NTNU – Trondheim
Norwegian University of
Science and Technology

Lexical analysis: Deterministic Automata

What we have

- A file, when you read it, is just a sequence of numbers from 0 to 255 (bytes):

  72, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100, …

- By convention, some of them represent text characters:

  ‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘ ‘, ‘w’,’o’,’r’,’l’,’d’,…

- At this level, a source program just looks like a gigantic pile of bytes, which is not very informative

What we don't want

- A programming language key word like, say, "while" will appear as the sequence

  w (119), **h** (104), **i** (105), **l** (108), **e** (10)

  and it would be very tiresome to write a compiler that detects this sequence every time the programmer wants to start a while loop.
- You can't stop them from calling a variable 'whilf':

  w (119), **h** (104), **i** (105), **l** (108), *(looks like we're starting a loop soon…)*

  ...**f** (102) *(dang, rewind to 119 and try again, this is not a loop)*

NTNU – Trondheim
Norwegian University of
Science and Technology

What we want

- A neat and tidy grouping of characters into meaningful lumps, so that we can operate on those without caring about each character they are made from:

  ‘i’, ‘f’, ‘(‘, ‘w’,’h’, ‘i’, ‘l’, ‘f’, ‘=’, ‘=’, ‘2’, ‘)’, ‘{‘, ‘x’, ‘=’, ‘5’, ‘;’, ‘}’

  is easier to read as

  if (whilf == 2) { x = 5; }

  because characters are grouped together as words and punctuation.
- We could even make the color-coding meaningful:

  keywords and punctuation

  delimiters of groups

  variables

  operators

  numbers

NTNU – Trondheim
Norwegian University of Science and Technology

What are the colors for?

- Consider this statement we already looked at:

  if (whilf == 2) { x = 5; }

- Consider this statement also:

  while (a < 42) { a += 2; }

  if we respect the same coloring, it piles up as

  while (a < 42) { a += 2; }

- These two statements have wildly different meanings, but they share the same structure as far as our colors are concerned:

  blue red green purple yellow red red green purple yellow blue red

- The structure they share is *syntactic* (or *grammatical*, if you like)
- The difference between them is *lexical*
- We're talking about *lexical* analysis today, but we'll need both, so we'll (eventually) try to get both from the stream of meaningless data.

Three useful words

- *Lexeme*
  - Lexemes are units of lexical analysis, words
  - They're like entries in the dictionary, "*house", "walk", "smooth"*
- *Token*
  - Tokens are units of syntactical analysis
  - They are units of sentence analysis, *"noun", "verb", "adjective"*
- *Semantic*
  - This is what something means, there is no sensible unit
  - It's like explanations in the dictionary
    - *"house: a building which someone inhabits"*
    - *"walk: the act of putting one foot in front of the other"*
    - *"smooth: the property of a surface which offers little resistance"*

("dictionary: a highly useful volume of text which was not consulted for these explanations")

Classes of lexemes

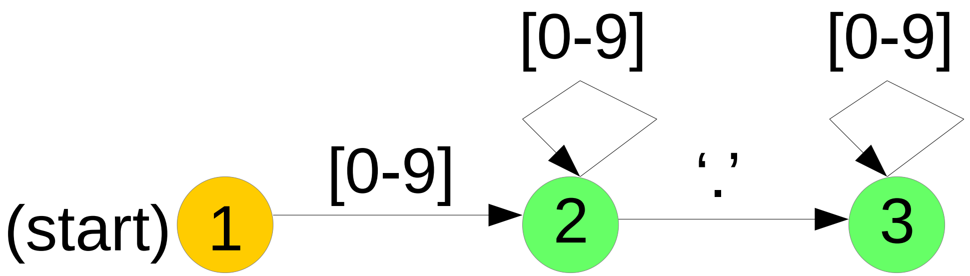
- Some of the words we want to classify are fixed:
  - “if”
  - “while”
  - “for”
  - “==”

    ...et cetera…
- Other classes have countably infinite instances:
  - 1
  - 2
  - …
  - ...65536…

    These are all specific cases of “integer”

Finite Automata

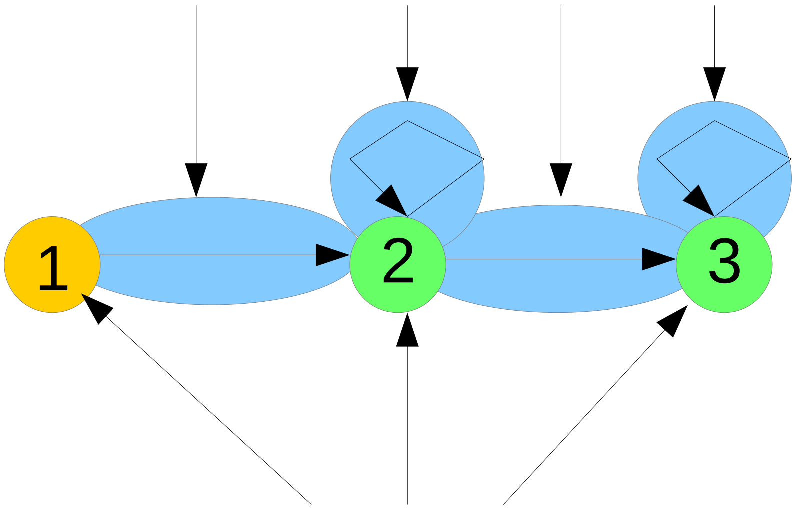
- We need a mechanism to identify not just single, specific words, but entire classes of them
- Forget all about specific numbers for a while, let's just try to find out whether we can make a rule to recognize a number when we see one
- Here's a *deterministic finite automaton,* (drawn as a directed graph, because that's easy to follow):



(You may remember these things from discrete mathematics, but I'll repeat them anyway)

Anatomy of a DFA

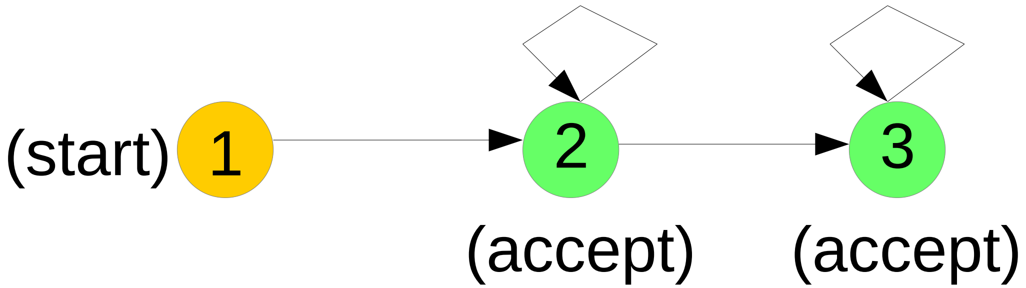
The edges/arcs represent *transitions* between states



These are the *states* (1, 2 and 3)

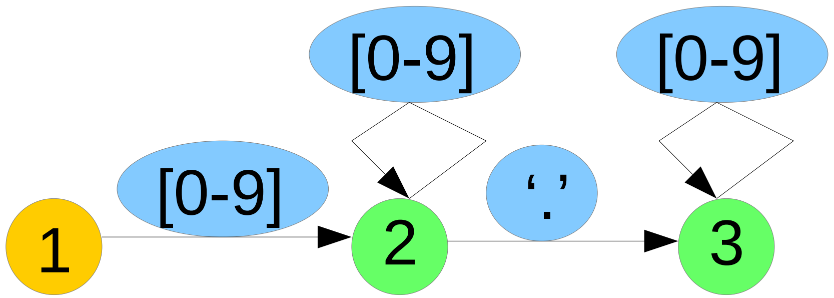
Start and finish

- One state is singled out as the *starting* state
- One or more states are identified as *accepting* states
  - I've colored them green here, other common notations are to use a double circle or thicker lines
  - Doesn't matter as long as we can tell what it means



Labels on the arcs

- Transitions are marked with sets of single characters that they apply to
  - ‘.’ means the period character
  - [0-9] is a shorthand for ‘0’ ‘1’ ‘2’ ‘3’ ‘4’ ‘5’ ‘6’ ‘7’ ‘8’ ‘9’



NTNU – Trondheim
Norwegian University of Science and Technology

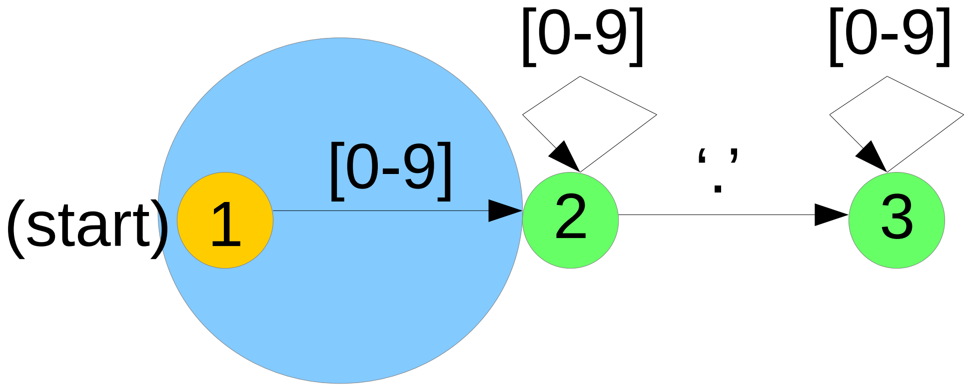
Traversing the graph

- The idea is that we start by pointing a finger at the starting state, and then
  - Read a character of text
  - Search for any transitions labeled with that character
  - Throw away* the character, and point at the new state instead
  - Repeat with another character until something fails
- When something fails, we're either pointing at an accepting state, or not.
  - If we are, the automaton accepts the text we read
  - If we are not, the text was wrong**

** Programs won't actually discard it, but the finite automaton no longer cares what it was*
*** "wrong" isn't really the best word, but it'll do for now*

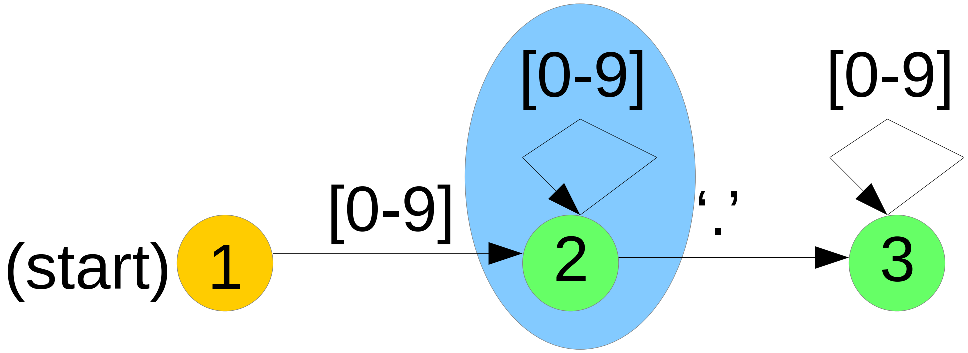
Take “42.64”

- We start in state 1
- Read ‘4’
- Find a transition



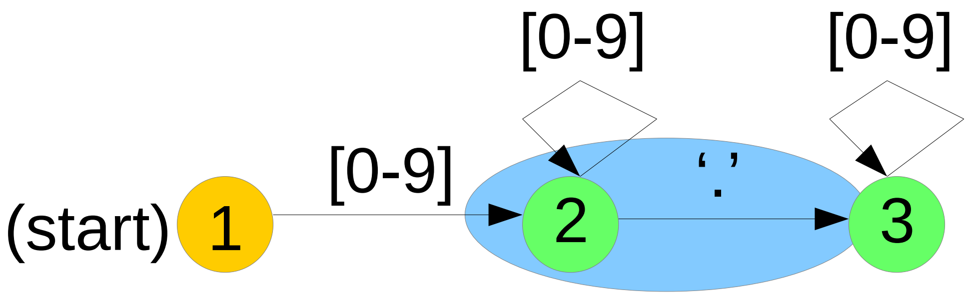
We’re left with “2.64”

- We’re in state 2
- Read ‘2’
- Find a transition



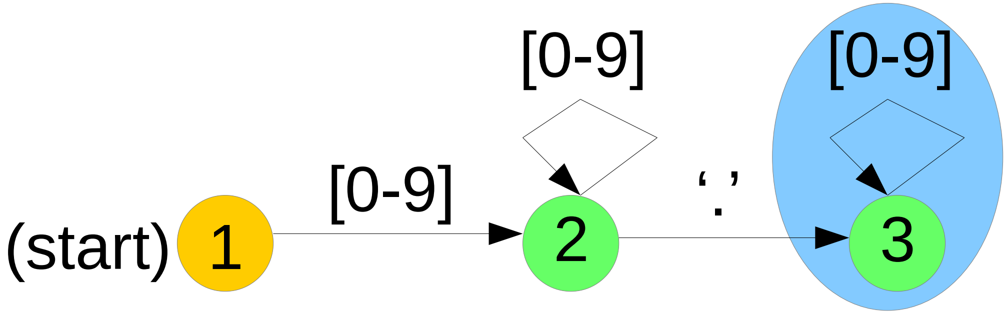
We’re left with “.64”

- We’re in state 2
- Read ‘.’
- Find a transition



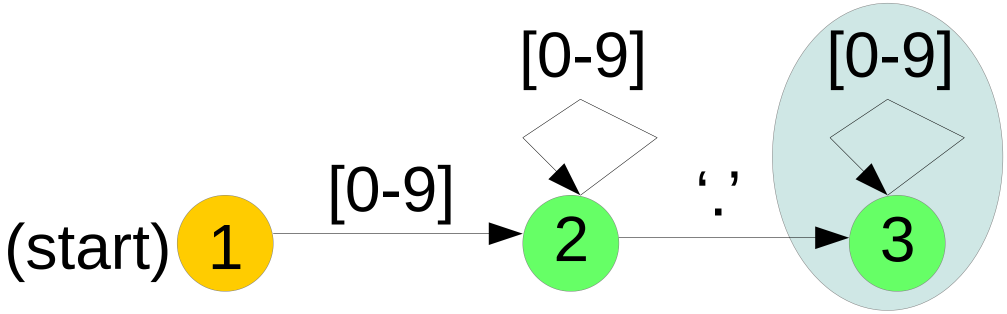
We're left with "64"

- We're in state 3
- Read '6'
- Find a transition



We’re left with “4”

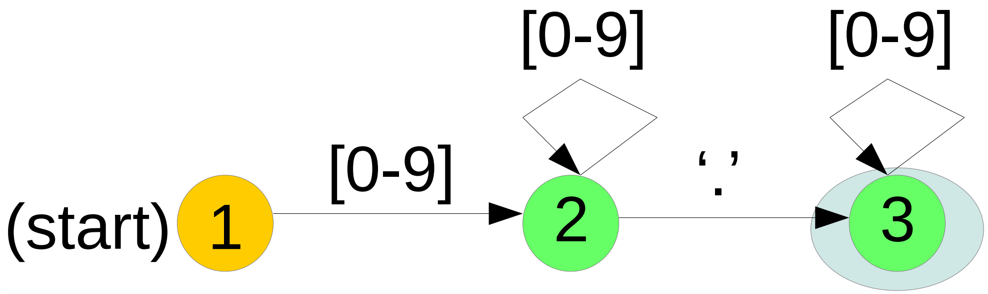
- We’re in state 3
- Read ‘4’
- Find a transition



We're out of characters...

- ...and standing in state 3
- That's an accepting state, so this automaton recognizes the word "42.64"
- The state sequence (1,2,2,3,3,3) which we just constructed is a *proof* of that

  (it's not so important to call *this* "a proof", but a couple of other proofs in this subject are constructed by just following a recipe, so we might as well say it right away.)

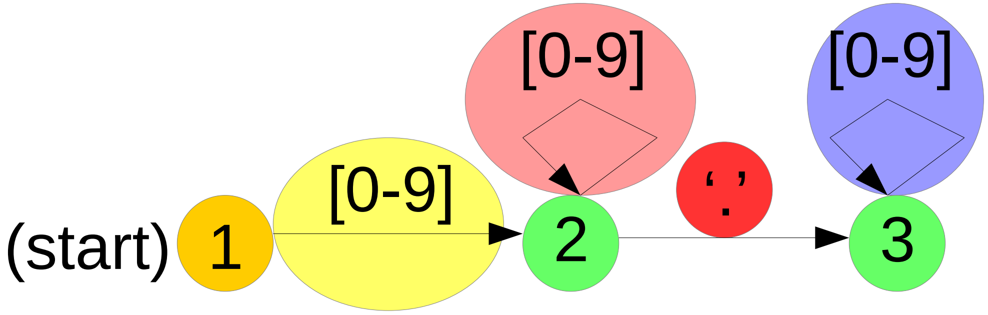


That was one class of words

- The DFA we just looked at recognizes integers with an optional (possibly empty) fractional part
  - How would you change it to reject, say, “42.” while still accepting “42.0”, or accept “.64”?
- Discriminating between all the classes of words in an entire programming language requires a whole bunch of different DFAs to work in conjunction
- Luckily, we can program them very generally

An alternative view

- One of the neat things about graphs is that we can write them up as tables
- Consider:



	Symbol(s)		
State	[0-9]	'.'	<*other*>
1	2	-	-
2	2	3	-
3	3	-	-

Here's "42.64" again, in the table view

- State 1, read '4', go to state 2

State	[0-9]	'.'	<*other*>	Accept?
1	2	-	-	No
2	2	3	-	Yes
3	3	-	-	Yes

- State 2, read '2', go to state 2

State	[0-9]	'.'	<*other*>	Accept?
1	2	-	-	No
2	2	3	-	Yes
3	3	-	-	Yes

Here’s “42.64” again, in the table view

- State 2, read ‘.’, go to state 3

State	[0-9]	‘.’	<*other*>	Accept?
1	2	-	-	No
2	2	3	-	Yes
3	3	-	-	Yes

- State 3, read ‘6’, go to state 3

State	[0-9]	‘.’	<*other*>	Accept?
1	2	-	-	No
2	2	3	-	Yes
3	3	-	-	Yes

NTNU – Trondheim
Norwegian University of Science and Technology

Here's "42.64" again, in the table view

- State 3, read '4', go to state 3

State	[0-9]	'.'	<*other*>	Accept?
1	2	-	-	No
2	2	3	-	Yes
3	3	-	-	Yes

- State 3, out of input, accept

State	[0-9]	'.'	<*other*>	Accept?
1	2	-	-	No
2	2	3	-	Yes
3	3	-	-	Yes

Implementation

- This is the algorithm in Dragon Fig. 3.27, p. 151
  - Store state (it's just a row index into the table)
  - Read character (it's just a column index)
  - Set state to the value found at entry (state,character) in the table
  - Repeat
- The beauty of this is that the same program logic works for any DFA, changes in the automaton only require a different table to work with, not a different algorithm

So far, so good

- We have a graph representation that we can draw on paper and follow by pointing fingers at the graph and text
- We have a table representation that we can turn into a program

Where we are going with this

- Programming a word-class recognizer (*lexical analyzer,* or *scanner*) with ad-hoc logic is complicated and error-prone
- Writing one using tables is a little easier, but requires punching in a bunch of boring table entries to represent specific DFAs
- *Generating* one is very convenient:
  - Specify word classes as regular expressions
  - Let a program write a gigantic table of states that includes all of the expressions

How can such a generator work?

- We'll need to write down the graph differently, programs have a really hard time understanding pictures
- We'll need a path from that notation and into tables
- Doing it automatically will give us bigger tables than we need
  - and thus, a great opportunity to shrink them to a minimum

(Stick around for the mesmerizing sequel, "*Lexical Analysis II: Attack of the NFA*")