## Вопросы по теории компиляторов

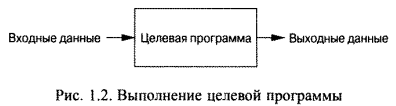
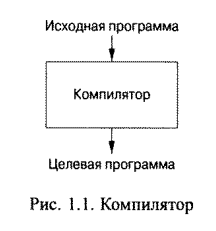
1. **Классификация трансляторов. Компиляторы, интерпретаторы.**



Транслятор – программа или техническое средство, выполняющее трансляцию программы.

Трансляция программы – преобразование программы, представленной на одном из языков программирования, в программу на другом языке и, в определенном смысле, равносильную первой.

Попросту говоря, компилятор – программа, которая считывает текст программы, написанный на одном языке – исходном, и транслирует его в эквивалентный текст на другом языке – целевом. Одна из важных ролей компилятора состоит в сообщении об ошибках в исходной программе, обнаруженных в процессе трансляции.



Интерпретатор представляет собой ещё один распространенный вид языкового процессора. Вместо получения целевой программы интерпретатор непосредственно выполняет операции, указанные в исходной программе, над входными данными, предоставляемыми пользователями.



Целевая программа на машинном языке, производимая компилятором, обычно гораздо быстрее, чем интерпретатор, выходные данные на основании входных. Однако интерпретатор обычно обладает лучшими способностями к диагностике ошибок, чем компилятор, поскольку он выполняет исходную программу инструкция за инструкцией.

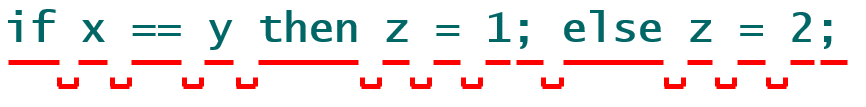
1. **Структура компилятора. Этапы (фазы) компиляции, их назначение.**

Структура компилятора:

1. Лексический анализ.
2. Синтаксический анализ (парсинг).
3. Семантический анализ.
4. Оптимизация.
5. Генерация кода.

Шаг 1: распознавание слов (наименьшая единица программы после букв)

На этапе лексического анализа программа разбивается на “слова” – токены



Лексический анализатор читает поток символов, составляющую исходную программу, и группирует эти символы в значащие последовательности, называющиеся лексемами. Для каждой лексемы строит выходной токен вида

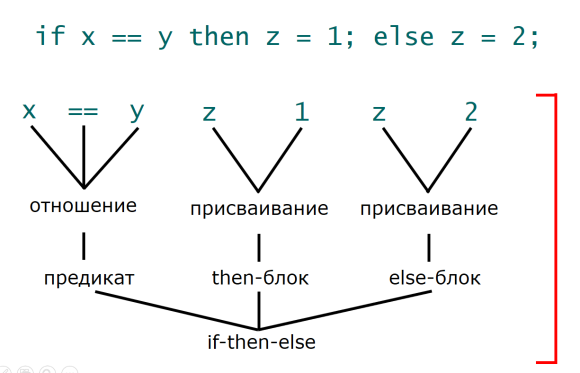


имя\_токена – абстрактный символ, использующийся во время синтаксического анализа;

значение\_атрибута – указывает на запись в таблице символов, соответствующая данном токену.

Шаг 2: понимание структуры предложения

Парсинг (синтаксический анализ)



Анализатор использует первые компоненты токенов, полученых при лексическом анализе, для создания древовидного промежуточного представления, которое описывает грамматическую структуру потока токенов. Типичным представлением является синтаксическое дерево, в котором каждый внутренний узел представляет операцию, а дочерние узлы – аргументы этой операции.

Шаг 3: понимание смысла предложения

Компиляторы выполняют ограниченный семантический анализ. Обнаружение несоответствий.

{

int Alice = 3;

{

int Alice = 4;

printf("%d", Alice);

}

}

Компиляторы выполняют различные проверки.



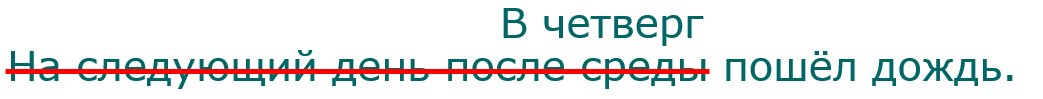
“Type mismatch” между словами Алиса и забыл

Семантический анализатор использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Также собирает информацию о типах и сохраняет её в синтаксическом дереве или таблице символов для последующего использования в процессе генерации кода.

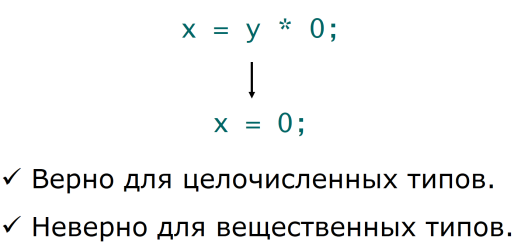
Важной частью семантического анализа является проверка типов.

Шаг 4: оптимизация

Не имеет аналога в естественных языках, похожая операция – правка текста



Автоматическое изменение программы таким образом, чтобы она выполнялась быстрее и использовала меньше памяти.



Фаза машинно-независимой оптимизации кода пытается улучшить промежуточный код, чтобы затем получить более качественный целевой код. Обычно более качественный означает более быстрый, но могут применяться такие критерии сравнения, как более короткий код или код, использующий меньшее количество ресурсов.

Шаг 5: генерация кода

Перевод на другой язык

Генератор кода получает в качестве входных данных промежуточное представление исходной программы и отображает его в целевой язык. Если целевой язык представляет собой машинный код, для каждой переменной, используемой программой, выбираются соответствующие регистры или ячейки памяти. Затем промежуточные команды транслируются в последовательность машинных команд, выполняющих те же действия. Ключевым моментом является аккуратное распределение регистров для хранения переменных.

1. **Лексический анализ. Задачи лексического анализа. Понятия лексемы и токена. Классы токенов.**

Основная задача лексического анализа состоит в чтении входных символов исходной программы, их группировании в лексемы и вывод последовательности токенов для всех лексем исходной программы. Поток токенов пересылается синтаксическому

анализатору для разбора. Одной из задач является отбрасывание

комментариев и пробельных символов. Еще одной задачей является

синхронизация сообщений об ошибках, генерируемых компилятором, с исходной

программой.

Токен представляет собой пару, состоящую из имени токена и

необязательного атрибута. Имя токена — это абстрактный символ, представляющий тип

лексической единицы, например конкретное ключевое слово или

последовательность входных символов, составляющую идентификатор. Имена то-

кенов являются входными символами, обрабатываемыми синтаксическим

анализатором.

Лексема представляет собой последовательность символов исходной

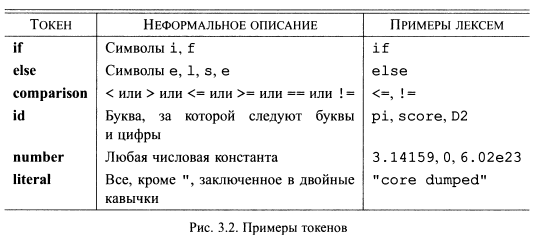
программы, которая соответствует шаблону токена и идентифицируется

лексическим анализатором как экземпляр токена.

Шаблон (pattern) — это описание вида, который может принимать

лексема токена. В случае ключевого слова шаблон представляет собой просто

последовательность символов, образующих это ключевое слово.



* Класс токена:
  + - идентификатор;
    - ключевое слово;
    - (
    - )
    - число;
    - и др.
* Классы токенов соответствуют множествам строк.
* Идентификатор:
  + строка букв и цифр, начинающаяся с буквы.
* Целое число:
  + непустая строка из цифр.
* Ключевое слово:
  + else, if, begin и т.д.
* Пробельные символы
  + непустая последовательность пробелов, переводов строк, табов и т.д.
* Классы токенов соответствуют множествам строк.



1. **Лексический анализ. Предпросмотр (backtracking, lookahead) и примеры его применения при лексическом анализе языков программирования.**

Задача чтения исходной программы усложняется тем, что часто надо заглянуть на один или несколько символов вперед, за очередную лексему, чтобы убедиться в корректности распознавания. Например, нельзя быть уверенным, что достигнут конец идентификатора, пока не встретится символ, не являющийся частью лексемы для id. Односимвольный оператор наподобие = может быть началом двухсимвольного оператора ==.

1. **Регулярные выражения. Назначение, базовые конструкции, составные конструкции.**

Регулярное выражение – шаблон, использующийся для сопоставления последовательности символов в строка.

Используется для описания языков, которые могут быть построены путем применения базовых и составных конструкций к символам некоторого алфавита.

Базовые конструкции:

* Одиночный символ

‘c’ = {“c”};

* Эпсилон

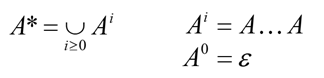
ε = {“”}, ε != ∅

Составные конструкции

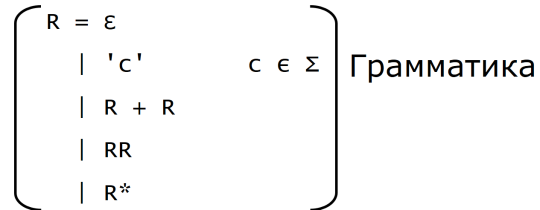
- Объединение A + B = {a | a ∈ A} ∪ {b | b ∈ B}

- Конкатенция AB = {ab | a ∈ A, b ∈ B}

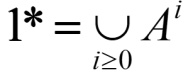
- Итерация A\*



* Регулярные выражения над алфавитом Σ — наименьшее множество выражений, включающее:

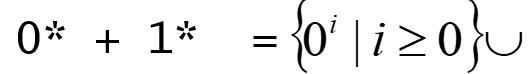


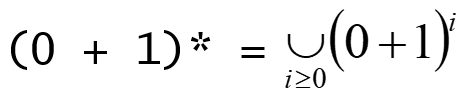
Σ = {0, 1}

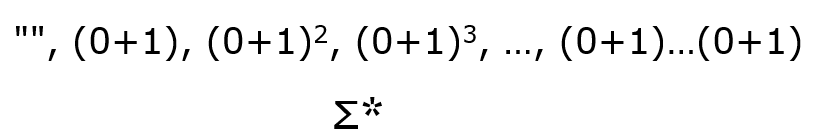


"" + "1" + "11" + "111" + "1111" + ... = все строки из единиц









Регулярные выражения задают регулярные языки. Регулярное выражение – синтаксис. Регулярный язык – множество строк.

1. **Формальные языки. Алфавит языка.**

Пусть Σ – множество символов(алфавит)

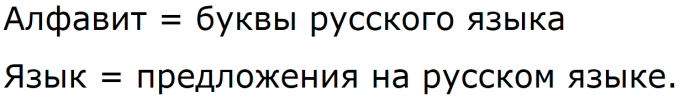
Формальный язык над алфавитом Σ – множество строк, состоящих из символов алфавита Σ.

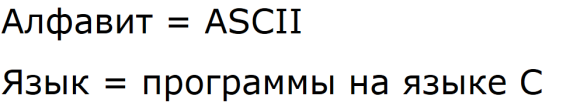
Термин алфавит означает любое конечное множество символов. Множество {0, 1} представляет собой бинарный алфавит.

Строка над некоторым алфавитом – конечная последовательность символов, взятых из алфавита.

Язык представляет собой любое счетное множество строк над некоторым фиксированным алфавитом.

Формальный язык – множество строк, состоящее из символов алфавита и составленных по определенным правилам (из символов данного алфавита).





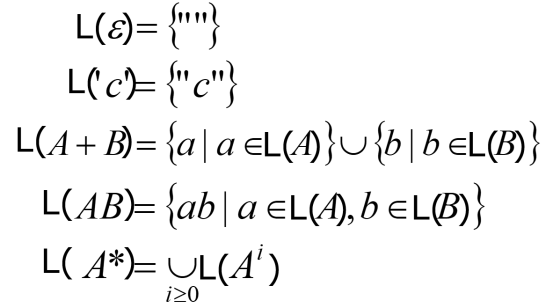
1. **Функция значения (meaning function).**

Функция значения L(x) задаёт взаимное соответствие синтаксиса и семантики.

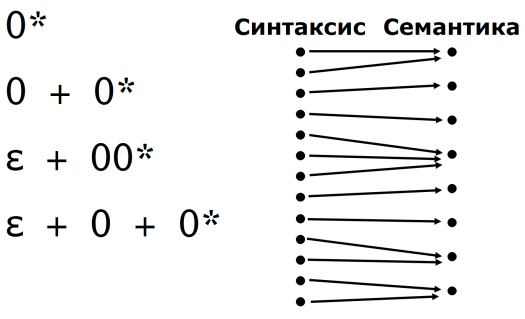
L(e) = M

e – регулярное выражения

M – множество строк



Функция значения позволяет разделить синтаксис и семантику, отделить проблему выбора нотации от остальных вопросов. Выражения и их значение(смысл) не всегда однозначно соответствуют друг-другу.



^

| ЧТО ЭТО?

1. **Лексические спецификации. Использование лексических спецификаций для лексического анализа.**

Первый этап разработки лексического анализатора – разработка лексической спецификации языка, т.е. описания правил записи лексем языка.

Часто для записи лексической спецификации используют регулярные выражения

Ключевые слова(“if”, “then”, “else”, …)

“i”f” + ‘t”h”e”n’ +

‘if’ + ‘then’ + ‘else’

Целое число – непустые строки из цифр.

digit = ‘0’ + ‘1’ + … + ‘8’ + '9';

digit digit\*

digit+

Идентификатор – множество строк, состоящих из букв и цифр и начинающихся с буквы

letter = [a-zA-z]

letter(letter+digit)\*

Пробельные символы – непустая строка из пробелов, переводов строк и tab-символов.

(‘ ’ + ‘\n’ + ‘\t’)+

Регулярные выражения могут использоваться для описания многих полезных языков. Регулярные выражения – спецификация языка. По-прежнему нужна реализация.

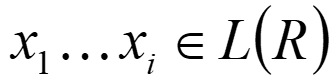
1. Написать регулярные выражения для каждого класса токенов.
   * Number = digit+
   * Keyword = 'if' + 'else' + …
   * Identifier = letter (letter + digit)\*
   * OpenParen = '('
   * …
2. Построить регулярное выражение R, задающее все возможные лексемы всех классов токенов.

R = Keyword + Identifier + Number + …

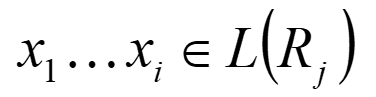
= R1 + R2 + …

1. Пусть входная строка — x1…xn.

Для 1 ≤ i ≤ n проверить:



1. Если условие выполняется, то:

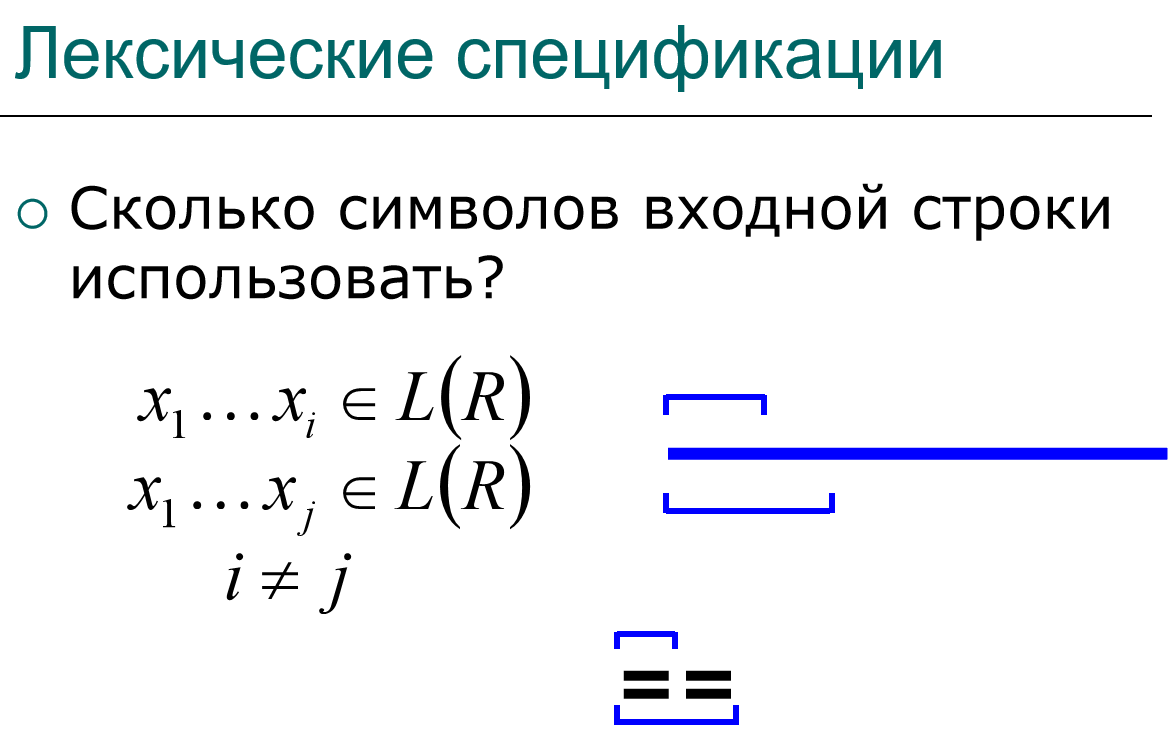


1. Удалить  из входной строки и перейти к 3.

Но для использования регулярных выражений необходимо избавиться от ошибок и неопределенностей.

1. **Правило «maximal munch». Проблемы применения и способы их решения.**

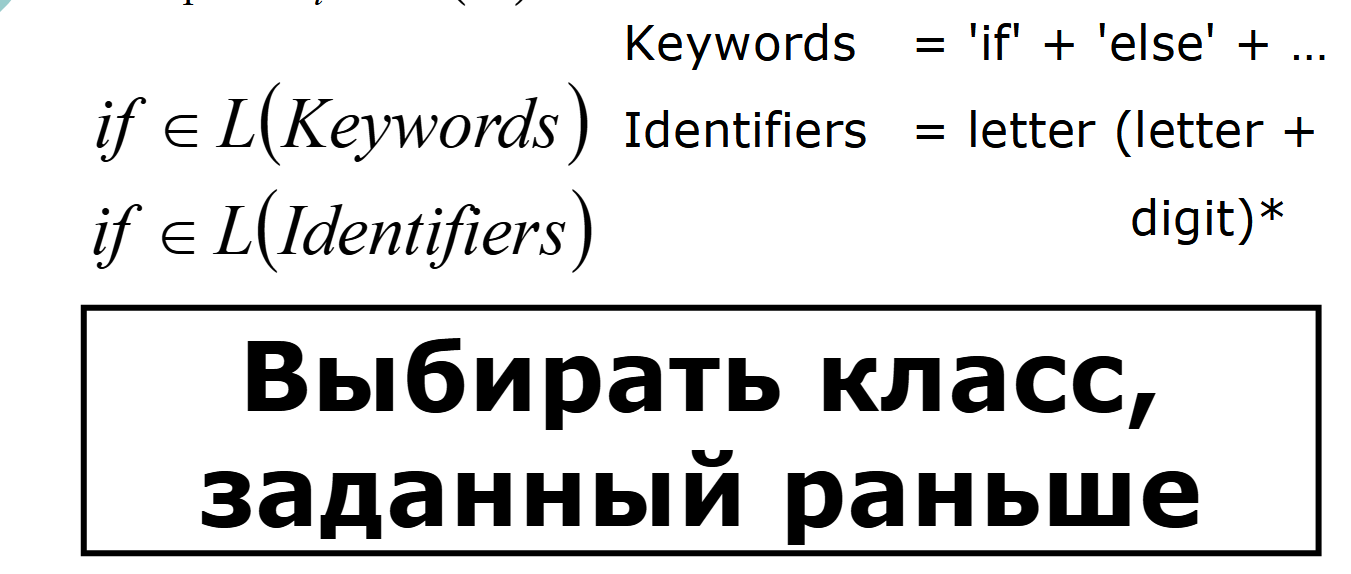
Правило “maximal munch” говорит нам о том, что из входной строки последовательность с максимальной длиной должна считаться совпадением



Строка x1..xi это просто =, строка x1…xj это уже ==. Соответственно из входной строки надо удалить == по алгоритму из предыдущего вопроса.

Проблемы:

Какой класс выбрать? Ответ: тот, который задан раньше



В случае, если ни одно правило, необходимо выдать ошибку.

1. **Конечные автоматы в лексическом анализе. Понятия «accept» и «reject». Связь между конечными автоматами и регулярными языками.**

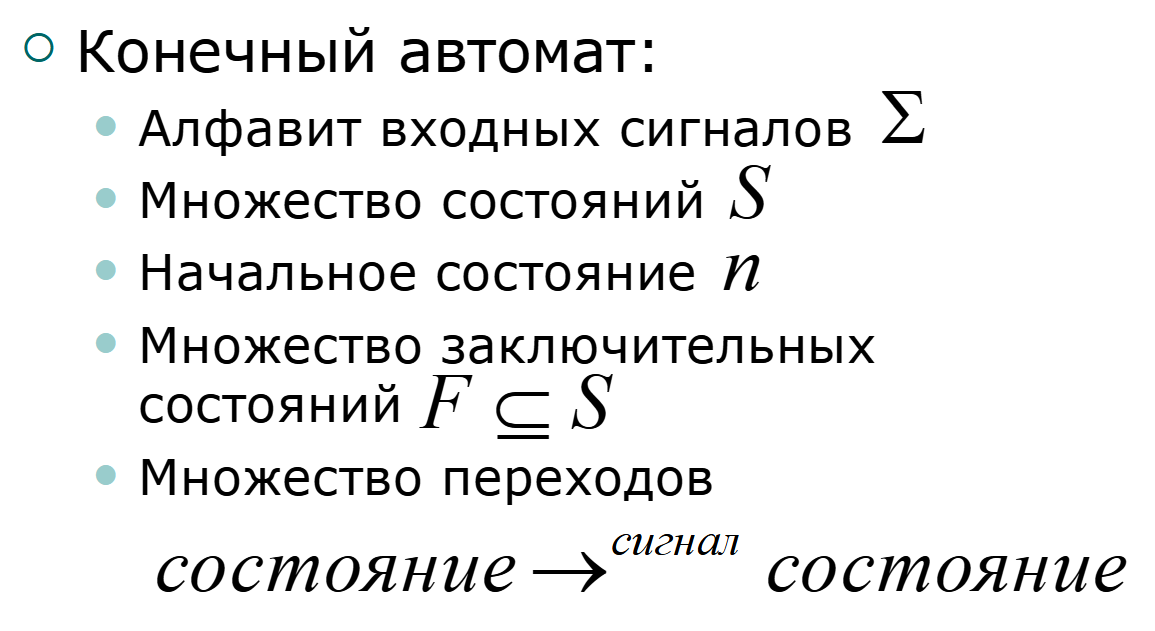
Регулярные выражения представляют собой спецификацию, а конечный автомат их реализацию.

Конечный автомат характеризуется алфавитом, множеством состояний, начальным состоянием, множеством заключительных состояний, множеством переходов

Состояние – сигнал 🡪 состояние

Конечные автоматы разделяются на два класса: НКА и ДКА. Оба класса способны распознавать одни и те же языки. По сути, это те же языки, именуемые регулярными языками, которые могут быть описаны регулярными выражениями.

Если достигнут конец входной строки и автомат находится в заключительном состоянии, то автомат допускает строку.



Графическое представление



Если достигнут конец входной строки и автомат находится в заключительном состоянии – он допускает(Accept). В ином случае – Reject.

Особый тип перехода – ε-перехода. Такие переходы позволяют автомату изменять своё состояние при этом не поглощая символа входной строка.

1. **Детерминированные и недетерминированные конечные автоматы (ДКА и НКА).**

Как НКА, так и ДКА можно представить в виде графа переходов, узлы которого представляют состояния, а помеченные дуги – функции переходов.

ДКА – детерминированный конечный автомат: для каждой пары “состояние-сигнал” только один переход, не должно быть ε-переходов.

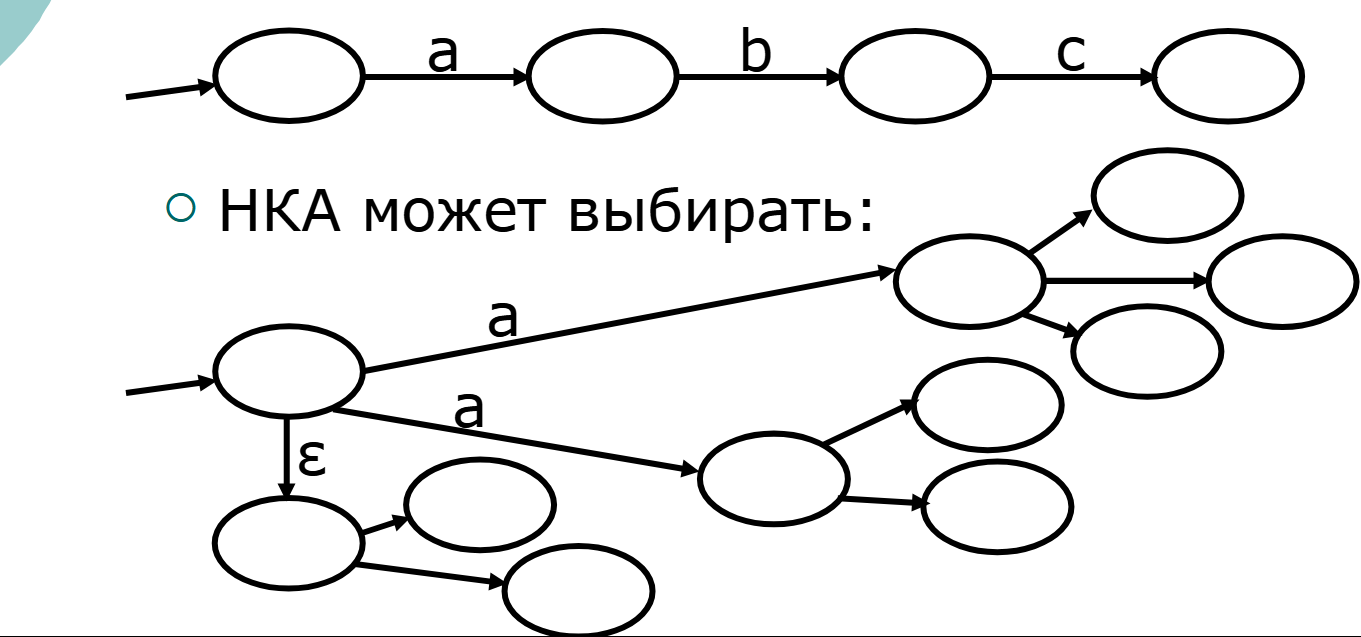
НКА – недетерминированный конечный автомат: для пары “состояние-сигнал” может быть несколько переходов, так же имеют место ε-переходы. Таким образом НКА способен выбирать.

НКА допускает строку, если хотя бы один набор решений приводит в заключительное состояние. НКА может находится в нескольких состояниях одновременно.

НКА и ДКА распознают одно и то же множество языков.

ДКА работают быстрее за счет отсутствия необходимости выбирать.

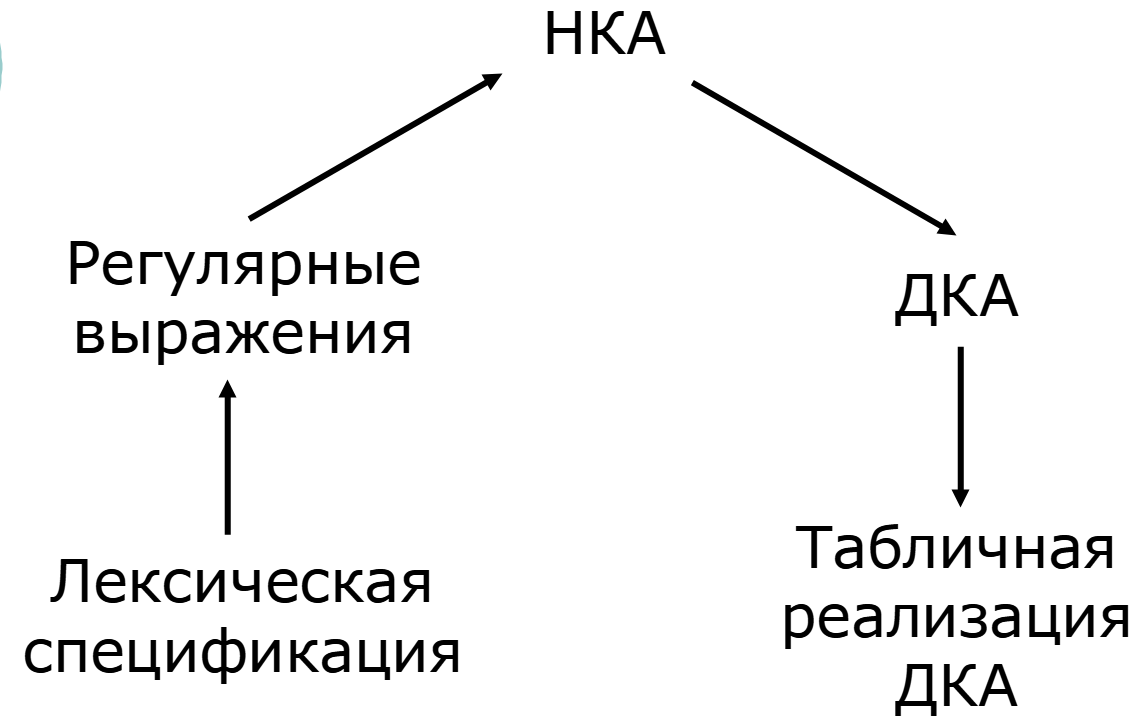
НКА имеет меньше состояний, поэтому компактнее.

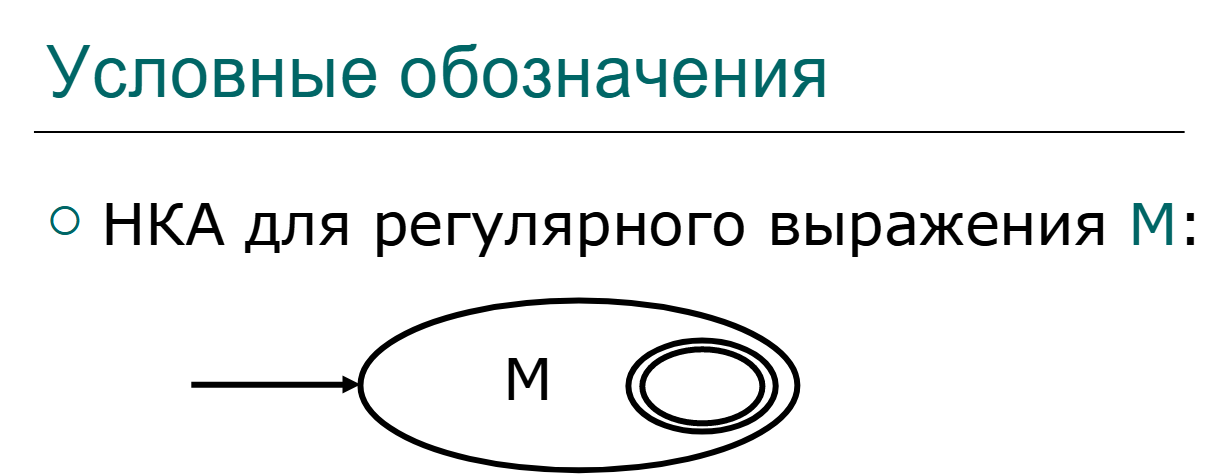


Вверху – ДКА, внизу – НКА.

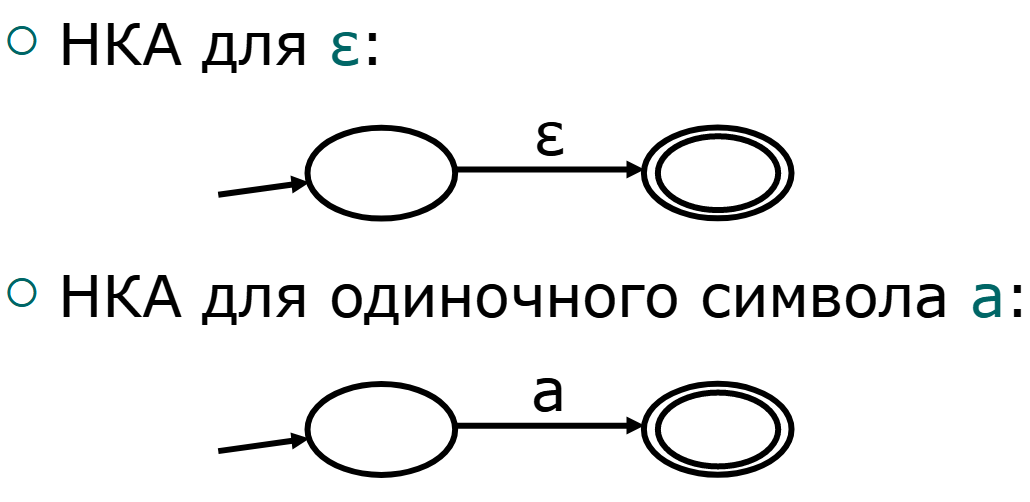
1. **Этапы построения лексического анализатора. НКА для простых и составных регулярных выражений.**

Этапы построения лексического анализатора

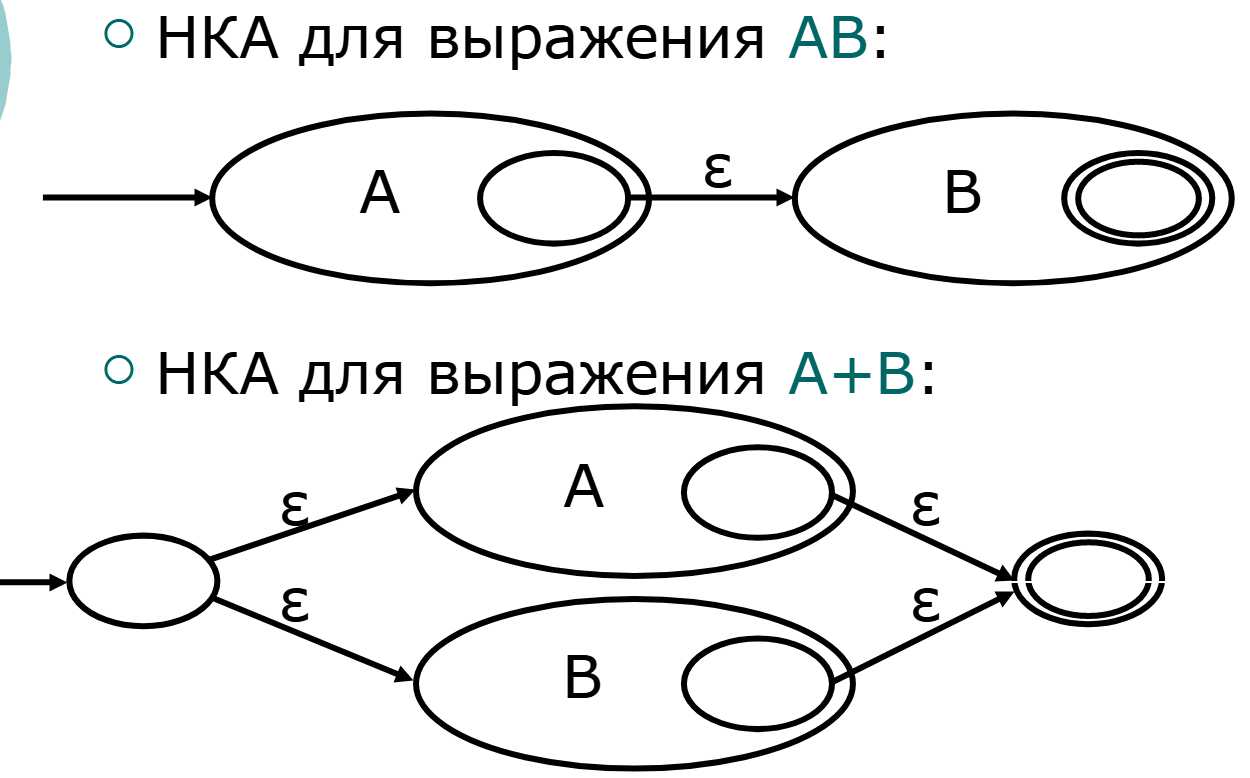


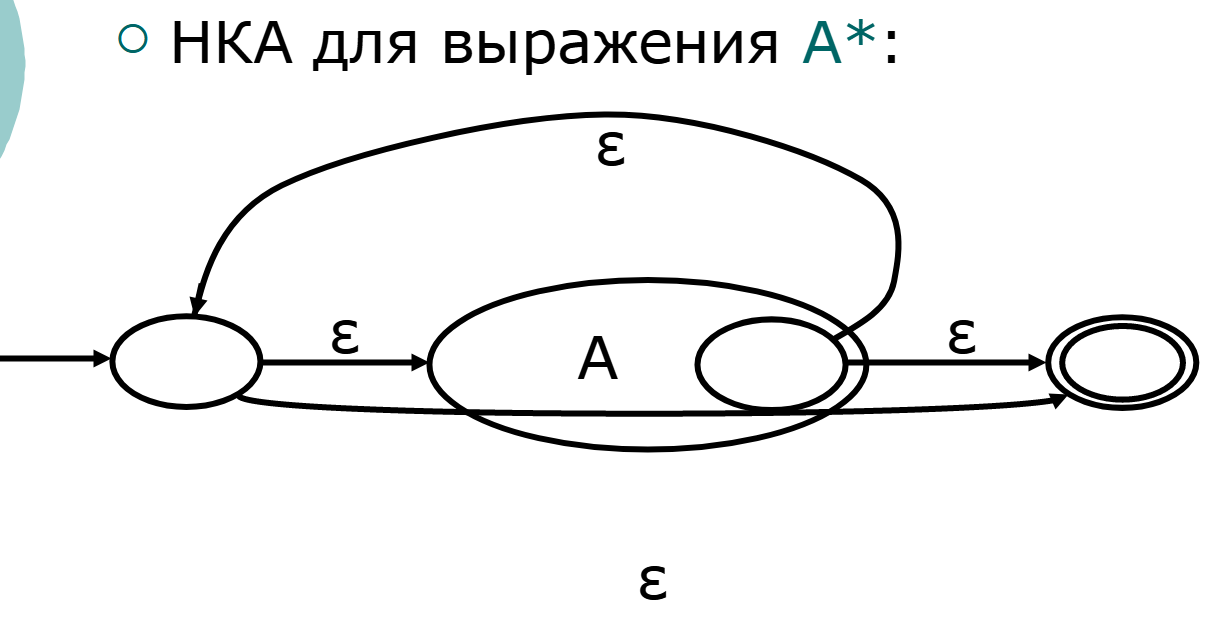


Простые регулярные выражения.



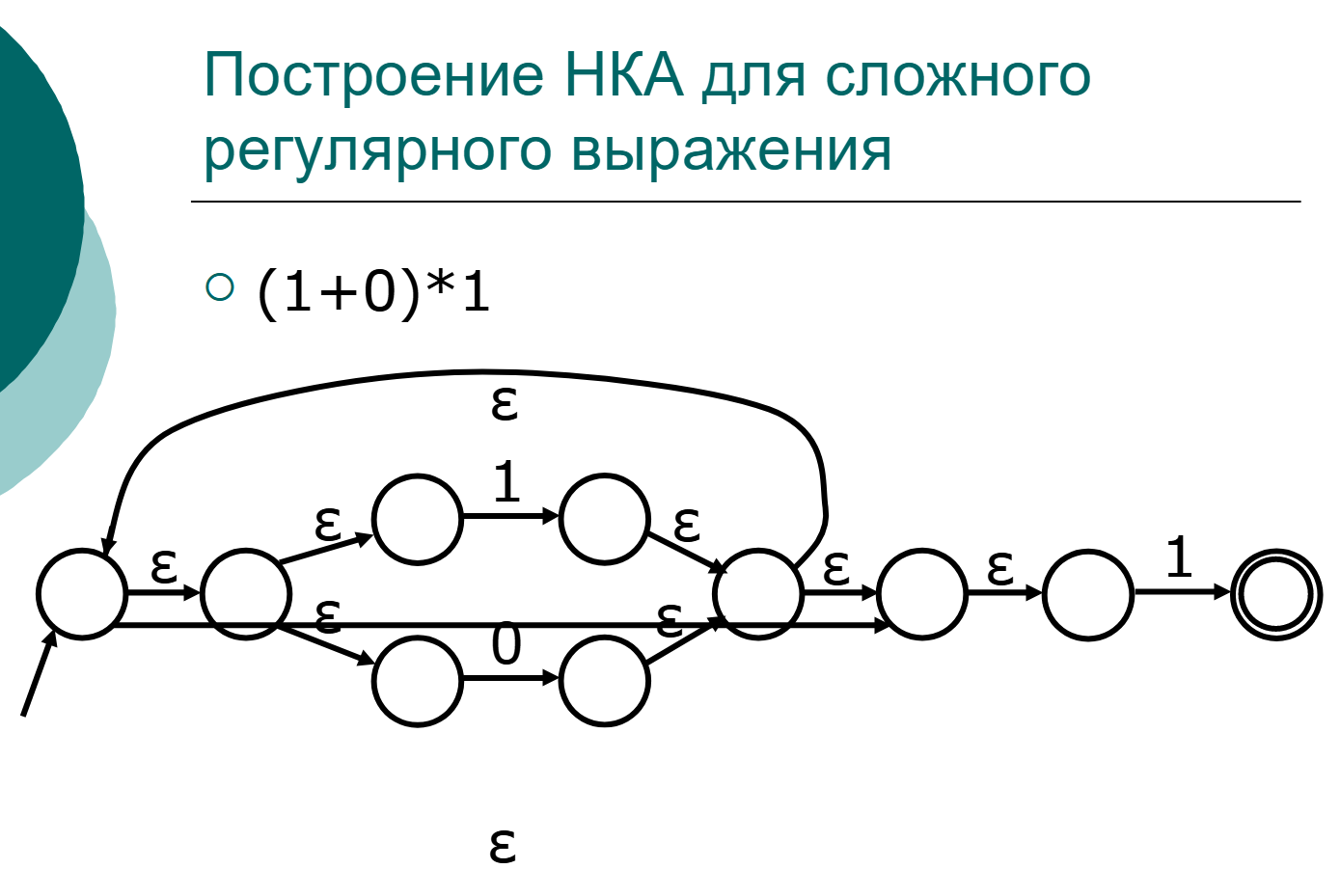
Составные регулярные выражения





Вот тут ебать стрелка не там нижняя не там проходит, надо её над нижним эпсилоном рисовать.

1. **Построение НКА для сложного регулярного выражения.**

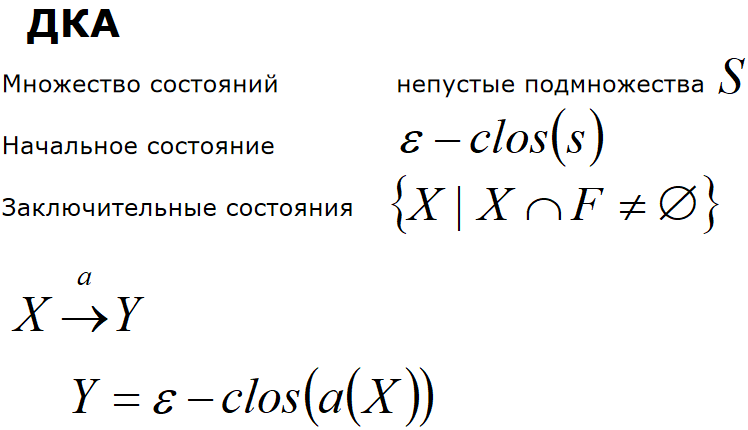
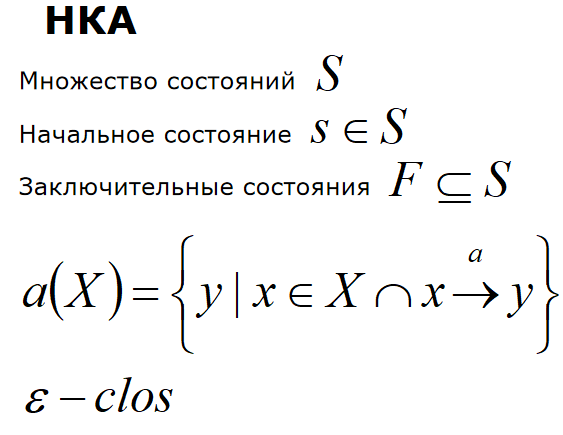
****

С нижней стрелкой тут такая же беда.

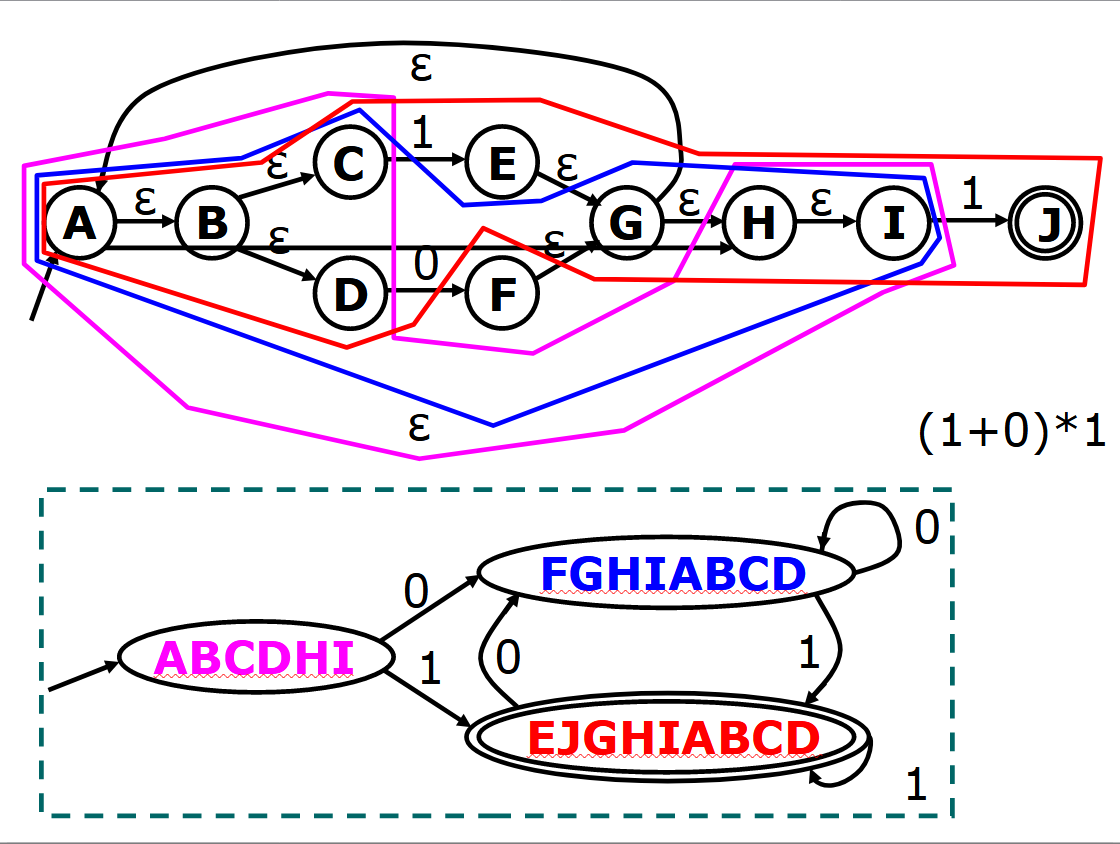
Можно ещё общую инфу про автоматы вкинуть из 11 вопроса

1. **Понятие ε-замыкания. Преобразование НКА в ДКА.**

ε-замыкание – множество состояний, достижимых из состояний исходного множества за 0 или более ε-переходов.



Каждое состояние ДКА соответствует множеству состояний НКА.



Вверху – НКА, внизу – соответствующий ему ДКА.

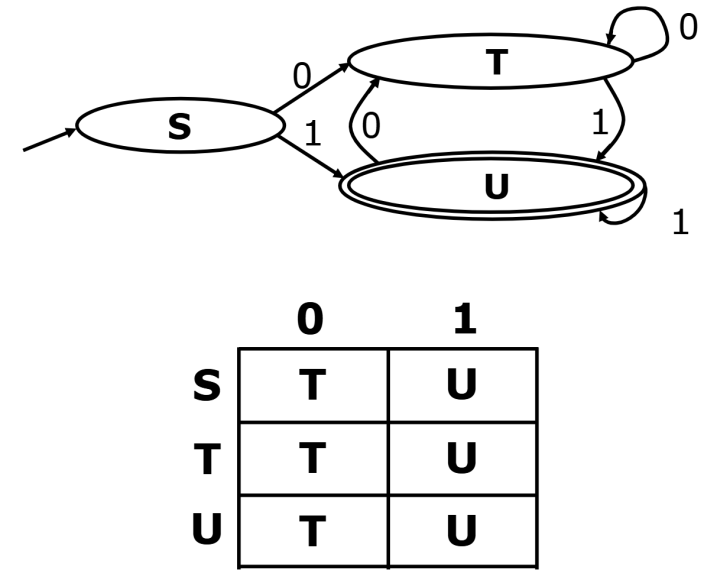
Пояснение по построению:

Берем начальный узел. Это у нас A. От этого начального узла смотрим, куда можем попасть по ε-переходам. Из A можем попасть в B C D H I. ABCDHI будет первым узлом ДКА. В этом узле есть вершины C и D. Их можно покинуть по сигналу либо 1, либо 0. Рассмотрим 0. Повторяем те же действия, что и для A. Из D можем попасть в FGHIABCD. Это следующий узел, в который переходим по сигналу 0. По сигналу 1 из C попадаем в EGHIJABCD. В этом узле у нас есть J, значит узел будет завершающим. Дальше по аналогии.

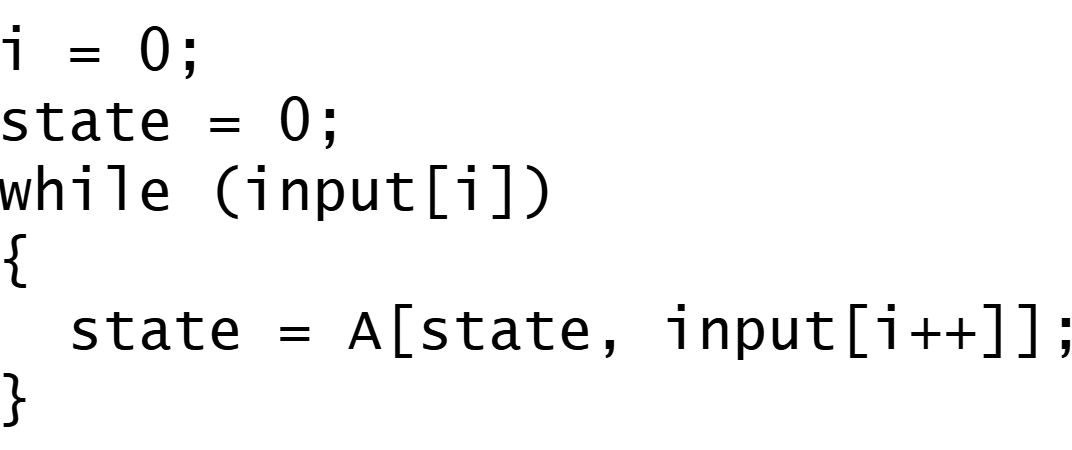
1. **Программная реализация конечных автоматов.**

ДКА можно реализовать на основе 2-мерной матрицы T с двумя измерениями: состояния и символ входного потока.

Каждому переходу  ставится в соответствие элемент T[i, k] = k;



Поясняю за таблицу: вверху – сигналы перехода, слева – узлы ДКА. Из вершины S по сигналу 0 автомат перейдет в состояние T, а по сигналу 1 перейдет в состояние U. Остальное по аналогии.



1. **Иерархия Хомского. Понятие грамматики. Проблема использования регулярных выражений в синтаксическом анализе.**

Иерархия Хомского – классификация формальных языков и грамматик, согласно которой они делятся на 4 типа по их условной сложности

1. Неограниченные
2. Контекстнозависимые
3. Контекстнонезависимые
4. Регулярные

Регулярные языки самые ограниченные формальные языки, много применений.

Грамматика – набор правил подстановки(продукции), задающих некоторый формальный язык.

Формальный язык – множество строк, удовлетворяющих некоторой грамматике

Проблема использования регулярных выражений в синтаксическом анализе хз какая

1. **Синтаксический анализ. Задачи синтаксического анализа. Понятие дерева разбора.**

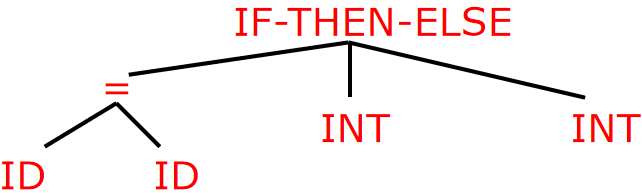
Синтаксический анализ — этап трансляции, задача которого состоит в построении древовидного представления анализируемого текста(например, исходного кода программы). Как правило, на вход синтаксическому анализатору(парсеру) подаются токены, получаемые в результате обработки анализируемого текста лексическим анализатором.



Язык: if x = y then 1 else 2 fi

Входные данные парсера: IF ID = ID THEN INT ELSE INT FI

Результат работы:





1. **Контекстно-свободные грамматики (КСГ). Понятия терминала, нетерминала, продукции. Связь КСГ с лексическим анализом.**

Не все последовательности токенов являются программами. Парсер должен различать корректные и некорректные последовательности токенов. Для этого нужны язык описания корректных последовательностей токенов и метод проверки корректности последовательности токенов.

ЯП имеют рекурсивную структуру. КСГ – удобный способ задания таких рекурсивных структур.

КСГ:

* Множество терминальных символов T
* множество нетерминальных символов N
* начальный символ s
* Множество продукций

Продукции можно рассматривать как правило подстановки.

1 Начать со строки, состоящей из единственного стартового символа S

2 Заменить любой нетерминальный символ Х на правую часть какой-либо из его продукций.

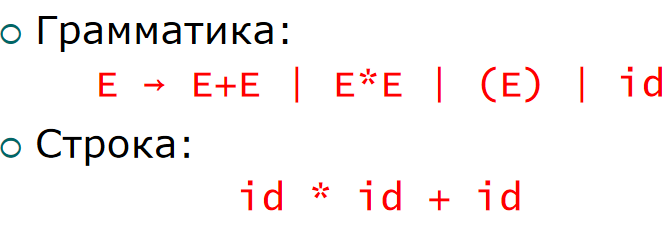
3 Повторять шаг 2, пока остаются нетерминальные символы.

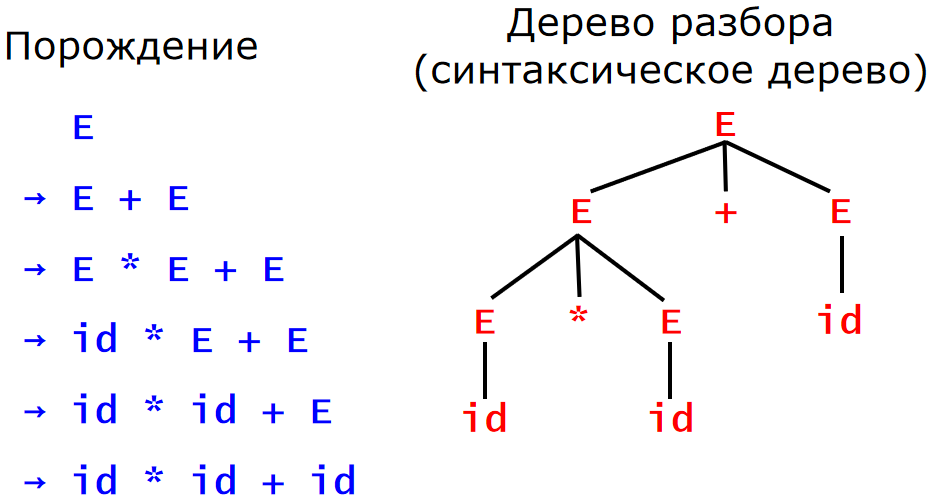
Терминальные символы – символы, для которых в грамматике нет правил подстановки. Появившись в строке, терминалы больше ничем не могут быть заменены. Терминалы – токены языка.

Идея КСГ – важный шаг, но нужна их реализация, необходимо обрабатывать ошибки во входных данных, необходимо строить дерево разбора.

1. **Порождения. Левое и правое порождения.**

Порождение – последовательность продукций. Порождение может быть представлено деревом, где стартовый символ – корень дерева, для продукции X 🡪 Y1..Yn к узлу X добавляются дочерние узлы Y1…Yn.





Синтаксическое дерево состоит из терминалов – листьев; нетерминалов – промежуточных узлов. Последовательный обход листьев позволяет восстановить исходную строку. (Пояснение: последовательный обход начнётся с самого левого id, потом \*, потом id, потом +, потом опять id и нихуя ж себе мы получаем исходную строку).

В левом порождении на каждом шаге заменяется “самый левый” нетерминал. В правом порождении на каждом шаге производится замена “самого правого” нетерминала.

Левому и правому порождению соответствует одно и то же дерево разбора.

1. **Неоднозначность. Проблемы использования неоднозначных грамматик в синтаксическом анализе. Примеры языков программирования, имеющих неоднозначные грамматики.**

Неоднозначная грамматика – формальная грамматика, которая может породить некоторую строку более чем одним способом. Т.е. для одной строки есть более одного разбора, для строки существует более одного левого или правого порождения.

Неоднозначность это плохо:

* значение некоторых программ может быть неопределенно
* синтаксический анализ неоднозначной грамматики детерминированным анализатором затруднен.
* синтаксический анализ неоднозначной грамматики недетерминированным анализатором это медленно.

Грамматики некоторых ЯП неоднозначны. При разборе таких языков необходимо учитывать семантическую информацию для выбора правильного разбора.

Пример неоднозначности:

x \* y

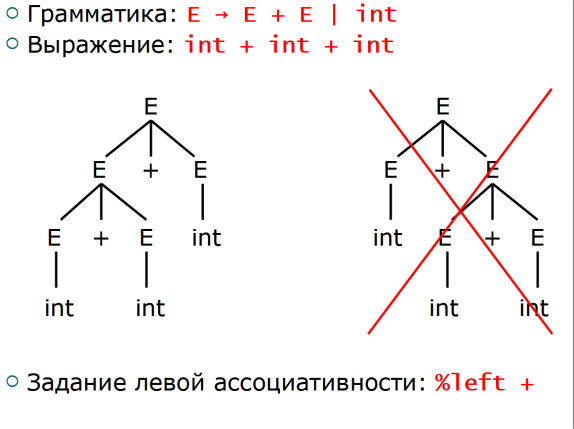
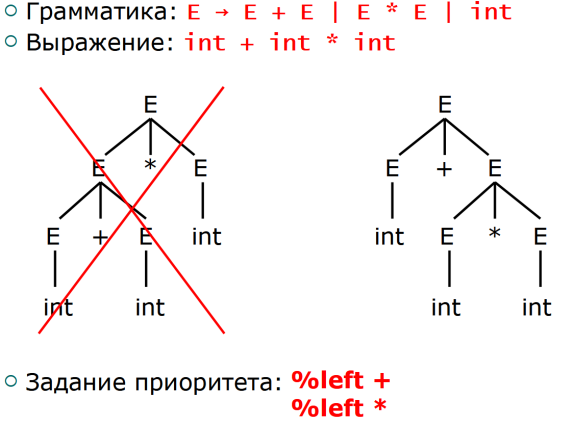
* Выражение, в котором x умножается на y, а результат игнорируется.
* Объявление идентификатора y с типом “указатель на x”

1. **Способы устранения неоднозначностей. Понятие ассоциативности, связь ассоциативности и приоритетов с неоднозначными грамматиками.**

Задача определения, является ли грамматика неоднозначной, алгоритмически неразрешима. Большинство конструкций, требующих синтаксического анализа, может быть распознано однозначными грамматиками. Некоторые неоднозначные грамматики могут быть преобразованы в однозначные.

Существует несколько способов устранения неоднозначности. Самый простой – переписать грамматику. Неоднозначные грамматики невозможно автоматически преобразовать в однозначные. Вместо переписывания грамматики может использоваться более естественная грамматика вместе с описаниями, позволяющими разрешить неоднозначность.

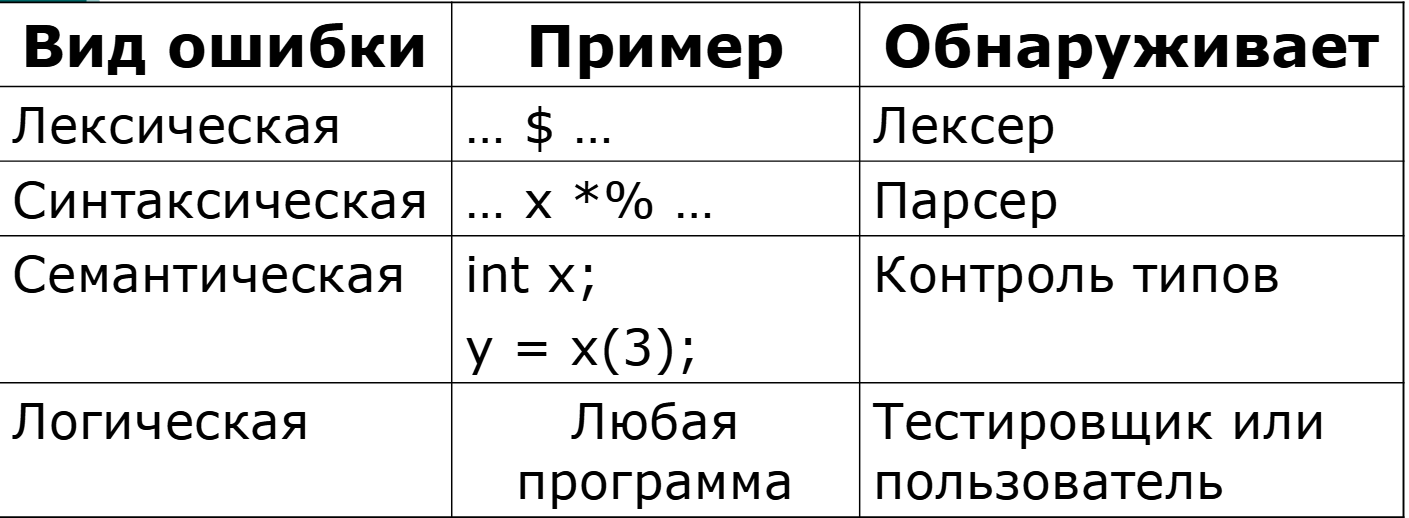
Ассоциативность – свойство операций, позволяющее восстановить последовательность их выполнения при отсутствии явных указаний на очередность, равном приоритете.

Уот так уот

1. **Обработка ошибок компилятором. Способы обработки ошибок, их преимущества и недостатки.**

Задачи компилятора – выявлять некорректные программы, транслировать корректные программы.



Способы обработки ошибок

* Восстановление в режиме паники

Самый простой и популярный метод. При обнаружении ошибки: пропускать токены, пока не встретиться токен определенного назначения, продолжить с этого места. Основан на поиске синхронизирующихся токенов – токены, завершающие операцию или выражение. (1 + + 2) + 3. В этом примере токены пропускаются до следующего числа 2. Могут быть пропущены дополнительные ошибки

- Восстановление на уровне фразы

Похож на восстановление в режиме паники. При обнаружении ошибки выполняется локальная коррекция входного потока. Пример: автоматическое закрытие круглых скобок перед ;

* Продукции ошибок

Известные распространенные синтаксические ошибки включаются в грамматику. Пример: для записи 5х вместо 5 \* х в грамматику включается продукция Е → .. | EE. Недостаток: грамматика становится сложнее.

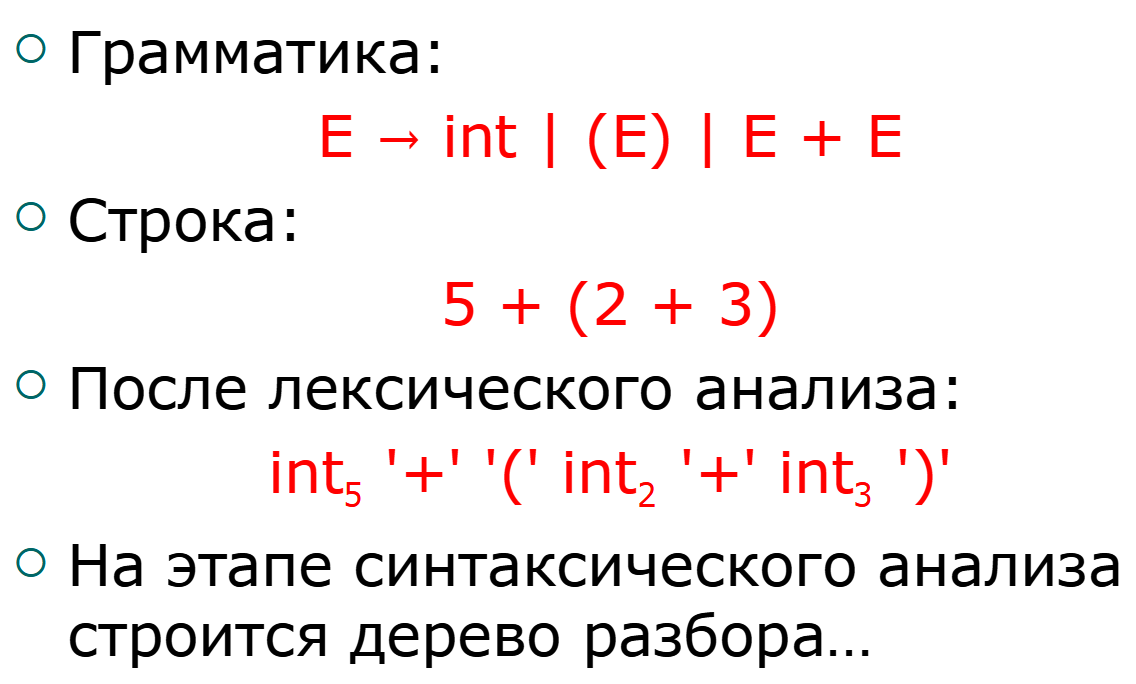
* Автоматическая коррекция ошибок

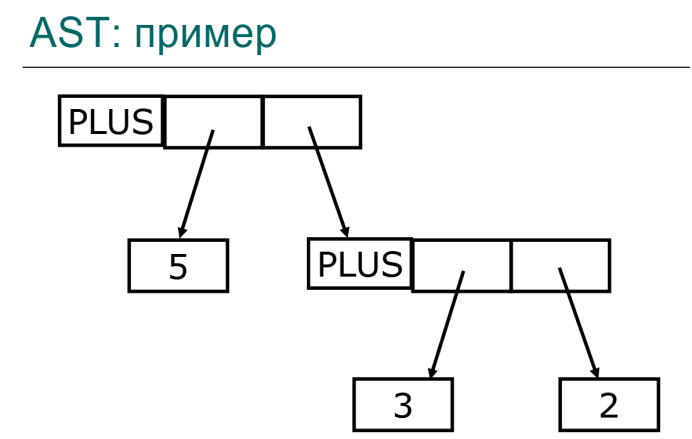
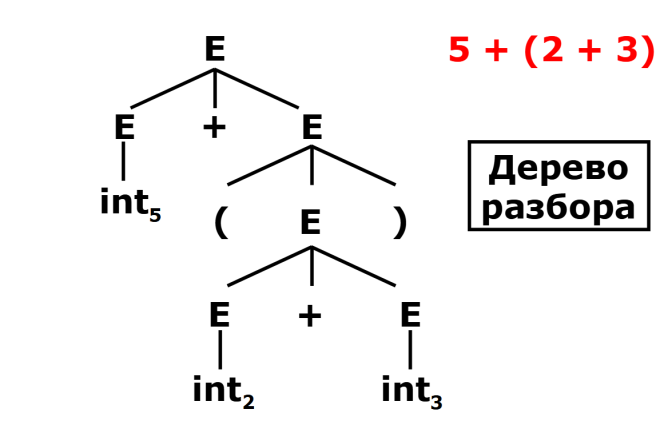
Идея заключается в нахождении (подборе) похожей (близкой) программы, не содержащей ошибки. Используется вставка/удаление токенов, полный перебор. Пример: PL/C — компилятор языка PL/I с поддержкой автокоррекции.

На данный момент используется короткий цикл компиляции/запуска, наблюдается тенденция к исправлению одной ошибки за каждый цикл, сложные механизмы обработки ошибок менее востребованы.

1. **Абстрактные синтаксические деревья (AST). Связь AST с деревьями разбора.**

Парсер отслеживает порождение последовательности токенов. Но остальным частям компилятора требуется структурное представление программы. Абстрактные деревья разбора (AST) похожи на деревья разбора, но не содержат некоторых деталей



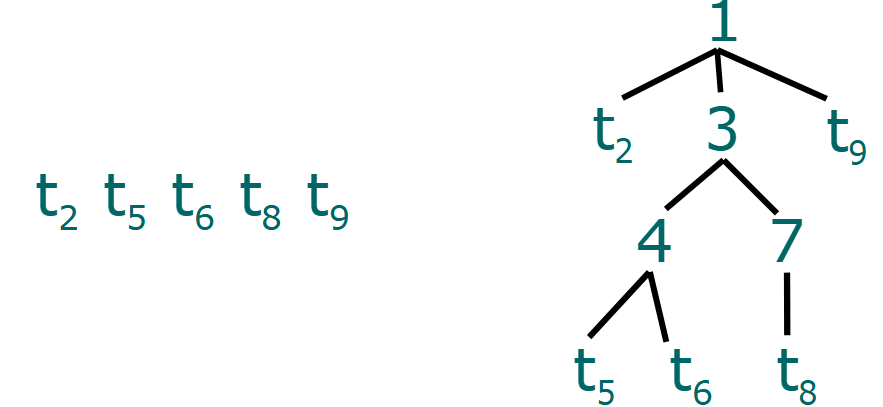


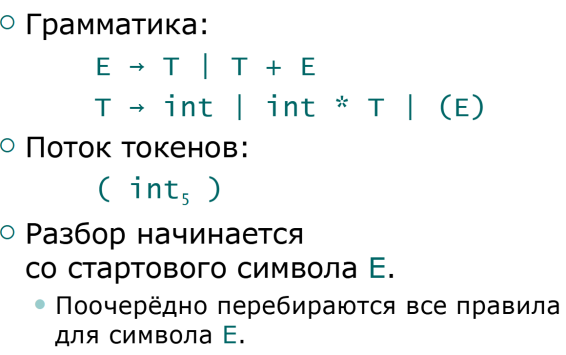
1. **Классификация методов синтаксического анализа. Метод рекурсивного спуска: теория.**

Методы синтаксического разбора: нисходящие (метод рекурсивного спуска, LL-анализ) и восходящие (LR-анализ, GLR-анализаторы).

Метод рекурсивного спуска:

Дерево разбора строится начиная с вершины (корня), слева направо. Терминалы появляются в порядке их появления в потоке токенов.





Тут советую открыть 13 презентацию и посмотреть как строилось дерево, но если кратко, то сначала E раскрывается в T, дальше T раскрывается в int. Продукция int нам не подходит, поэтому раскрываем T в int \* T. Это нам тоже не подходит, поэтому раскрываем в (E). И погнали дальше.

Метод рекурсивного спуска – простая и универсальная стратегия синтаксического разбора, необходимо избегать левой рекурсии, используется в разработке компиляторов.

1. **Метод рекурсивного спуска: программная реализация.**

TOKEN – перечисляемый тип. Значения: INT, OPEN, CLOSE, PLUS, TIMES,…

next – глобальная переменная, указывающая на следующий токен из входной последовательности.

Функции, проверяющие соответствие входной последовательности:

* Токену-терминалу

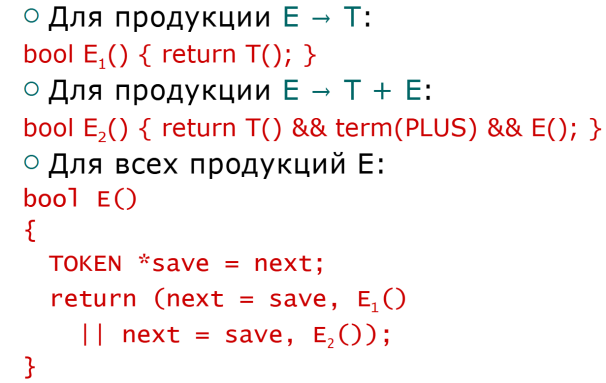
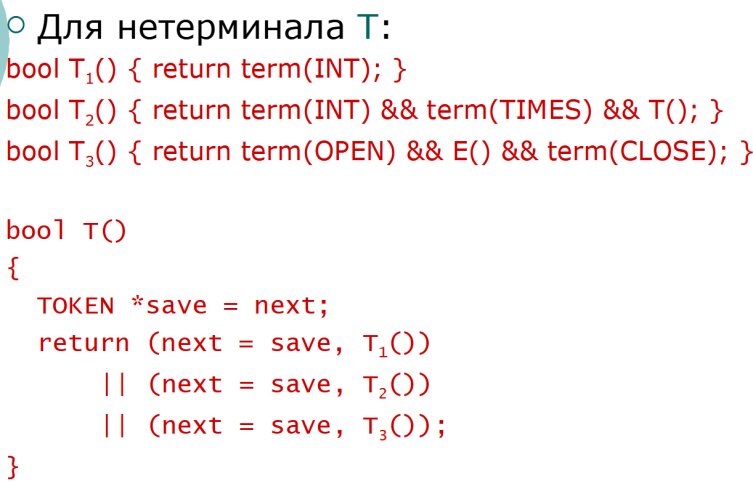
bool term(TOKEN tok) {return \*next++ == tok}

* n-й продукции S

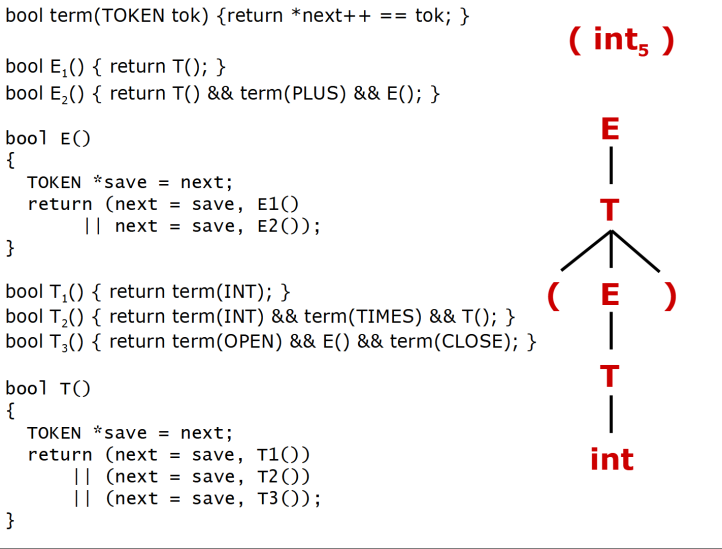
bool Sn() {…}

* всем продукциям S

bool S() {…}

Для запуска парсера необходимо инициализировать next (должна указывать на первый токен), вызвать E();



1. **Левая рекурсия, связанные с ней проблемы и способы её устранения.**

Леворекурсивная грамматика – грамматика, в которой есть такой нетерминал S, что S🡪Sa

bool S1() {return S() && term(a)}

bool S() {retrun S1()}

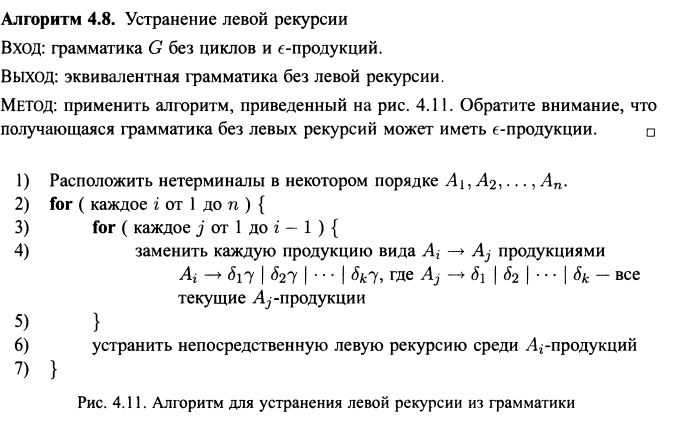
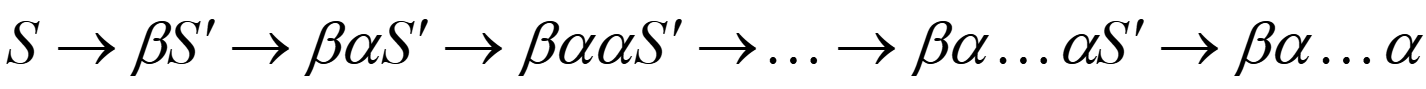
Вызов S() будет зацикливаться.

Метод рекурсивного спуска в таких случаях не работает.

Можно переписать грамматику с использованием правой рекурсии:

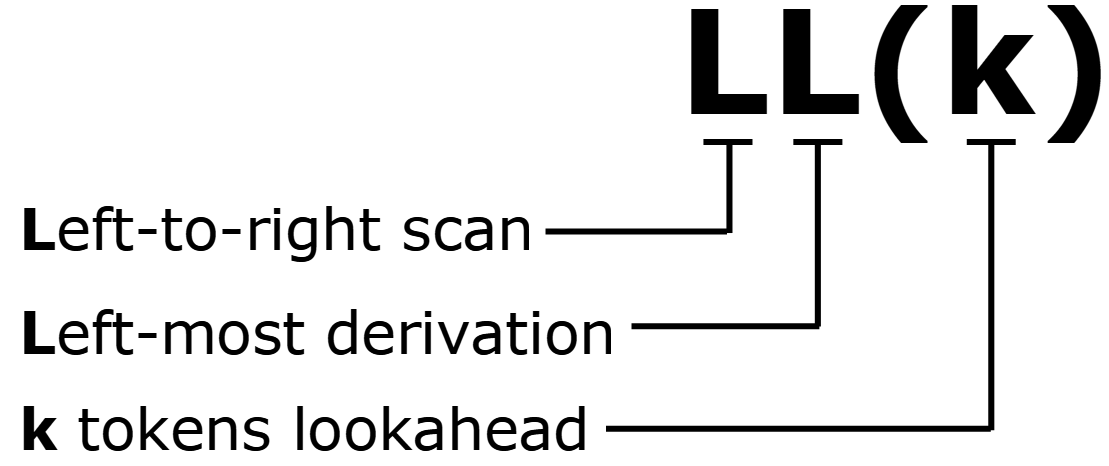
S → bS’

S’ → aS’|ε



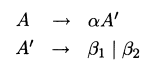
1. **Предиктивный анализ. Понятие LL(k)-грамматик. Левая факторизация грамматики.**

Похож на метод рекурсивного спуска, но может “предсказывать” какую продукцию лучше выбрать: заглядывая на несколько токенов вперед и не возвращаясь назад(backtracing). Предиктивные анализаторы работают с LL(k)-грамматиками.



LL(k) – класс грамматик, для которых можно построить предиктивный синтаксический анализатор, просматривающий k символов во входном потоке.

Левая факторизация представляет собой преобразование грамматики в пригодную для предиктивного синтаксического анализатора.

Пояснение: в общем случае, если  и входной поток начинается с α, то непонятно, какую продукцию использовать. Однако можно преобразовать продукцию в 

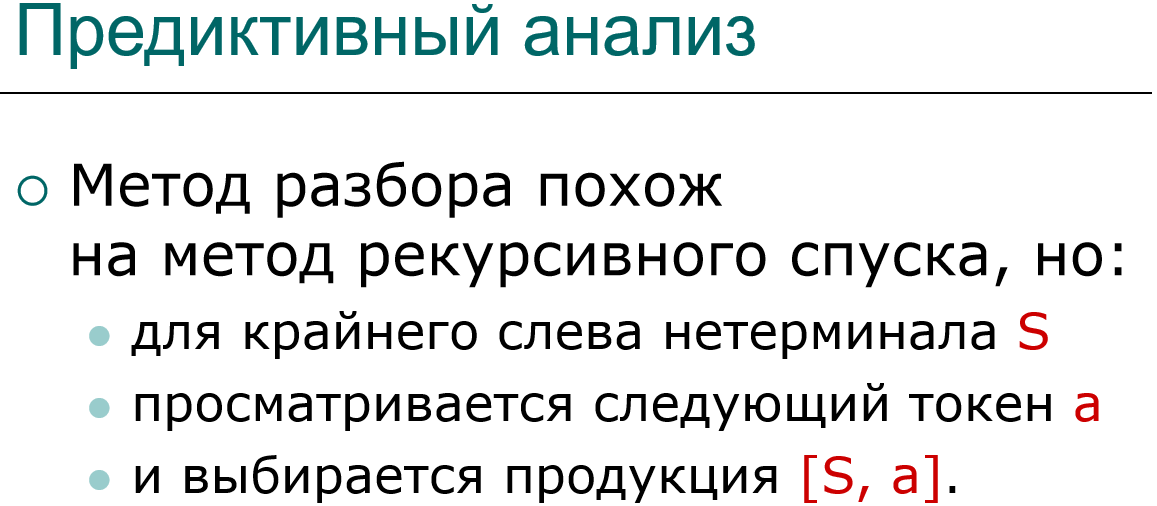
и все будет хорошо.

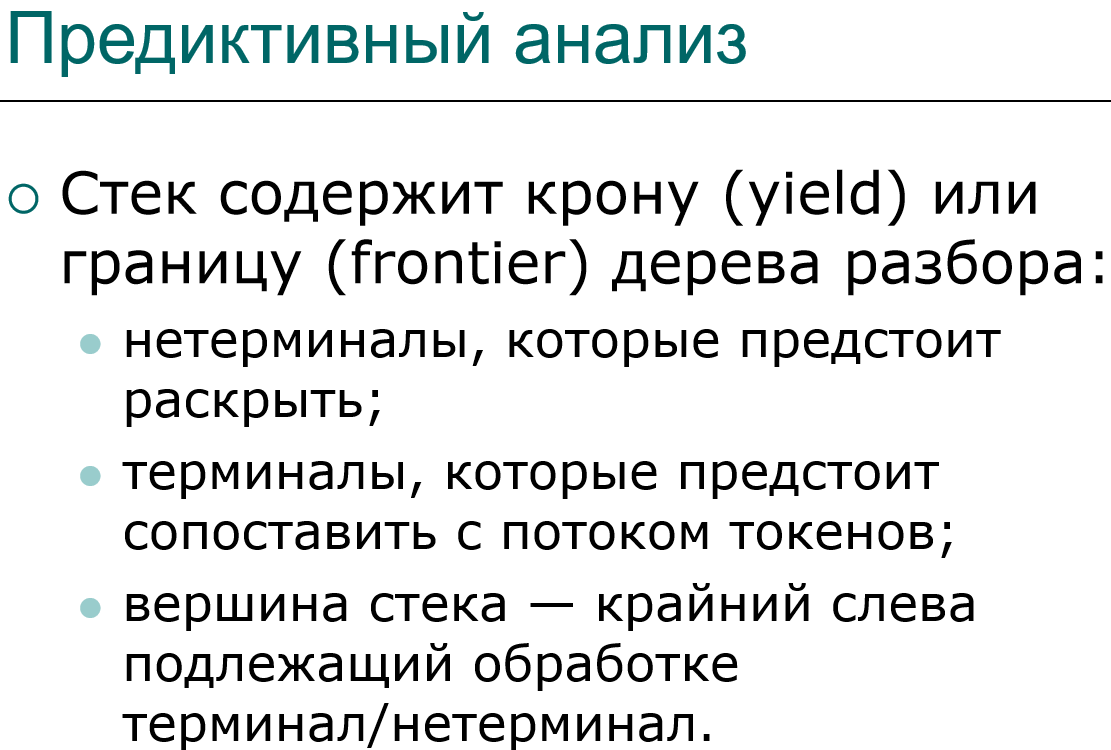
1. **LL(1)-разбор.**

LL(1): просмотр слева-направо, левое порождении, на каждом этапе существует не более 1 подходящей продукции.

Метод разбора похож на метод рекурсивного спуска, но для крайнего слева нетерминала S просматривается следующий токен a и выбирается продукция [S, a]



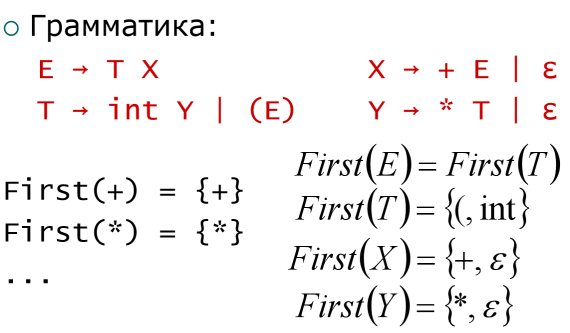


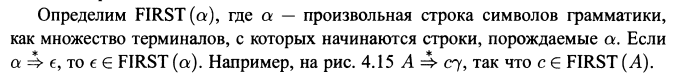


1. **First-множества в LL(1)-анализе.**



Это все символы, с которых могут начинаться всевозмжные выводы строки из нетерминалов терминалов

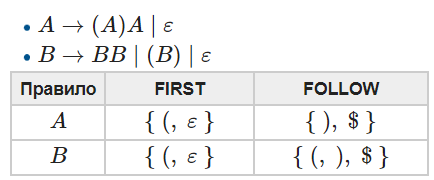


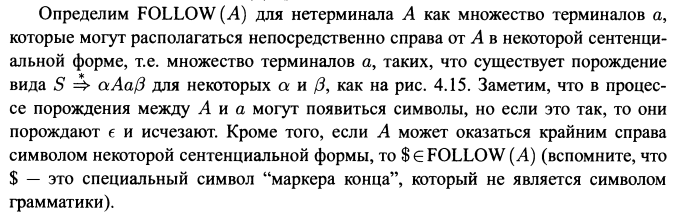
****

1. **Follow-множества в LL(1)-анализе.**

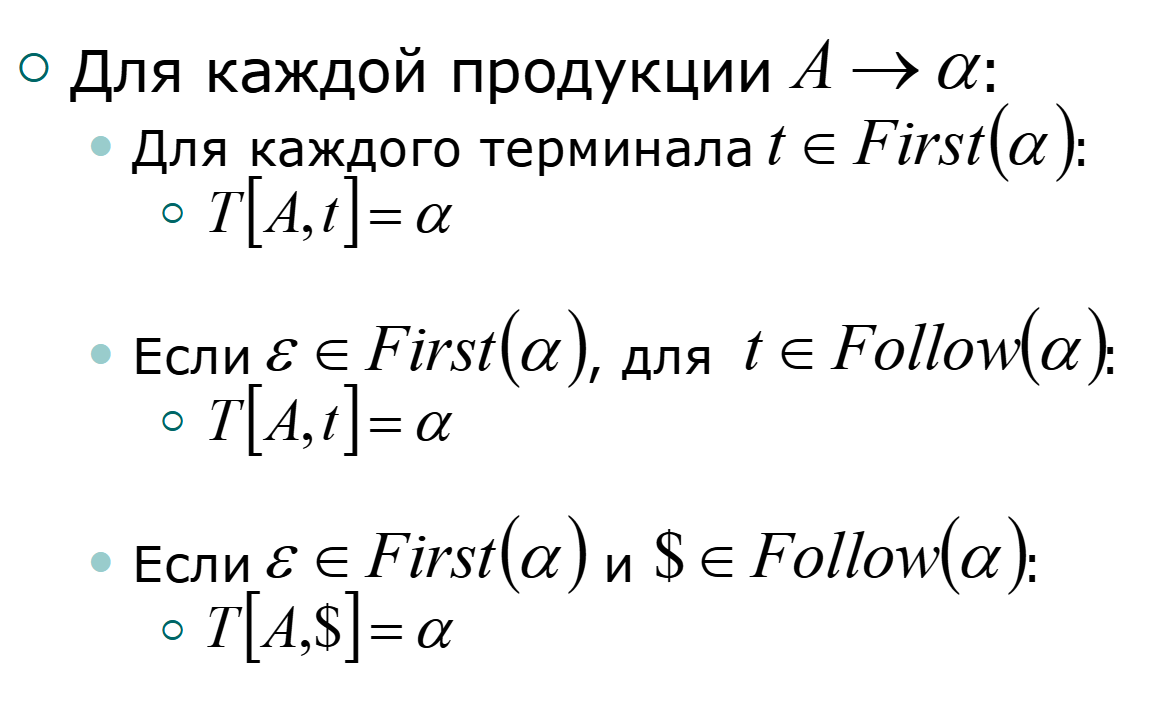


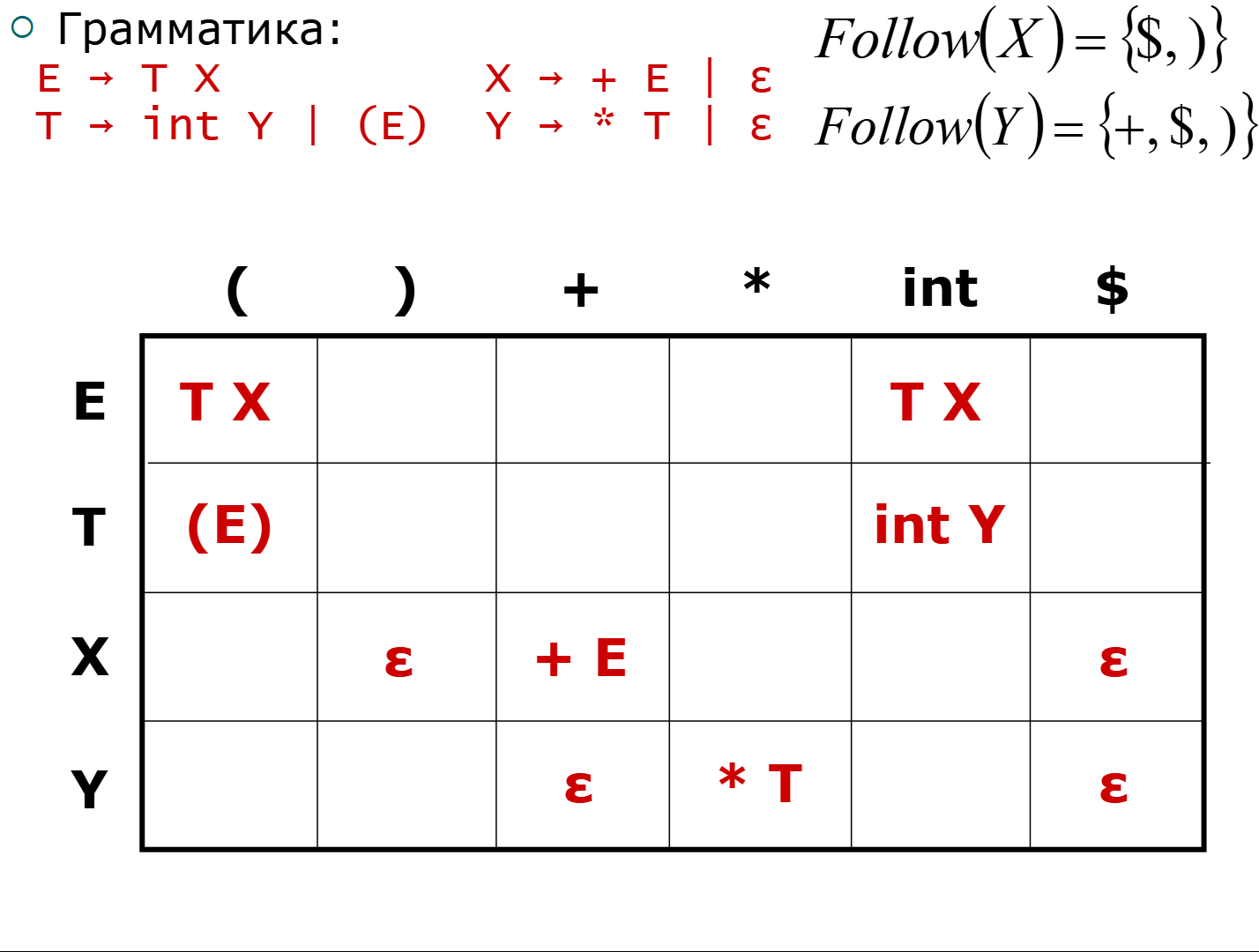
Это все возможные символы, которые могут встречаться после нетерминала Х

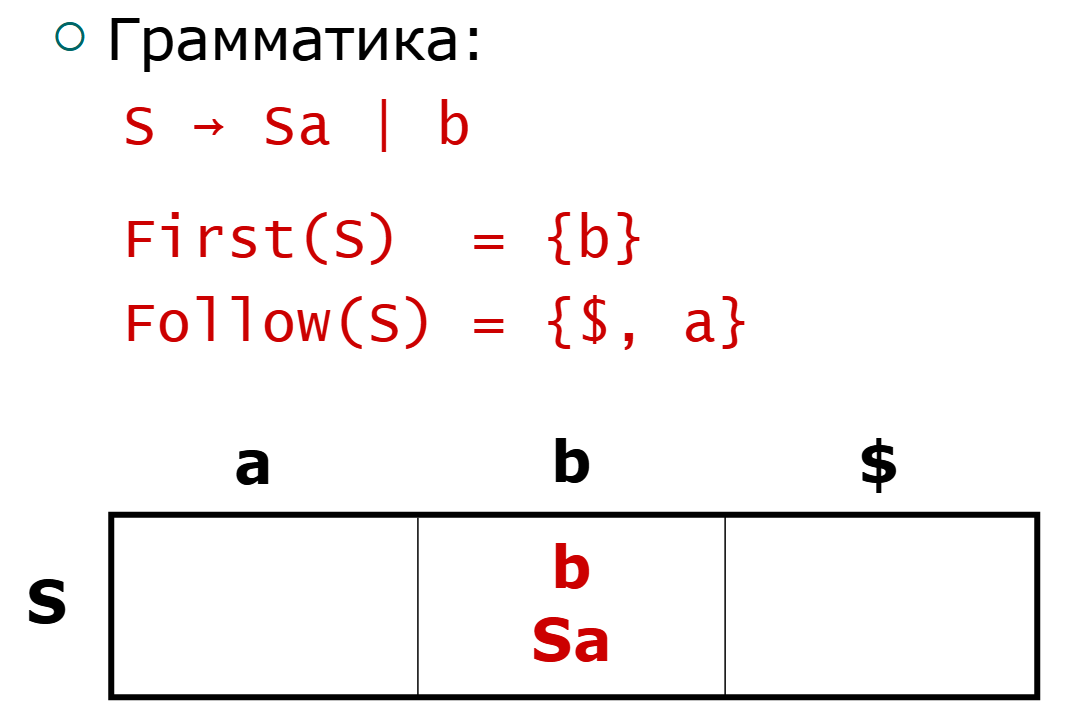


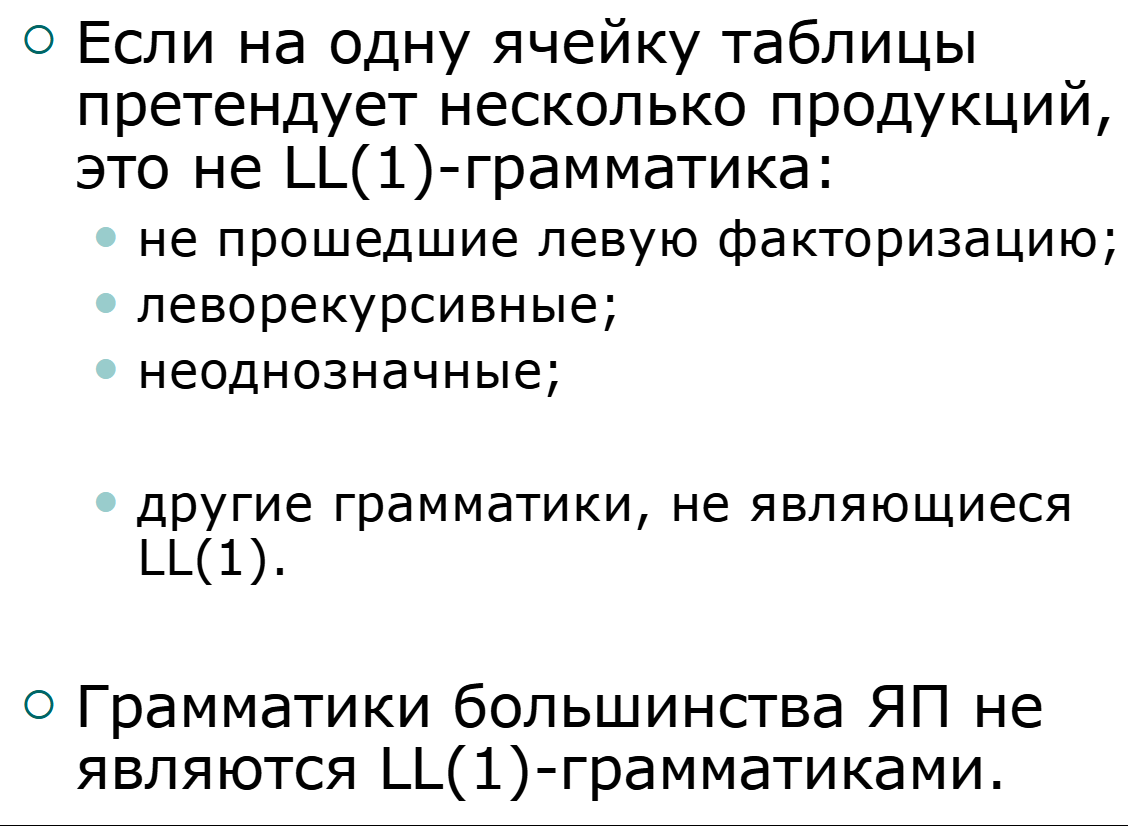
****

1. **Построение таблицы LL(1)-разбора. Выявление грамматик, не являющихся LL(1)-грамматиками.**

****

****

****

****

1. **Восходящий разбор: отличия от нисходящих методов, преимущества. Понятие свёртки.**

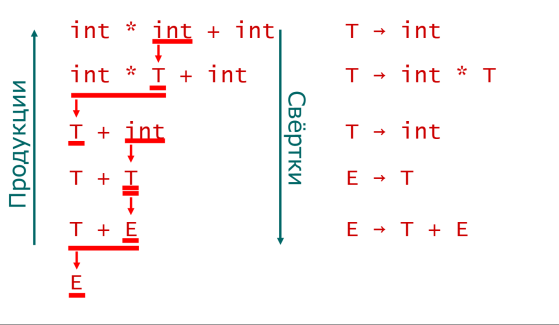
Восходящие: LR-анализ и GLR-анализаторы.

Восходящий разбор – более общий, чем (детерминированный) нисходящий разбор: столь же эффективный, основывается на идеях нисходящего разбора. Методы восходящего разбора более предпочтительны.

Для восходящего разбора не требуется левофакторизованная грамматика, что позволяет вернуться к более “естественной” грамматике.



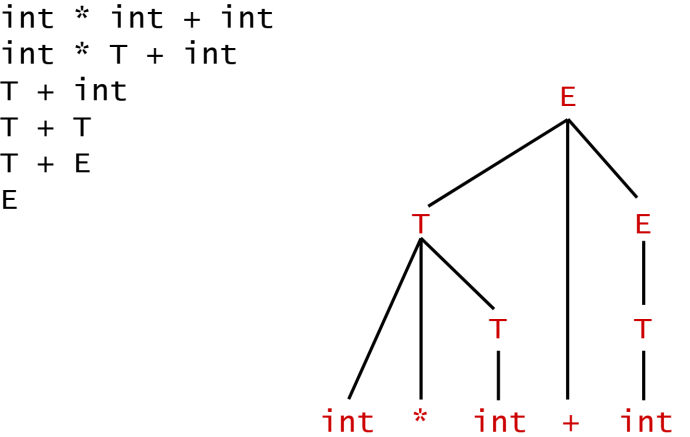
Восходящий разбор заключается в свёртке строки к стартовому символу с использованием продукции в обратном направлении.

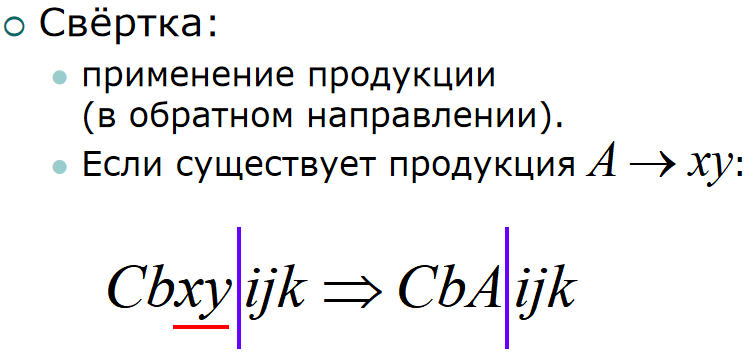


Это восходящей строки int \* int + int ок да?

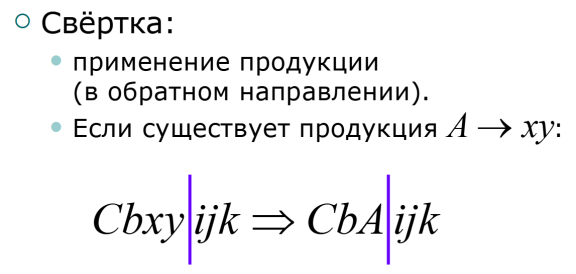
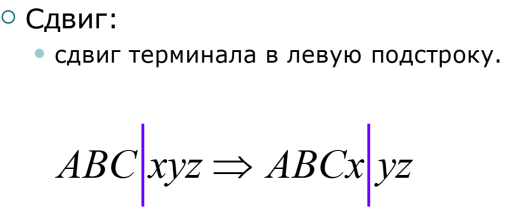
ИНТЕРСНЫЕ ФАКТЫ О ВОСХОДЯЩЕМ РАЗБОРЕ

1. Восходящий разбор строит правое порождение в обратном порядке.





1. В разборе методом “сдвиг-свёртка” дескрипторы появляются только в вершине стека.
2. Для любой грамматики множество активных префиксов является регулярным языком.
3. **Операции сдвига и свёртки в восходящем разборе. Виды конфликтов.**

****

Левая подстрока может быть реализована с использованием стека. Синяя черта – вершина стека. Сдвиг помещает терминал стек, свертка удаляет символы из стека, помещает нетерминал в стек.

Конфликты: если возможны сдвиг и свёртка – конфликт сдвиг-свёртка, если возможны несколько свёрток – конфликт свертка-свертка.

Выполнять свёртку нужно только в том случае, если это позволит получить стартовый символ. Дескриптор – свёртка, которая позволит последующим свёрткам дойти до стартового символа. Свёртки нужно выполнять только для дескрипторов. Основная проблема восходящего синтаксического анализа – распознавание дескрипторов.

1. **Семантический анализ. Задачи семантического анализа. Области видимости и способы их определения в различных языках.**

Лексический анализ – выявляет недопустимые токены, синтаксический анализ – выявляет недопустимые комбинации токенов, семантический анализ – отлавливает все остальные ошибки.

Некоторые ошибки не могут быть выявлены парсером, некоторые языковые конструкции не являются контекстно-свободными.

Проверки: все идентификаторы определены, соответствие типов, отношение наследования, классы объявляются только 1 раз, методы в пределах класса объявляются по одному разу, зарезервированные идентификаторы используются корректно и др.

Выявления соответствия между объявлениями и использованиями идентификаторов – важный шаг статистического анализа для большинства языков. Решение этой проблемы связано с понятием область видимости.

Статическое определение области видимости: область видимости идентификатора определяется в тексте программы. Динамическое определение области видимости: область видимости зависит от хода выполнения программы.

1. **Семантический анализ. Таблицы символов.**

Большая часть семантического анализа сводится к обходу AST сверху вниз: сначала: обработать узел n, рекурсия: обработать дочерние элементы n, после: завершить обработку узла n.

Для обработки AST на этапе семантического анализа необходимо иметь информацию об идентификаторах.

Таблица символов – структура данных, хранящая информацию об идентификаторах.

Операции: AddSymbol(x) – добавление символа, FindSymbol(x) – поиск символа, RemoveSymbol() – удаление символа.

Более сложная таблица символов: EnterScope() – начало ОВ, FindSymbol(x) – поиск символа x, AddSymbol(x) – добавить символ, CheckScope(x) – проверка принадлежности, ExitScope() – оконачение ОВ.

1. **Семантический анализ. Понятие типа, виды типизации.**

Типы данных: множество значений, множество операций.

В языке СИ: сложение указателя на функцию с целым числом – недопустимая операция, сложение целого числа с целым числом – допустимая операция.

В ассемблере (и машинном коде) обеим операциям соответствует одна и та же команда.

ЯП делятся на 3 группы: статические типизированные: все или почти все проверки соответствия типов происходят при компиляции программы; динамически типизированные: все или почти все проверки соответствия типов происходят во время выполнения; нетипизированные: машинные языки.

Статическая типизация: выявляет многие ошибки при компиляции, не требует дополнительных затрат на проверки во время выполнения.

Динамическая типизация: в меньшей степени ограничивает программиста, упрощает быстрое протипирование.

Большая часть кода написана на языках со статической типизацией, имеющих “лазейки”. Пример: небезопасное приведение типов.

1. **Оптимизация и генерация кода. Способы оптимизации. Проблемы оптимизации.**

Оптимизация –модификация программ с целью улучшения их характеристик без изменения функциональности (логики работы). Влияет на производительность(увеличивает), компактность.

Оптимизации: машинно-зависимые, машинно-независимые, локальные, внутрипроцедурные, межпроцедурные, внутримодульные, глобальные.

Способы оптимизации:

1. Сворачивание констант
2. Распространение констант
3. Распространение копий
4. Удаление недоступного кода
5. Устранение общих подвыражений

Низкоуровневые оптимизации:

1. Разворачивание циклов
2. Векторизация кода
3. Подстановка процедур(inline)
4. Изменение порядка следования переменных в памяти

Генерация кода первоначально выполняется только ОС. При запуске программы ОС выделяет место в памяти для программы, в отведенное место загружается код программы, ОС передаёт управление на точку входа программы.

Компилятор генерирует код, определяет способ использования памяти данных.

Способы организации компиляторов: генерация кода сразу на целевом языке, генерация кода в 2 этапа с использование промежуточного языка, JIT-компиляция.