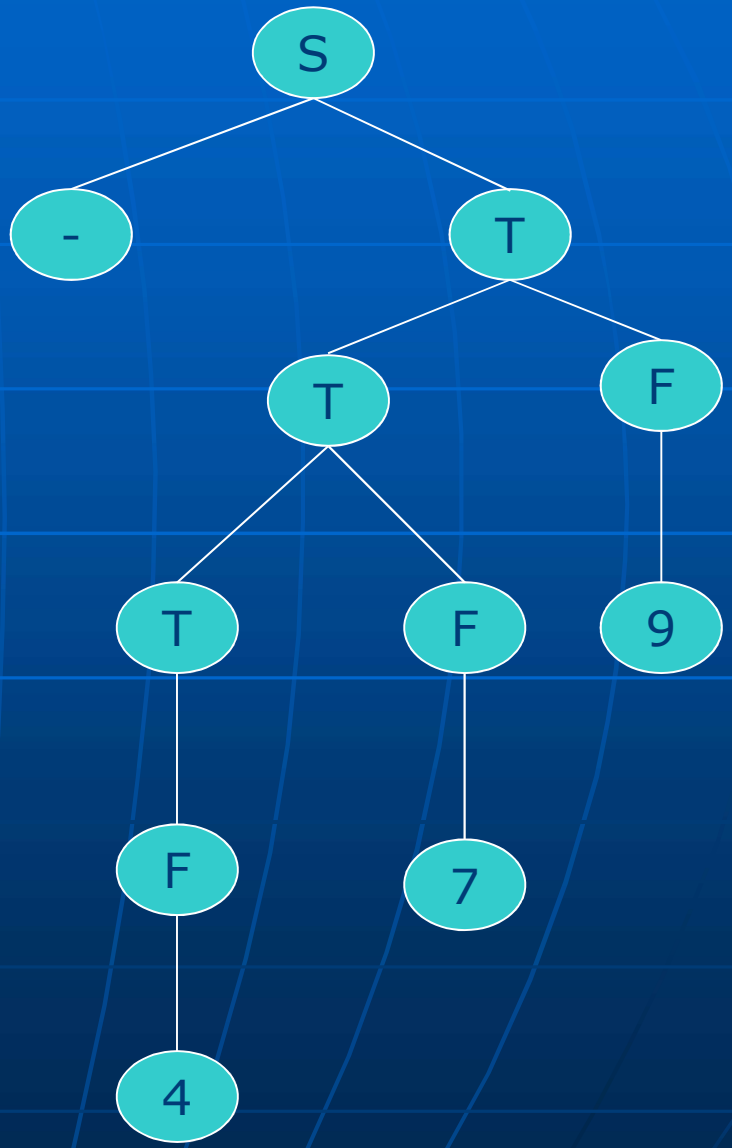
- **Пример 1**:
  - G({0,1,2,3,4,5,6,7,8,9, -, +,},{S,T,F},P,S)
  - P: S->T |+T |-T T->F|TF F->0|1|2|3|4|5|6|7|8|9

- S=>-T=>-TF=>-TFF=>-FFF=>-4FF=>-47F=>-479

Дерево вывода (построение «Сверху вниз»)

S=>-T=>-TF=>-TFF=>-FFF=>-4FF=>-47F=>-479

1. Целевой символ помещается в корень дерева
2. Корневой символ раскрывается на несколько символов первого уровня
3. Выбирается крайняя вершина обозначенная нетерминальным символом
   - выбирается правило
   - раскрывается на несколько вершин следующего уровня
4. Построение заканчивается, если все вершины обозначены терминальными символами
   - Иначе переходим к 2



Дерево вывода

- Восходящий способ посторения дерева
  - Исходную цепочку пытаются свернуть к начальному символу S
  - На каждом шаге ищут подцепочку, которая совпадает с правой частью какого-либо правила вывода
  - Если цепочка находится, то она заменяется нетерминалом из левой части

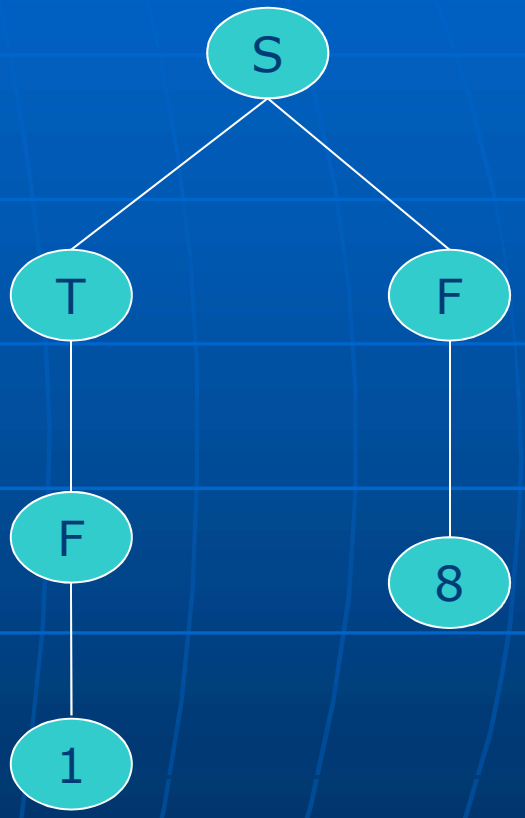
Дерево вывода

- **Пример 2**:
- G({0,1,2,3,4,5,6,7,8,9, -, +,},{S,T,F},P,S)
- P: S->T |+T |-T T->F|TF F->0|1|2|3|4|5|6|7|8|9

- S =>-T =>TF=>T8=>F8=>18

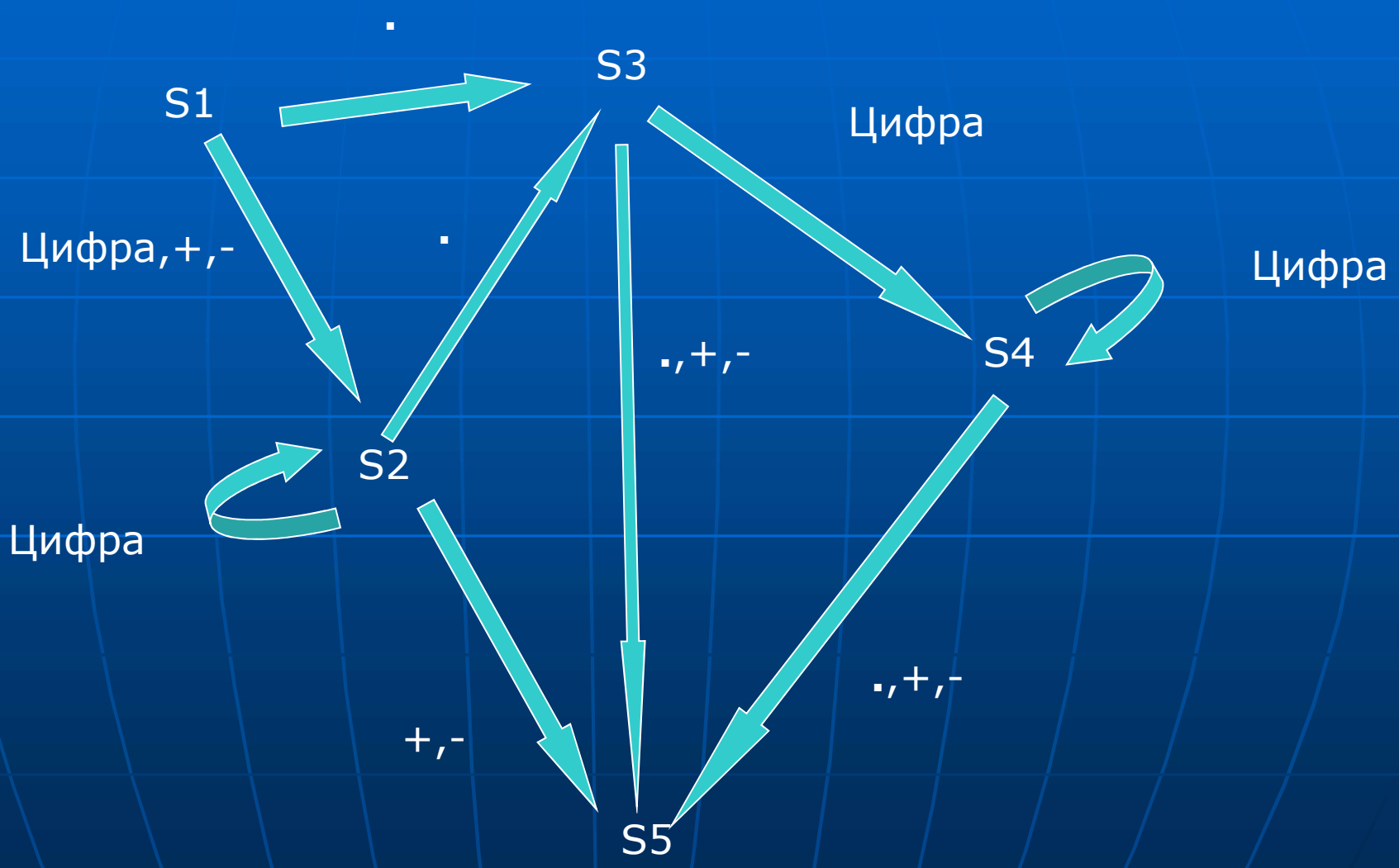
Дерево вывода (метод построения «Снизу вверх»)

1. Выбираются терминальные символы конечной цепочки
2. Выбирается правило, правая часть которого соответствует крайним символам в слое дерева
3. Выбранные вершины соединяются с новой вершиной, выбранной из левой части правила
4. Построение закончено, если достигнута корневая вершина
   - Иначе вернуться к 2



S =>-T =>TF=>T8=>F8=>18

Конечный автомат



Лексический анализ

- Просмотр компилируемой программы
- Распознавание лексем, составляющих предложение исходного текста
- Лексический анализатор преобразовывает исходную программу в последовательность символов
- Лексический анализатор выполняет чтение строк исходной программы
- Печать листинга исходной программы.

Лексический анализатор

- Замена в программе идентификаторов,констант ограничителей, и служебных слов лексемами;
- Лексический анализ уменьшает длину программы, устраняя из ее исходного представления несуществующие пробелы и комментарии
- Если будет изменена кодировка в исходном представлении программы, то это отразится только на лексическом анализаторе.

Лексический анализатор

- Каждая лексема определяется каким-либо кодом(не зависит от программы)
- Получение кодов идентификатора происходит последовательно, начиная с начала по тексту программы
- Каждый экземпляр идентификатора заменяется одним и тем же кодом

Лексический анализатор

- Необходимо создавать таблицу идентификаторов, где будут храниться соответствие кодов и самих идентификаторов
- Если идентификатор с одинаковым именем присутствует в процедуре и в основной программе, необходимо создавать в таблице столько строк идентификатора, сколько раз объявлена эта переменная
- В таблицу необходимо добавить поле для занесения имени процедур.

Алгоритм работы лексического анализатора

1. Просмотреть входной поток символов программы на исходном языке до обнаружения очередного символа, ограничивающего лексему
2. Для выбора части входного потока выполняется функция распознавания лексемы
3. При успешном распознавании информация о выделенной лексеме→таблица лексем и к 1.
4. При неуспешном распознавании, выдается сообщение об ошибке и дальнейшие действия
   - Выполнение прекращается
   - Выполняется попытка распознать следующую лексему (п.1)

Синтаксический анализ

- Во время синтаксического анализа предложения исходной программы распознаются как языковые конструкции описываемые используемые грамматикой. Можно рассматривать этот процесс как построение дерева грамматического разбора для транслируемых предложений.

Восходящий метод (метод предшествования)

- Основан на анализе пар последовательно расположенных операторов исходной программы и решений вопроса о том который из них должен выполняться первым.

Восходящий метод

- Анализируемое предложение рассматривается **слева направо** до тех пор, пока не будет найдено подвыражение, операторы которого имеют более высокий уровень операторного предшествования, чем соседние операторы
- Это подвыражение распознается в терминах правил перехода, используемой грамматики.
- Процесс продолжается до тех пор, пока не будет достигнут корень дерева, что и будет означать конец грамматического разбора
- Первым шагом при разработке процессора грамматического разбора, основанного на методе предшествования должно быть установление отношений предшествования между операторами грамматики.

Восходящий метод

	Begin	Read	Id	(	)
Begin		<·	<·		
Read				=	
Id					·>
(			< ·		=
)					

Восходящий метод

- Знак = означает что обе лексемы имеют одинаковый уровень предшествования и должна рассматривается грамматическим процессором в качестве одной конструкции языка.
- Если для пар отношение предшествования не существует это означает что они не могут находиться рядом ни в каком грамматически правильном предложении.
- Необходимо чтобы отношения предшествования были заданы однозначно.

Восходящий метод

- Процесс сканирования слева направо продолжается на каждом шаге грамматического разбора лишь до тех пор пока не определится очередной фрагмент предложения для грамматического распознавания
- Как только подобный фрагмент выделен он интерпретируется как некоторый очередной терминальный символ в соответствии с каким либо правилом грамматики.
- Этот процесс продолжается до тех пор пока предложение не будет распознано целиком, следует обратить внимание на то что каждый фрагмент дерева грамматического разбора строится начиная с конечных узлов вверх в сторону корня дерева
- Вся информация о грамматике и синтаксических правилах языка содержатся в матрице операторного предшествования

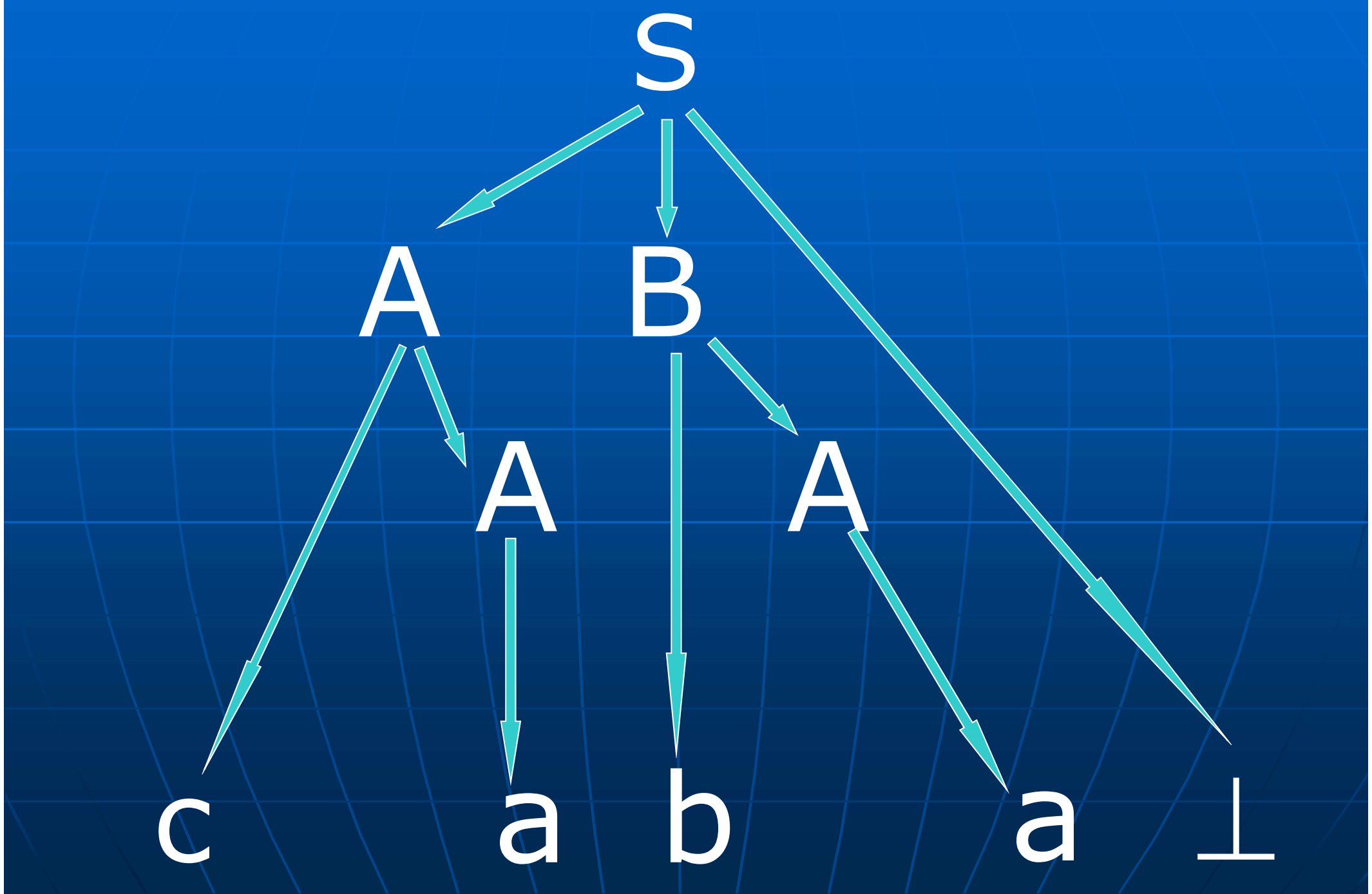
Низходящий метод (Метод рекурсивного спуска)

- Для каждого нетерминала грамматики создается своя процедура, носящая его имя
- Если такую подцепочку считать не удается, то процедура завершает свою работу вызовом процедуры обработки ошибки (цепочка не принадлежит языку, и останавливает разбор)
- Если подцепочку удалось найти, то работа процедуры считается нормально завершенной и осуществляется возврат в точку вызова.
- Тело каждой такой процедуры пишется непосредственно по правилам вывода соответствующего нетерминала: для правой части каждого правила осуществляется поиск подцепочки, выводимой из этой правой части.
- Терминалы распознаются самой процедурой, а нетерминалы соответствуют вызовам процедур, носящих их имена.

Метод рекурсивного спуска

- **Пример**: пусть дана грамматика G =({a,b,c, ⊥}, {S,A,B}, P, S), где
- P: $S \rightarrow AB\perp$
- $A \rightarrow a \mid cA$
- $B \rightarrow bA$
- и надо определить, принадлежит ли цепочка caba языку L(G).
- Построим вывод этой цепочки:
- $S \rightarrow AB\perp \rightarrow cAB\perp \rightarrow caB\perp \rightarrow cabA\perp \rightarrow caba\perp$
- Следовательно, цепочка принадлежит языку L(G).

$S \rightarrow AB\perp \rightarrow cAB\perp \rightarrow caB\perp \rightarrow cabA\perp \rightarrow caba\perp$



Семантический анализ.

- Контекстно-свободные грамматики, с помощью которых описывают языки программирования, не позволяют задавать контекстные условия, имеющиеся в любом языке.
- К контекстным условиям относятся:
  - каждый используемый в программе идентификатор должен быть описан, но не более одного раза в одной зоне описания;
  - при вызове функции число фактических параметров и их типы должны соответствовать числу типам формальных параметров;
  - обычно в языке накладываются ограничения на типы операндов любой операции, определенной в этом языке; на типы левой и правой частей в операторе присваивания, на тип параметра цикла, на тип условия в операторе цикла и условном операторе.

Семантический анализ.

- На этапе семантического анализа обрабатываются структуры, распознанные синтаксическим анализатором
  - На этой стадии выполняются и такие функции, как ведение таблицы символов
  - обнаружение ошибок
  - макрорасширений и выполнение инструкций, относящихся ко времени компиляции

Этапы семантического анализа

- Семантический анализатор выполняет следующие основные действия:
  - Проверку соблюдения во входной программе семантических соглашений входного языка
  - Дополнение внутреннего представления программы в компиляторе операторами и действиями, неявно предусмотренными семантикой входного языка
  - Проверку элементарных семантических (смысловых) норм языков программировния,напрямую не связанных со входным языком

Генерация кода

- После анализа программы, когда во все таблицы занесена информация, необходимая для генерации кода, компилятор должен переходить к построению соответствующей программы в машинном коде.
- Для получения машинного кода требуются два отдельных прохода:
  - генерация промежуточного кода;
  - генерация собственно машинного кода

Тетрады

- Представляет собой запись операций в форме из 4 составляющих:

- <операция> (<операнд_1>,<операнд_2>,<результат>)

Тетрады

- Тетрады представляют собой линейную последовательность команд
- Тетрады не зависят от архитектуры вычислительной системы

Тетрады

- **Пример:**

(-a+b)*(c+d)

можно представить тетрады следующим образом:

-a = 1

1+b=2

c+d=3

2*3=4

Триады

- Представляют собой запись операций в форме 3 составляющих
- <операция> (<операнд_1>,<операнд_2>

Триады

- Один или оба операнда могут быть ссылками на другую триаду, в том случае, если в качестве операнда данной триады выступает результат выполнения другой триады
- Триады при записи последовательно нумеруются для удобства указания ссылок одних триад на другие
- Триады требуют меньше памяти для своего представления, чем тетрады, кроме того, они явно отражают взаимосвязь операций между собой
- Триады ближе к двухадресным машинам, чем тетрады.

Триады

- **Пример**

a+b+c*d

можно представить тетрады следующим образом:

1. +(a,b)
2. *(c,d)
3. +(1,2)

Префиксная запись

- В префиксной нотации каждый знак операции ставится перед своими операндами
- **Например**, инфиксное выражение a+b в префиксной нотации примет вид +ab

Префиксная запись

- **Пример:**

  Выражение (a+b)*(c+d)

  в префиксной записи будет выглядеть

  *+ab+cd

Постфиксная запись

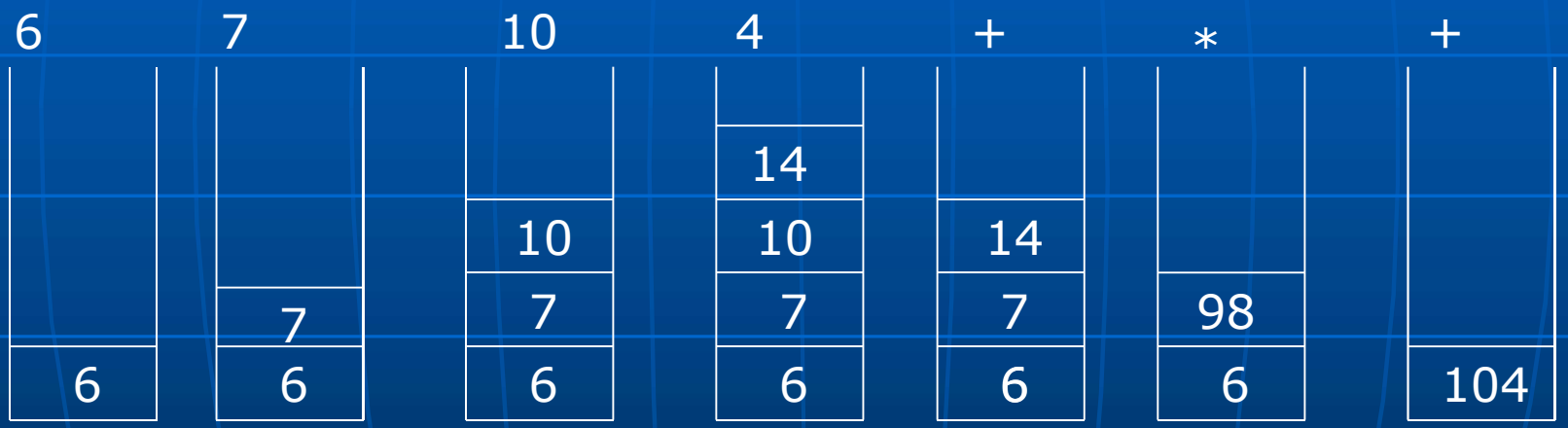
- В постфиксной нотации каждый знак операции ставится после своих операндов
- Операнды следуют в том же порядке, что и в инфиксной записи, а знаки операций строго в порядке их выполнения
- Не требует учитывать приоритет операций
- Не употребляются скобки

Вычисление выражений с помощью обратной польской записи

- Если встречается операнд, то он помещается в стек(попадает на верхушку стека)
- Если встречается знак унарной операции ,то операнд выбирается с верхушки стека, операция выполняется и результат помещается в стек
- Если встречается знак бинарной операции, то два операнда выбираются с верхушки стека, операция выполняется, и результат помещается в стек

Вычисление выражений с помощью обратной польской записи

6+7*(10+4)=104



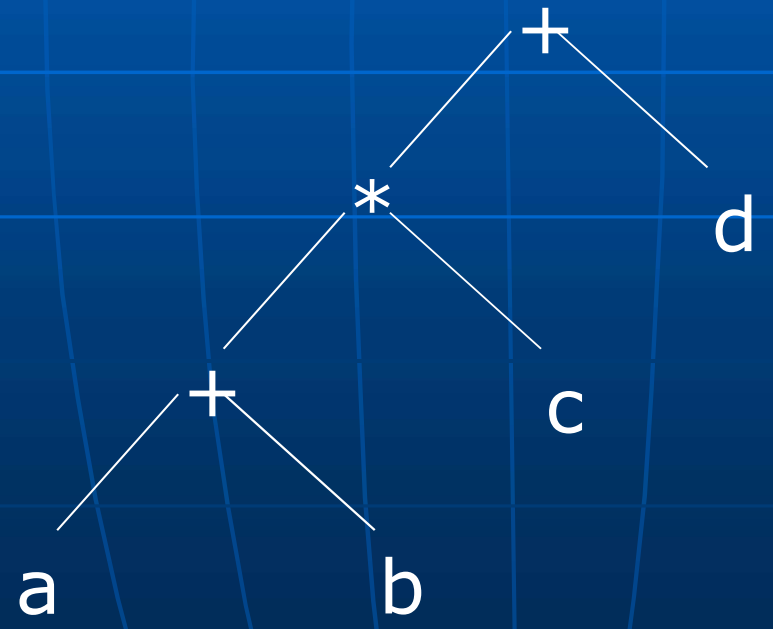
6 + 7 * (10 + 4) =104

Префиксная и постфиксная записи

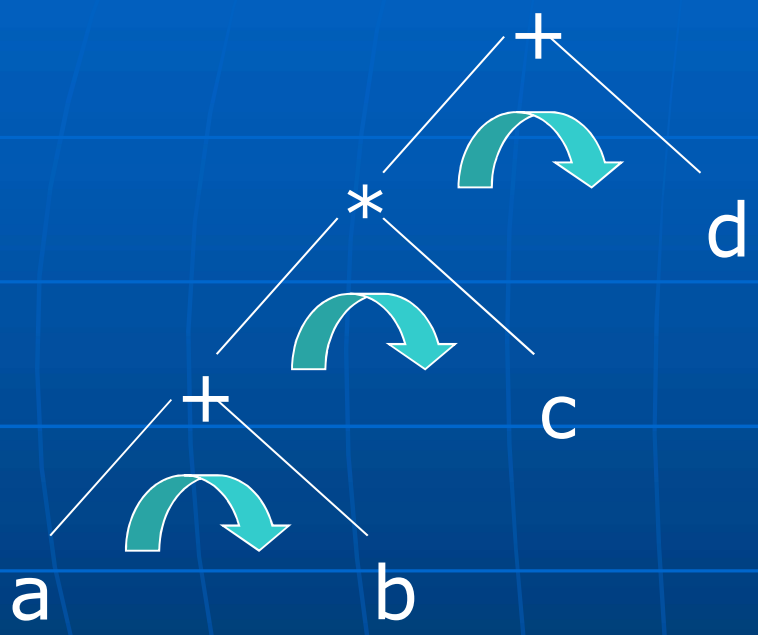
- Получение постфиксного или префиксного представления возможно разными путями. В частности используется алгоритм стека, который в простейшем случае сведется к:
  - сохранению идентификатора, когда он встретится при чтении слева направо;
  - помещению в стек знака операции;
  - в момент встречи конца выражения выдача из стека знака операции, который находится на вершине.

Метод получения постфиксного выражения из выражения, представленного в виде бинарного дерева

Выражение (a+b)*c+d



Чтобы получить постфиксное выражение используют порядок обхода, определенный Кнутом:



1.обход левого поддерева снизу

2.обход правого поддерева снизу

3.посещение корня

что дает в результате

ab+c*d+

Генерация машинного кода

- Самым простым способом генерации объектного кода является метод, который генерирует объектный код для каждого фрагмента программы как только распознан синтаксис этого фрагмента.
- Для реализации такого метода необходим набор подпрограмм, соответствующих каждому правилу и каждой альтернативе в правилах грамматики.

Генерация машинного кода

- Различают три формы объектного кода:
  - абсолютные команды, расположены в фиксированных ячейках памяти, после окончания компиляции такая программа немедленно выполняется.
  - программа на языке ассемблера, которую придется потом еще раз транслировать.
  - программа на машинном языке, представленная образами кодов и записанная во внешней памяти в виде двоичного объектного модуля.

Генерация кода

- При генерации собственно машинного кода используются таблицы, полученные на предыдущих этапах и сформированная программа в виде промежуточного кода в одной из его записей
- В генерации в данном случае можно выделить этап предварительной подготовки
- Каждая строка промежуточного кода читается, обрабатывается и на ее основе генерируется соответствующий ассемблерный или двоичный код

Генерация кода

- Для формализованного анализа в задаче необходимо построить формализованную модель ресурсов целевого компьютера
- В их состав должны входит:
  - модель информационных или запоминающих ресурсов.
  - модель операционных ресурсов, эквивалентных операторам языка программирования в виде таблиц команд, которая включает коды операций и базовых модификаций адресов в качестве ключей и соответствующие машинные коды как характеристики поиска.

Генерация кода

- Задача - обработать элементы таблицы символов, соответствующих формальным параметрам, а также переменных, описанных в этой процедуре и присвоить им адреса
- Для каждого элемента в таблице символов выполняется следующее:
  - элементу таблицы символов присваивается смещение, равное значение текущего предела, выделяемого в памяти, т.е. ему присваивается адрес первой свободной ячейки или области данных.
  - текущая таблица увеличивается на длину элементов, отведенных под формальные параметры.

Генерация кода

- Процесс генерации кодов отдельных команд отражается на каноническое множество адресов и модификаций команд. Во множество адресов включается как минимум:
  - прямая адресация глобальных и динамических данных.
  - относительная адресация аргументов и локальных данных, процедур и функций в стеке.
  - индексная адресация глобальных и локальных данных.

Генерация кода

- При изучении форматов объектных модулей решить следующие вопросы:
  - как указывается внешняя подпрограмма и данные, требуемые объектному модулю и каким образом определяются точки входа командам либо данных модуля, которые могут использоваться другими программами.
  - как достигается перемещаемость объектных программ, т.е. возможность размещать ее в любом месте ОП.

Загрузчики: абсолютные и перемещающие

Загрузчики: абсолютные и перемещающие

- Загрузчик копирует готовый к выполнению модуль в ПМ по команде ОС
- Загрузчик может быть либо двоичным (абсолютным), либо перемещаемым
  - При двоичной загрузке готовому к выполнению модулю присваиваются физические адреса еще во время компоновки
  - Перемещаемый загрузчик загружает программу в любую область ОП

Абсолютный загрузчик

- выполняет запись объектов программы в ОП и передачу управления на адрес начала ее исполнения
- Все выполняется за один просмотр
  - В начале просматривается запись-заголовок
  - Затем последовательно считываются записи тело программы, и содержащийся в них объектный код помещается в ОП по указанному адресу
  - Как только будет прочитана запись-конец, загрузчик передает управление по адресу, заданному в качестве адреса начала исполнения программы

Абсолютный загрузчик

- Одним из очевидных недостатков абсолютных загрузчиков является то, что требуется определить фактически адрес начала загрузки программы до ее ассемблирования
- Абсолютные загрузчики благодаря своей простоте используются в качестве начальных загрузчиков ОС.

Перемещаемые загрузчики

- обеспечивают эффективное разделение ресурсов компьютера при одновременном выполнении нескольких независимых программ, совместно использующих ОП и другие ресурсы. На ряду с простой функцией размещение программы в ОП
- выполняет также перемещение программ и при этом может использовать аппаратные средства.

Алгоритм построения перемещающих загрузчиков

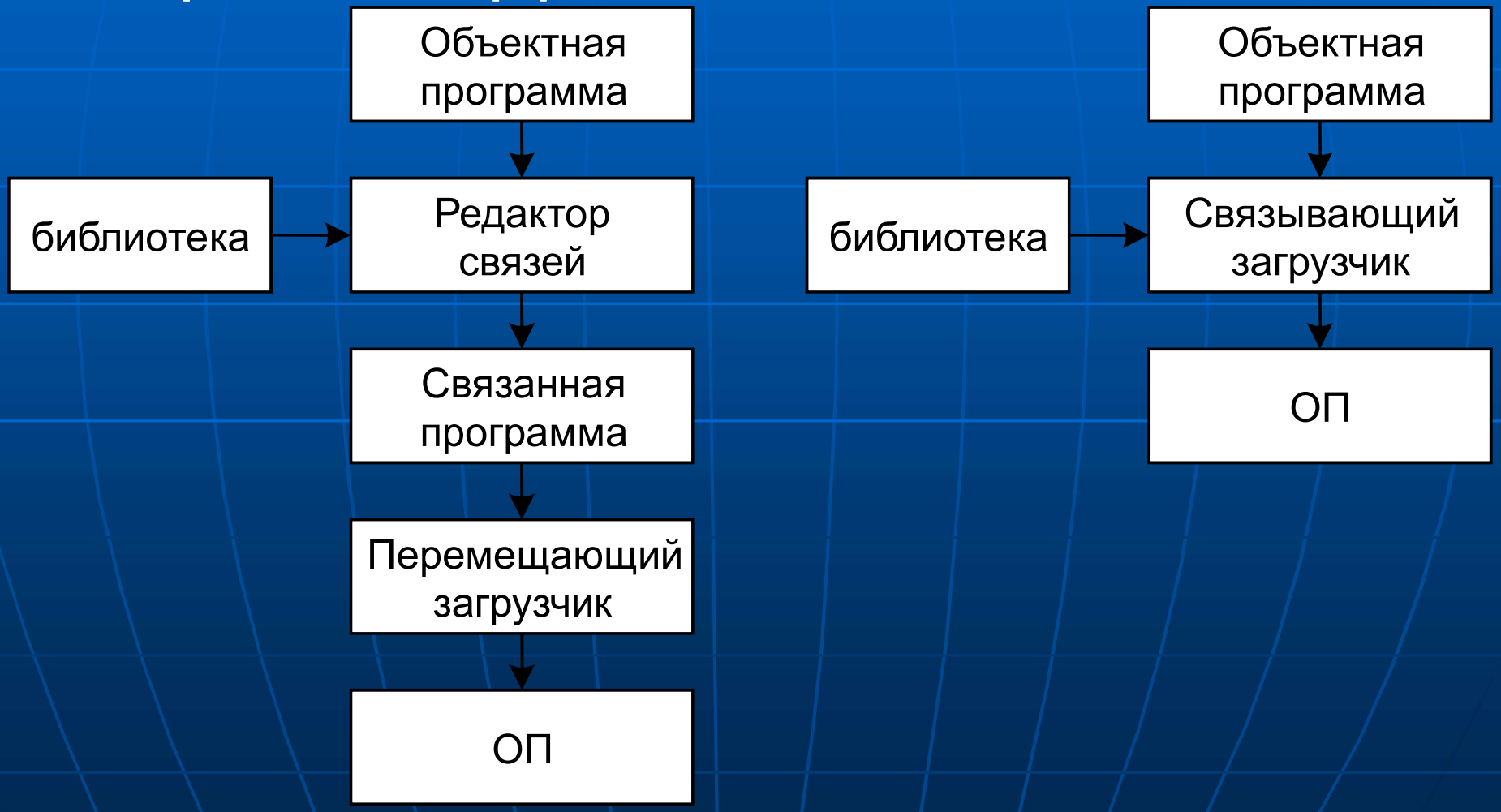
1. Перемещаемые с помощью записей-модификаторов – для описания каждого фрагмента объектного кода, требующего изменения при перемещении используется специальная запись-модификатор. Каждая запись-модификатор определяет начальный адрес и длину поля, значение которого необходимо изменить, а также тип требуемой модификации. Записи-модификаторы являются удобным средством для задания информации о перемещаемой программе. Алгоритм работы загрузчика, использующего записи-модификаторы состоит из следующих этапов:
   - Прочитать запись-заголовок;
   - Проверить имя и длину программы;
   - Получить PROGADDR от ОС;
   - Прочитать первую запись тела программы;
   - Пока тип записи не равен «E» выполнить следующие шаги:
     - Если объектный код задан в символическом виде, то преобразовать его во внутреннее представление;
     - Если тип записи равен «T», то переписать объектный код в заданное место ОП;
     - Если тип записи равен «N», то модифицировать заданное поле, т.е. прибавить к нему значение PROGADDR;
     - Прочитать следующую запись объектной программы;
   - Передать управление по адресу, заданному в записи конец, т.е. тип записи «E» плюс PROGADDR.

Алгоритм построения перемещающих загрузчиков

- Перемещение с помощью маски – эффективен на компьютерах, где используется прямая адресация и фиксированный командный формат.
- Формат записей тела программы тот же, что и ранее, за исключением того, что теперь с каждым словом объектного кода связан разряд перемещения.
- Эти разряды собираются вместе и образуют маску, которая записывается после указателя длины каждой записи тела программы.
- Если разряд перемещения установлен в единицу, то при перемещении программы начальный адрес программы добавляется к слову, соответствующему этому разряду.
- Значение разряда перемещения равное нулю показывает, что никаких преобразований при перемещении делать не нужно.

Связывающие загрузчики

Выполняют связывание и перемещение во время загрузки.



Связывающие загрузчики

- Работа загрузчика разделяется на две части :
  - Первая часть вырабатывает загрузочный модуль, состоящий из всех объектных сегментов, связанных и перемещаемых вместе относительно стандартного базового адреса.
  - Другая часть – операция загрузки – загружает модуль в основную память, настраивая адреса в соответствии с распределением памяти для модуля.

Связывающие загрузчики

- Редактор связей выполняет часть работы по распределению памяти, но основную работу по распределению памяти выполняет загрузчик.
- Вся необходимая основная память для пользовательской программы и данных назначается до начала выполнения программы, а все адреса настраиваются так, чтобы отразить это назначение. Если настройка происходит во время выполнения, непосредственно предшествуя каждому обращению к памяти, то адреса настраиваются динамически.
- Входящая информация для связывающего загрузчика состоит из набора объектных программ, т.е. управляющих секций, которые должны быть связаны друг с другом.
- Во время первого просмотра назначаются адреса для всех внешних ссылок, а во время второго – выполняет фактическая перемещение, связывание и загрузка.

Связывающие загрузчики

- Основная структура данных, необходимая для связывающего загрузчика – это таблица внешних имен (ESTAB).
  - Двумя другими важными переменными являются PROGADDR и CSADDR – начальный адрес той управляющей секции, которая обрабатывается загрузчиком в данный момент.
  - Этот адрес добавляется к всем относительным адресам данной управляющей секции для того, чтобы преобразовать их в фактические адреса.

Связывающие загрузчики

- Во время первого просмотра загрузчик обрабатывает только запись-заголовок и записи определения управляющих секций. PROGADDDR становится начальным адресом первой управляющей секции входного потока. Имя управляющей секции, полученное из записи-заголовка, записывается в ESTAB и ему присваивается текущее значение CSADDR
- Все внешние имена из записей-определений также заносятся в ESTAB. Значения их адресов получается путем сложения значения из записи-определения с CSADDR.
- После того, как прочитана запись-конец к CSADDR добавляется длина управляющей секции и таким образом получается адрес начала следующей управляющей секции.
- После завершения первого просмотра ESTAB содержит все внешние имена, определенные в данном наборе управляющих секций вместе с назначенными им адресами
- Далее осуществляется связывание, перемещение и загрузка.
- Запись-конец каждой управляющей секции может содержать адрес первой команды данной секции, с которой должно начинаться ее исполнение. Если адрес передачи управления задан более чем в одной управляющей секции, то загрузчик использует последний встретившийся. Если ни одна из управляющих секций не содержит адрес передачи управления, то используется PROGADDR.

Загрузчики

Загрузка исполняемых файлов. Концепция привязки.

Загрузка исполняемых файлов. Концепция привязки

- Привязка это адресная привязка объектного кода, запрашиваемая транслятором и выполненная сборщиком.
- Для специфицирования привязки необходимо указать четыре типа данных:
  - Место и тип привязываемого адресного пола
  - Один из двух возможных режимов привязки.
  - Цель, т.е. адрес в памяти, к которому обращается адресное поле.
  - Фрагмент, к которому имеет место обращение.
- Существует пять типов адресных полей:
  - Указатель (старшее слово имеет больший адрес).
  - База это старшее слово указателя (сборщику безразлично, есть ли младшее слово указателя или нет).
  - Смещение это младшее слово указателя (сборщику безразлично, следует ли за ним старшее слово).
  - Старший байт это старший байт смещения (сборщику безразлично, предшествует ли младший байт).
  - Младший байт это младший байт смещения (сборщику безразлично следует ш за ним старший байт).

Загрузка исполняемых файлов. Концепция привязки

- Сборщик Microsoft linker выполняет привязку в одном из двух режимов:
  - Внутрисегментный режим используется для привязки 8-битовых и 16-битовых смещений, указываемых в командах CALL, JUMP и JUMP SHORT.
  - Межсегментный режим используется при всех других режимах адресации (8086).

Защита информации в персональных ЭВМ

Средства, позволяющие контролировать доступ.

Средства, позволяющие контролировать доступ.

Системы защиты информации можно разделить на три класса:

- системы разделения доступа;
- защита от копирования;
- криптографическая защита информации

Функции системы защиты

- Идентификация защищаемых ресурсов;
- Аутентификация защищаемых ресурсов, т.е. установление их подлинности на основе сравнения с эталонными идентификаторами;
- Разграничение доступа к ПЭВМ;
- Разграничение доступа пользователей по операциям над ресурсами, защита с помощью программных средств;
- Администрирование (определение прав доступа к защищенным ресурсам, обработка регистрационных журналов, установка и снятие системы защиты);
- Регистрация событий (вход пользователей в систему, выход пользователей из системы, нарушение прав доступа к защищенным ресурсам);
- Контроль целостности и работоспособности системы защиты;
- Обеспечение безопасности информации при проведении ремонтно-профилактических работ;
- Обеспечение безопасности информации в аварийных ситуациях.

Защита от копирования

- Система защиты от копирования должна выполнять следующие функции:
- создание дискет, защищенных от копирования – эта функция предполагает запись на дискету информации, некопируемой обычными средствами.
- проверка ключевой дискеты – функция состоит в чтении записанной ранее информации и сравнении ее с эталонной. Наряду с функцией проверки ключевой дискеты необходимо предусмотреть функцию проверки и модификации определенных счетчиков, содержащихся в данных и скрытых на ключевой дискете. Данные могут быть расположены либо в секторах, невидимых ОС, либо в межсекторных промежутках.
- установка программы на жесткий диск – состоит в привязке программы к конкретной ПЭВМ и обеспечивает защиту от копирования программы с жесткого диска.
- защита программы от отладчика и модификации. Служит для предотвращения анализа алгоритма работы программ, как входящих в состав системы защиты от копирования, так и защищаемых этой системой. Защита программ от модификаций предупреждает несанкционированное изменение кода программы и обычно реализуется путем подсчета контрольной суммы по определенному алгоритму. В программе, защищенной от отладчика нежелательно использовать стандартные средства DOS и BIOS
- использование нестандартного форматирования дискет – форматирование с непоследовательными номерами секторов, запись нестандартного количества секторов на дорожку, пропуск одного или нескольких секторов, запись сектора нестандартной длины, запись сектора с особым номером и запись неверного значения для байтов, описывающих номер цилиндра и номер поверхности в адресном поле.