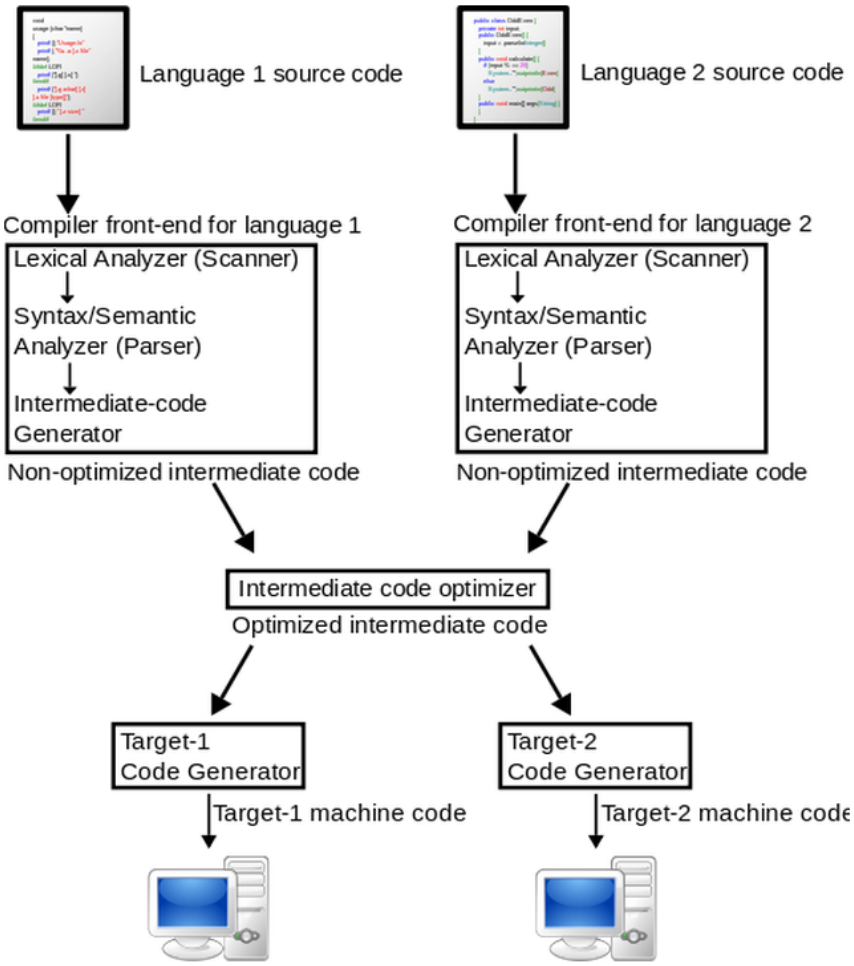
Стадии компиляции

Как мы уже с вами обсуждали ранее, файлы с исходным кодом на языке C/C++ сначала подаются по одному на выход компилятору (предварительно проходя через препроцессор), в результате чего по каждому из них получается объектный файл с бинарным кодом. Далее эти файлы подаются на вход линковщику, который из них делает уже исполняемых бинарный файл.

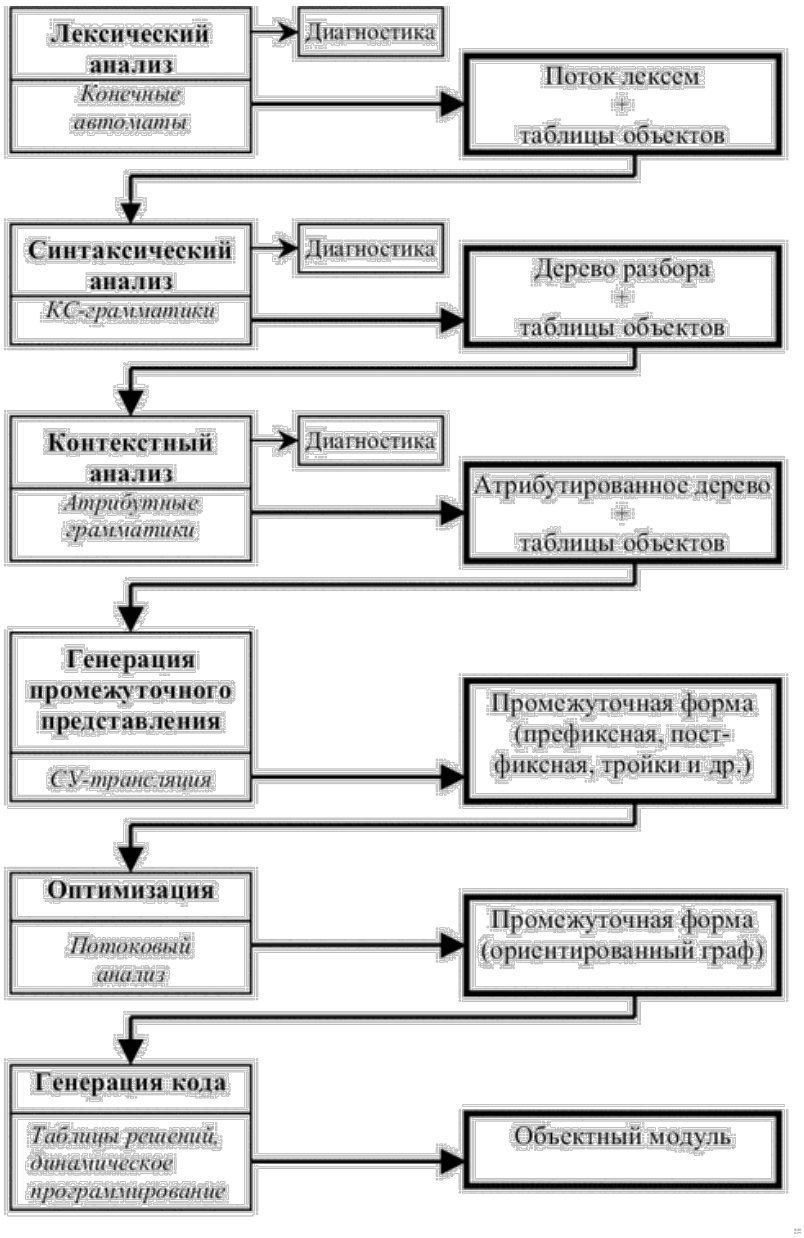
Это и следующее занятие мы потратим на то, чтобы разобраться, что происходит внутри компилятора. Там происходит довольно дофига всего, поэтому мы ограничимся только несколькими первыми стадиями. Особый интерес представляет то, что то, о чем мы с вами будем говорить на этих двух парах, может пригодиться вам, даже если вы будете весьма далеки от разработки компиляторов. На последующих курсах вам эти вопросы будут раскрывать гораздо более подробно, у нас же тут, как водится, что-то типа введения.

Общий принцип работы компилятора изображен на следующем рисунке.



Более подробно, традиционно в процессе компиляции выделяют следующие фазы (на рисунке также отображены артефакты, создаваемые на каждом этапе и

передаваемые между ними):



На фазе лексического анализа входная программа, представляющая собой поток символов, разбивается на лексемы-слова в соответствии с определениями языка. Основными формализмами, лежащим в основе реализации лексических анализаторов, являются конечные автоматы и регулярные выражения. Лексический анализатор может работать в двух основных режимах: либо как подпрограмма, вызываемая синтаксическим анализатором для получения очередной лексемы, либо как полный проход, результатом которого является файл лексем.

В процессе выделения лексем лексический анализатор может как самостоятельно строить таблицы объектов (идентификаторов, строк, чисел и т.д.), так и выдавать значения для каждой лексемы при очередном к нему обращении. В этом случае

таблицы объектов строятся в последующих фазах (например, в процессе синтаксического анализа). На этапе лексического анализа обнаруживаются некоторые (простейшие) ошибки (недопустимые символы, неправильная запись чисел, идентификаторов и др.).

Основная задача синтаксического анализа — разбор структуры программы. Результатом синтаксического анализа является синтаксическое дерево со ссылками на таблицы объектов. В процессе синтаксического анализа также обнаруживаются ошибки, связанные со структурой программы.

На этапе контекстного (семантического) анализа выявляются зависимости между частями программы, которые не могут быть описаны контекстно-свободным синтаксисом (про грамматики мы будем говорить на следующем занятии). Это в основном связи «описание-использование», в частности, анализ типов объектов, анализ областей видимости, соответствие параметров, метки и другие.

Затем программа может быть переведена во внутреннее представление. Это делается для целей оптимизации и/или удобства генерации кода. Еще одной целью преобразования программы во внутреннее представление является желание иметь переносимый компилятор. Тогда только последняя фаза (генерация кода) является машинно-зависимой. В качестве внутреннего представления может использоваться префиксная или постфиксная запись, ориентированный граф и другие.

Фаз оптимизации может быть несколько. Оптимизации обычно делят на машинно-зависимые и машинно-независимые, локальные и глобальные. Часть машинно-зависимой оптимизации выполняется на фазе генерации кода. Глобальная оптимизация пытается принять во внимание структуру всей программы, локальная — только небольших ее фрагментов. Глобальная оптимизация основывается на глобальном потоковом анализе, который выполняется на графе программы и представляет по существу преобразование этого графа. При этом могут учитываться такие свойства программы, как межпроцедурный анализ, межмодульный анализ, анализ областей жизни переменных и т.д.

Наконец, генерация кода — последняя фаза трансляции. Результатом ее является либо ассемблерный модуль, либо объектный (или загрузочный) модуль. В процессе генерации кода могут выполняться некоторые локальные оптимизации, такие как распределение регистров, выбор длинных или коротких переходов, учет стоимости команд при выборе конкретной последовательности команд.

Сегодня мы будем говорить подробно про лексический анализ, а на следующей паре — про синтаксический. Программисты много работают с языками, и формальная теория языков крайне полезна (по крайней мере, ее применения на практике), хотя бы даже для того, чтобы понимать, как оно внутри работает и писать код более осознанно. Ну и не говоря о создании собственных небольших языков (DSL), которые часто бывают крайне полезны и которые сейчас довольно сильно популярны.

Лексический анализ

Итак, лексический анализатор (lexer) преобразует поток символов в поток токенов, которые передаются на вход синтаксическому анализатору. Например, строка “position := initial + rate * 60” после обработки лексером будет преобразовано в следующий поток токенов:

1. Идентификатор position
2. Символ присвоения

3. Идентификатор initial
4. Знак сложения
5. Идентификатор rate
6. Знак умножения
7. Число 60

"идентификатор", "число", "знак умножения" обычно называют токенами, а "position", "rate", "*", "60" и т.д. — лексемами. Т.е. лексема — это что-то вроде значения токена, а токен — что-то вроде типа лексемы.

Делается это все затем, чтобы избавить синтаксический анализатор от ненужных деталей, связанных с лексическим анализом. Лексические анализаторы могут применяться не только в синтаксическом анализе, но и в других задачах — например, реализация scanf, поиск по шаблону. Основные функции лексического анализатора:

- удаление пробелов и комментариев
- распознавание констант - 31 + 28 + 59 преобразуется в последовательность <num, 31> <+, > <num, 28> <+, >, <num, 59>
- распознавание идентификаторов и ключевых слов: count = count + increment преобразуется в id = id + id. Здесь в качестве значения может использоваться указатель на запись в таблице символов. Идентификаторы часто выглядят как ключевые слова, поэтому лексер должен возвращать идентификатор только тогда, когда он не является ключевым словом.

Чтобы создавать лексические анализаторы, для начала нужно научиться задавать токены. Каждый токен описывается неким шаблоном. Например, шаблон для идентификатора в C++ неформально может быть описан как буква, за которой может идти несколько букв или цифр. Каждому шаблону соответствует множество строк, для формального задания таких множеств могут использоваться так называемые регулярные выражения.

Для того, чтобы определить, что такое регулярные выражения и какое отношение они имеют к лексическому анализу, нам потребуются следующие определения:

- ***алфавит*** — любое конечное множество символов.
- ***строка*** над некоторым алфавитом — конечная последовательность символов, взятых из алфавита. Длина строки s (обозначается как |s|) — количество символов в строке.
- ε — пустая строка (строка нулевой длины)
- ***конкатенация строк*** x и y (записывается как xy) — строка, сформированная путём дописывания y к x.
- ***язык*** — множество строк над некоторым алфавитом. Например, пустой язык {}, множество всех корректных программ на языке C++, все грамматически корректные предложения русского языка.

Над языками вводятся следующие операции:

Операция	Определение
Объединение (union) L и M — $L \cup M$	$L \cup M = \{s \mid s \in L$ или $s \in M\}$
Конкатенация (concatenation) L и M — LM	$LM = \{st \mid s \in L$ и $t \in M\}$
Замыкание Клини (Kleene closure); или *итерация* L — L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* обозначает "нуль или более конкатенаций L"
Позитивное замыкание (positive closure) L — L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ обозначает "одна или более конкатенаций L"

Примеры:
пусть L = {A,B,...,Z,a,b,...,z}, D = {0,1,2...9}

1. $L \cup D$ представляет собой множество букв и цифр.
2. LD — множество строк, состоящих из буквы, за которой следует цифра.
3. L^4 — множество всех четырехбуквенных строк.
4. L^* — множество всех строк из букв, включая пустую строку ε.
5. $L(L \cup D)^*$ — множество всех строк из букв и цифр, начинающихся с буквы.
6. D^+ — множество всех строк из одной или нескольких цифр.

Регулярные выражения

Для задания токенов в лексических анализаторах обычно применяют регулярные выражения. Например, "letter (letter | digit) *" — идентификаторы в Pascal. Каждое регулярное выражение задаёт язык L(r). Формально регулярные выражения над алфавитом Σ определяется следующим образом:

1. ε представляет собой регулярное выражение, обозначающее {ε}, т.е. множество, содержащее пустую строку.
2. Если a является символом из Σ, то a — регулярное выражение, обозначающее {a}.
3. Пусть r и s — регулярные выражения, обозначающие языки L(r) и L(s). Тогда:
    a. (r) | (s) представляет собой регулярное выражение, обозначающее объединение языков L(r) и L(s)
    b. (r)(s) — регулярное выражение, обозначающее L(r)L(s) (конкатенацию)
    c. (r)* обозначает язык (L(r))*
    d. (r) обозначает язык L(r), т.е. регулярное выражение можно помещать в дополнительные скобки.

Для удобства записи часто используют сокращения r+ (один или несколько экземпляров), r? (r | Ɛ) и классы символов ([abc] <=> a | b | c, [a..z] <=> a | b | ... | z)

Приоритеты операций:

1. * имеет наибольший приоритет
2. конкатенация имеет второй по значимости приоритет
3. | (объединение) имеет самый низший приоритет

Язык, задаваемый регулярным выражением, называется регулярным множеством. Регулярные выражения не могут описать сбалансированные или вложенные конструкции, например, вложенные строки. {wcw | w - строка из a и b} тоже не может быть описано регулярным выражением (и даже КС-грамматикой, забегая вперед).

Примеры: пусть Σ — {a, b}.
Регулярное выражение a | b обозначает множество {a, b}.
Выражение (a | b)(a | b) — {aa, ab, ba, bb}.
Другое РВ для того же множества — aa | ab | ba | bb.
a* — {Ɛ, a, aa, aaa, ...}.

Для удобства при задании токенов используются регулярные определения, например:
letter -> A | B | ... | Z | a | b | ... | z
digit -> 0 | 1 | ... | 9
id -> letter (letter | digit) *

Распознавание

Теперь можно научиться токены распознавать. Рассмотрим простой язык, в котором есть конструкции if вида
if <expr> then <stmt> | if <expr> then <stmt> else <stmt> | e
, где expr -> <term> <relop> <term> | <term>
, a term -> <id> | <num>.

Токены в таком языке будут задаваться такими регулярными определениями:
if -> if
then -> then
else -> else
relop -> < | <= | = | <> | > | >=
id -> letter (letter | digit)*
num -> digit+ (. digit+)? (E(+ | -)? digit+)?
Нам ещё нужно будет уметь выкидывать пробелы:
delim -> blank | tab | newline
ws -> delim+

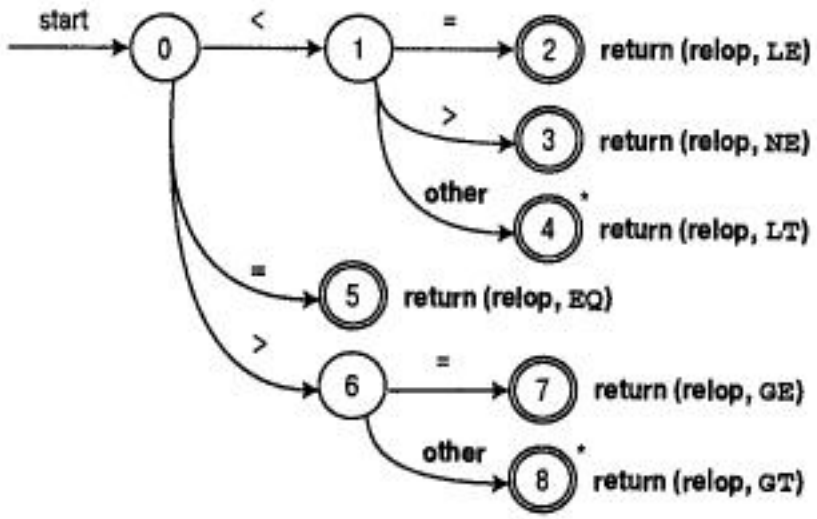
Для визуализации действий лексического анализатора при разборе входного потока используются диаграммы переходов. В узлах таких диаграмм находятся состояния лексического анализатора, дуги помечены символами, которые могут появиться во входном потоке. Например, для разбора токенов >= и > может

получиться такая диаграмма:

```
--start--> (0) -->--> (6) --=--> ((7))
                   \--other-->((8))*
```

(0) - начальное состояние, ((7)) и ((8)) — заключительные (или допускающие). * у ((8)) означает, что считанный символ надо вернуть во входной поток.

Лексический анализатор может иметь целый набор таких диаграмм, тогда поиск нужного токена будет просто последовательностью попыток применения диаграмм ко входной строке, если попытка неудачна, мы должны возвратить во входной поток все символы и попробовать следующую диаграмму. Если следующей нет — ошибка лексического анализа. Пример — диаграмма переходов для relop:



Как по такой диаграмме написать код:

Каждое состояние даёт часть кода. Если из состояния выходят дуги, то его код считывает очередной символ и выбирает дугу следования (если это возможно). Если существует дуга, помеченная считанным символом (или классом, к которому принадлежит считанный символ), управление передаётся коду состояния, на которое указывает данная дуга. Если такой дуги нет, это лексическая ошибка, и надо либо откатиться до начала лексемы и продолжить со следующей диаграммы, либо сообщить об ошибке, если другой диаграммы нет. Для перехода между состояниями используется switch по номерам состояний. Как-то так:

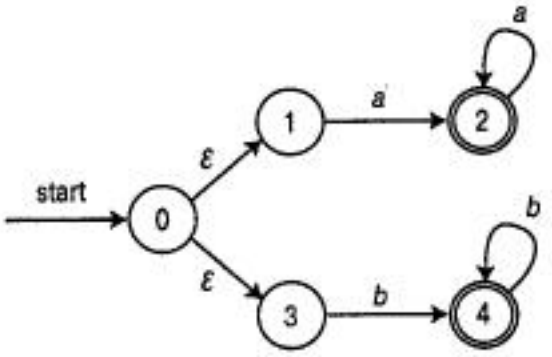
```
Token nextToken()
{
  while (true) {
    switch (state) {
      case 0:
         c = nextChar();
         if (c == blank || c == tab || c = newline) {
           state = 0;
           lexeme_beginning++;
         } else if (c == '<')
```

```
            state = 1
         else if (c == '=')
            state = 5;
         else if (c == '>')
            state = 6;
         else
            state = fail;
         break;
      case 1: ...
   }
}
```

Конечные автоматы

Обобщением диаграмм переходов являются конечные автоматы. Они бывают детерминированными и недетерминированными, оба вида способны распознавать регулярные выражения. Недетерминированный конечный автомат (НКА) — это (S, Σ, move, s0, F), где S — множество состояний, Σ — входной алфавит, move — функция переходов ((символ, состояние) -> множество состояний), s0 — начальное состояние, F — множество допускающих состояний. ДКА — это то же самое, только move возвращает не множество состояний, а одно состояние, и нет Ɛ-переходов (дуг, помеченных Ɛ). По любому НКА можно построить ДКА.

Пример: a a* | b b*



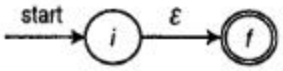
По регулярному выражению НКА строится так: пусть есть регулярное выражение r над алфавитом Σ, построим НКА N, допускающий L(r).

Метод (построение Томпсона)

Разберём r на составляющие подвыражения. Затем применяем описанные ниже правила (1) и (2) для символов алфавита или Ɛ из регулярного выражения. Если один символ встречается несколько раз, для каждого вхождения создаётся отдельный НКА. Затем, следуя синтаксической структуре регулярного выражения r, комбинируем получившиеся НКА с использованием правила (3), пока не будет получен НКА для

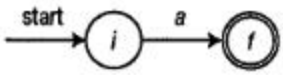
всего выражения.

1. Для ε строим НКА.



Здесь i — новое начальное состояние, а f — новое заключительное. Очевидно, что этот НКА распознает $\{\varepsilon\}$.

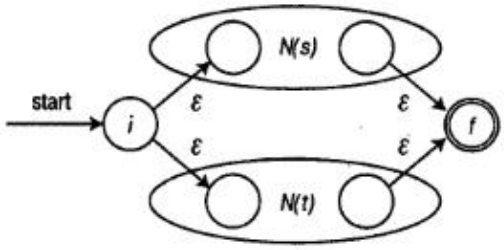
2. Для $a \in \Sigma$ строим НКА



Здесь также i — новое начальное состояние, а f — новое заключительное. Этот НКА распознает $\{a\}$.

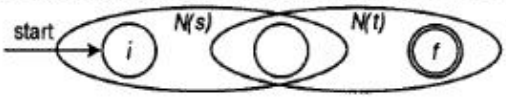
3. Пусть $N(s)$ и $N(t)$ представляют собой НКА для регулярных выражений s и t.

a) Для регулярного выражения $s \mid r$ строим следующий составной НКА $N(s \mid r)$:



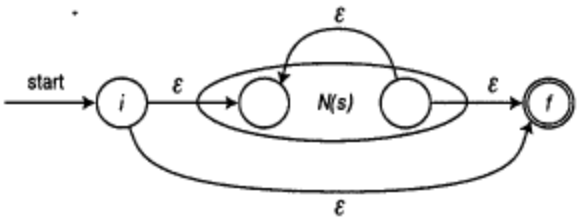
Здесь i — новое начальное состояние, а f — новое заключительное. Имеются переходы по ε от i к стартовым состояниям $N(s)$ и $N(t)$, а также переходы по ε из заключительных состояний $N(s)$ и $N(t)$ в новое заключительное состояние f. Начальные и заключительные состояния $N(s)$ и $N(t)$ не совпадают с начальными и заключительными состояниями $N(s \mid t)$. Заметим, что любой путь от i к f должен проходить либо только через $N(s)$, либо только через $N(t)$. В соответствии со сказанным составной НКА распознает $L(s) \cup L(t)$.

b) Для регулярного выражения st строится составной НКА $N(st)$:



Начальное состояние $N(s)$ становится начальным состоянием составного НКА, а заключительное состояние $N(t)$ — заключительным состоянием составного автомата. Заключительное состояние $N(s)$ сливается с начальным $N(t)$; таким образом, все переходы из начального состояния $N(t)$ преобразуются в переходы из заключительного состояния $N(s)$. Новое объединенное состояние теряет свой статус начального/заключительного в комбинированном автомате. Путь от i к f должен пройти вначале через $N(s)$, а затем — через $N(t)$, так что метки на этом пути представляют собой строку из $L(s)L(t)$. Поскольку нет дуг, которые входят в начальное состояние $N(t)$ или покидают заключительное $N(s)$, не существует пути от i к f, который бы возвращался от $N(t)$ к $N(s)$. Следовательно, составной НКА распознает $L(s)L(r)$.

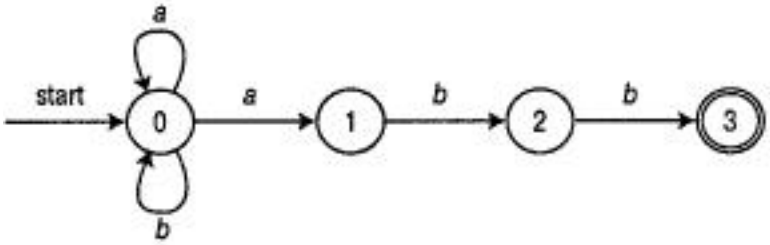
c) Для s* НКА такой:



d) Для (s) используем просто N(s)

В процессе создания НКА каждый шаг вносит не более двух новых состояний. В результате НКА, построенный по регулярному выражению, имеет количество состояний, превышающее число символов и операторов в регулярном выражении не более, чем в два раза.

Таблица переходов для НКА - это вот такое:



Состояние	Входной символ	
	a	*b*
0	{0, 1}	{0}
1	—	{2}
2	—	{3}

НКА может быть промоделирован непосредственно, однако в лексических анализаторах обычно используют ДКА, потому как их моделировать быстрее (хотя они и получаются больше по числу состояний). Типичный лексический анализатор имеет имитатор конечного автомата, который работает со сгенерированной по регулярным выражениям, описывающим токены, таблицей переходов.

Моделирование ДКА

Вход. Входная строка *x*, завершаемая символом конца файла **eof,** и ДКА *D* со

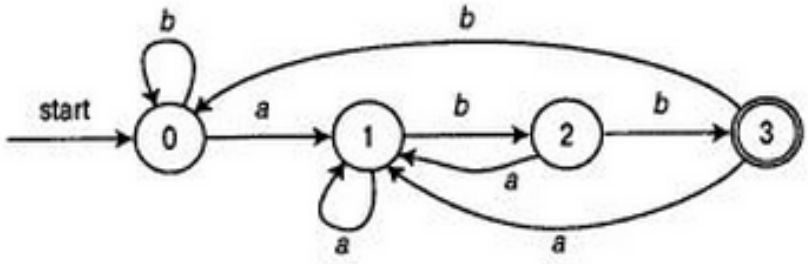
стартовым состоянием s0 и множеством заключительных состояний F.

Выход. "Да", если *D* допускает *x*, и "нет" в противном случае

Метод. Ко входной строке x применяется алгоритм, приведенный на рисунке ниже. Функция *move(s,c)* дает состояние, в которое происходит переход из состояния *s* при входном символе c. Функция *nextchar* возвращает очередной символ строки i.

```
while c ≠ eof do begin
    s := move(s,c);
    c := nextchar;
end;
if s ∈ F then
    return "да"
else return "нет"
```

На следующем рисунке показан граф переходов ДКА, допускающего язык (a|b)*abb, который мы уже рассматривали ранее. При работе с этим ДКА и входной строке ababb алгоритм моделирования пройдет последовательность состояний 0, 1, 2, 1, 2, 3 и ответит "да".



Литература

А. Ахо, Р. Сети, Дж. Ульман, М. Лам. Компиляторы. Принципы, технологии, инструменты. http://www.ozon.ru/context/detail/id/3829076/