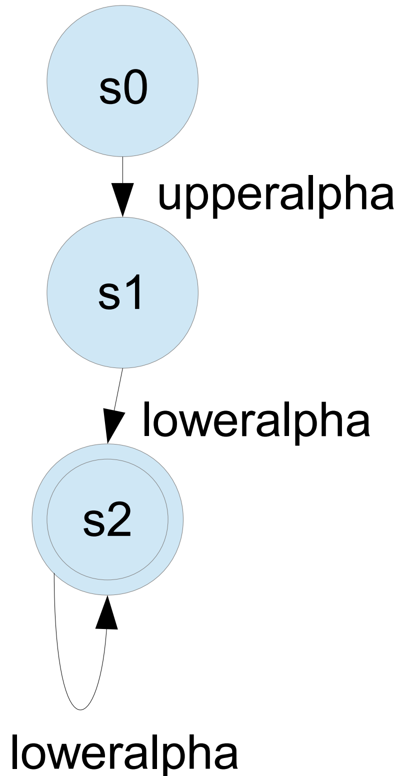
Table-driven scanners

- Uses stock algorithm driven by three tables to simulate the DFA
- Input classification table: given a char, identifies what kind of character it is *relevant to our DFA*, e.g. a digit, lowercase alpha, whitespace, etc
- Transition table: given the current state and the classification for the current input character, identifies the correct next state
- Token type table: given the ending state, identifies what kind of token that represents (or invalid: no token recognized)

Example: identifier [A-Z][a-z]+



Transition table

curr state	loweralpha	upperalpha	other
s0	reject	s1	reject
s1	s2	reject	reject
s2	s2	reject	reject
reject	reject	reject	reject

Input classifier

char	class
[a-z]	loweralpha
[A-Z]	upperalpha
other	other

Token type

end state	type
s0	invalid
s1	invalid
s2	identifier
reject	invalid

Notes on the tables

- Division into three tables allows size of each to be minimized (think database normalization), reducing memory costs and improving paging/cache performance
- Input classification table can get really large if alphabet is huge (e.g. Unicode), so might be replaced with classification functions

Generating tables

- Transition table: start with one char for each column, one row for each state, fill in as you read DFA. Once table filled in, collapse identical columns
- Classifier table: the set of chars for each column in transition table forms one classification group
- Token type table: in the beginning we had one RE for each token type, then join with |'s to form initial big NFA, annotate each accept state with matching token type, retain those annotations during transformation to DFA

The generic algorithm

- Initialization
- Initialize current state, curr = s0
- Initialize current token, tok = “”
- Initialize empty state stack, S
- Push special <END> state on stack

Algorithm: continued

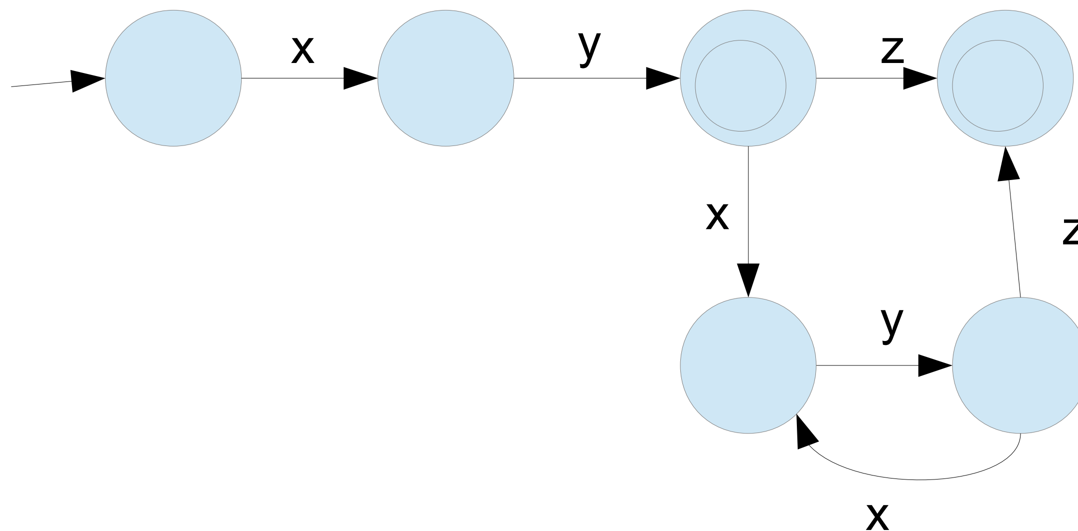
- Part 1: process as many input chars as possible:
- While not in reject state
  - Read next char, append to t
  - If S is an accept state then clear stack and push S
  - Lookup character type
  - Given character type and S, lookup next state in transition table
  - Set S to new state

Algorithm: continued

- Part II: roll back if necessary
- While S isn't an accept state and isn't <BOTTOM>
  - Pop top state off of stack into S
  - Chop last char off token
  - Roll back one character in input stream
- Lookup token type based on S

Excessive rollback

- Sometimes will be grabbing tiny tokens from front of input stream, but to do so reads all the way to the end then rolls back (can be O(n^2) in number of input characters)
- ((xy)+z)|xy
- ***xy***xyxyxy
- ..***xy***xyxy
- ....***xy***xy
- ......***xy***



Maximal munch scanner

- Uses counter, i, to track current position in input stream and a table to keep track of which state and input-position combinations are known cannot reach an accept state
- Initialize table entries to false, later will set some to true based on what we learn during run of algorithm

state	i=0	i=1	i=2	i=...etc..
s0	f	f	f	f
s1	f	f	f	f
s2	f	f	f	f
reject	f	f	f	f

Revised algorithm

- Initialization:
- i = 0 (current position in input stream)
- S = s0
- Empty stack
- Push <BOTTOM,i> on stack (a state and position)

Algorithm: continued

- Part 1: reading forward through input
- While not in reject state
  - Read next input char, concat to token
  - Increment i
  - If Failed[S][i] then exit loop
  - If S is an accept state then clear stack
  - Push <S,i>
  - Lookup char type/transition, update S

Revised algorithm: continued

- Part 2: rollback if needed
- While S is not accept state and not <Bottom,0>
  - Set Failed[S][i] to true (we just realized it goes nowhere)
  - Pop top element off stack, use to update S and i
  - Chop last char off token
  - Roll back one character in input stream
- Look up resulting token type