Scanning & Regular Languages: Part 2

An Important Insight

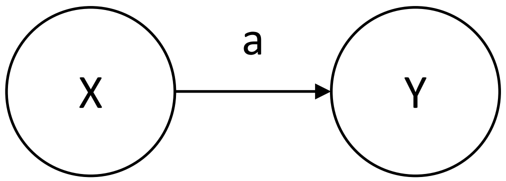
- Some regular languages are easy to describe with regular expressions. But some are actually a little unintuitive to describe using regular expressions, and are easier to describe by just **specifying the recognition program directly**.
- We're going to explore how to specify simple recognition programs using **finite automata** (also known as finite state machines).
- Recognition programs based on finite automata are efficient and easy to implement, and can actually recognize any regular language (though this fact is not obvious!)
- We will ultimately use finite automata to implement our scanner.

Deterministic Finite Automata

- **Deterministic finite automata (DFAs)** are a tool for describing simple language recognition programs, which work as follows:
  - The program has a *finite* number of distinct *states*.
  - The program can occupy one state at a time.
  - The program reads one character of input at a time, and cannot backtrack in the input.
  - Each time the program reads a character, the state is updated, following a *deterministic* process. The new state is completely determined by the previous state and the character that was read.
  - Some states are designated *accepting states*. Once all input has been read, the string is accepted if the current state is accepting, and rejected otherwise.

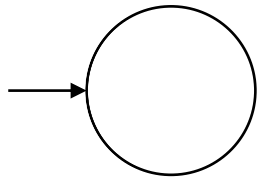
DFA State Diagrams

- Although DFAs represent recognition programs, we often represent them with *state diagrams,* rather than code.
- Each state in the program is represented by a circle.
  - The state can have a *name* written inside the circle.
  - Names are optional and don't have any meaning, but can make it easier to understand.
- If reading character "a" takes the program from state X to state Y, we draw it like this:

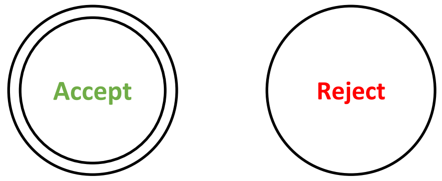


DFA State Diagrams, Continued

- The initial state of the program is represented by a circle with an arrow pointing inwards.

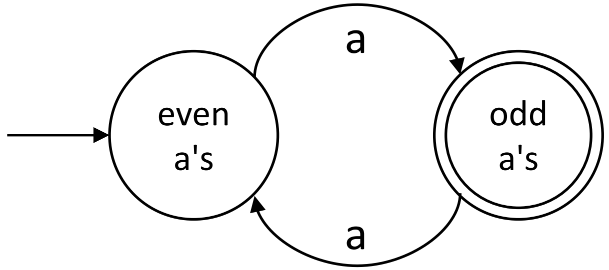


- If a state is "double-circled", it is an accepting state. If the program ends up in this state after reading all input, it will accept the input. Otherwise, it will reject the input.

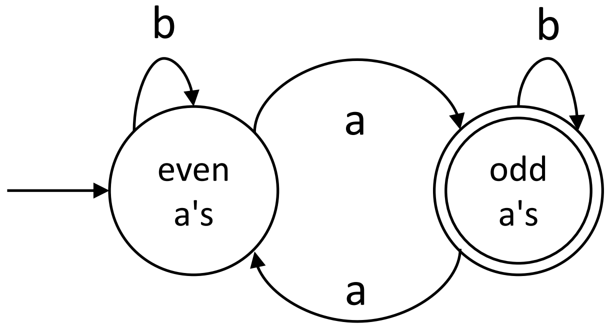


DFA Examples

- The following DFA recognizes strings of a's that have odd length.

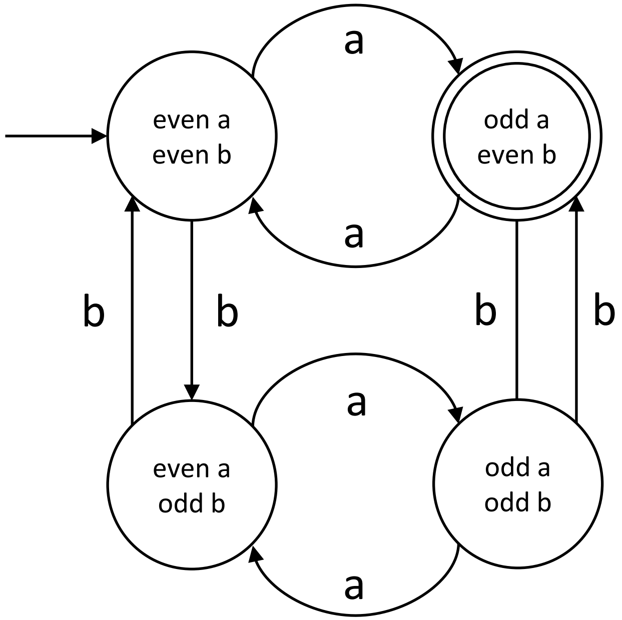


- The following DFA recognizes strings of a's and b's with an odd number of a's and an arbitrary number of b's.



DFA Examples

- Construct a DFA that recognizes strings of a's and b's with an odd number of a's and an even number of b's.
- If we change which state is accepting, we can recognize any combination of parity.

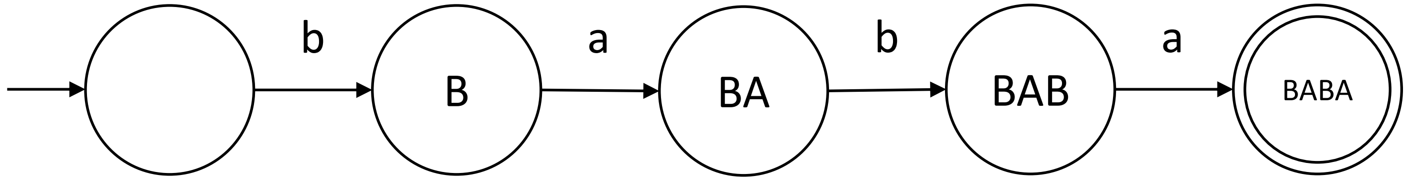


DFA: Strings Ending in "baba"

- The languages on the previous slides are probably easier to describe with DFAs than regular expressions.
- However, sometimes regular expressions do have the advantage.
- Consider this language:
  Strings of a's and b's that end with "baba".
- This is very easy to describe with a regular expression: "(a|b)*baba"
- Let's try to come up with a DFA.

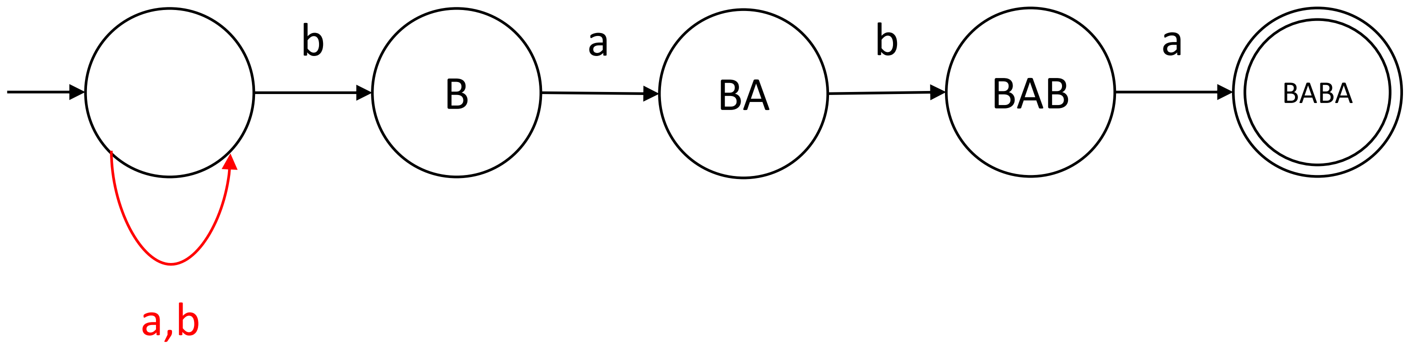
DFA: Strings Ending in "baba"

- We start out with a structure like this which just recognizes "baba".



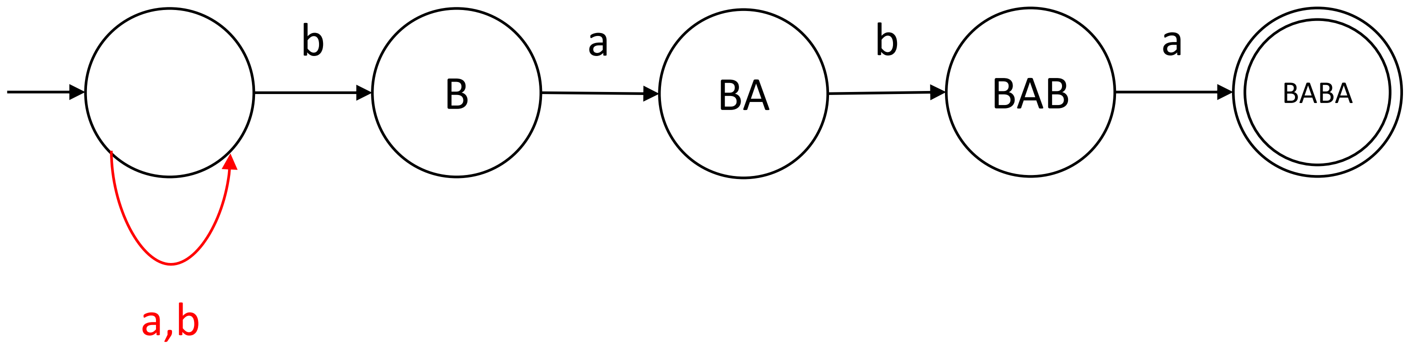
DFA: Strings Ending in "baba"

- Is there any problem if we do this?



DFA: Strings Ending in "baba"

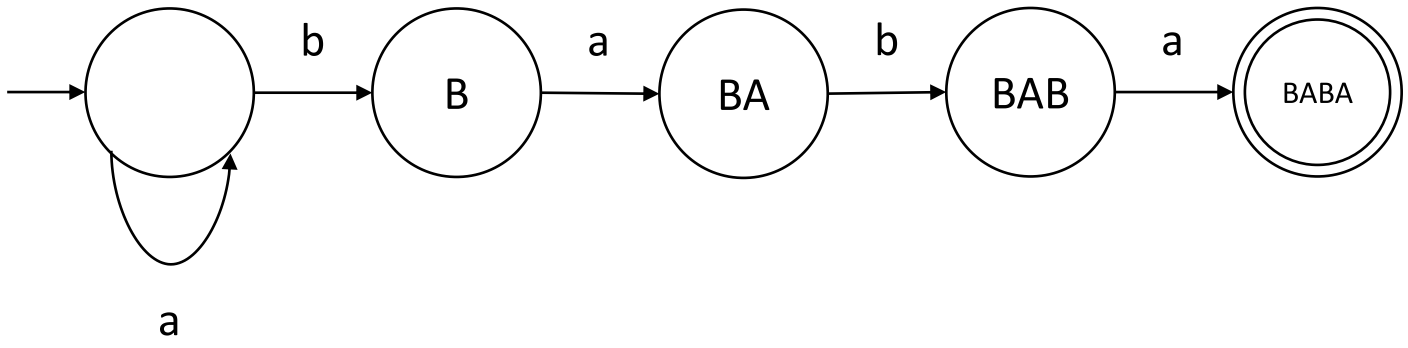
- Is there any problem if we do this?
- **Yes.** Remember that the next state depends **only** on the current state and the next symbol.



- In the empty state, if the next symbol is b, we have *two choices* (loop or go to state B). This is not allowed since **D**FAs are **deterministic**.

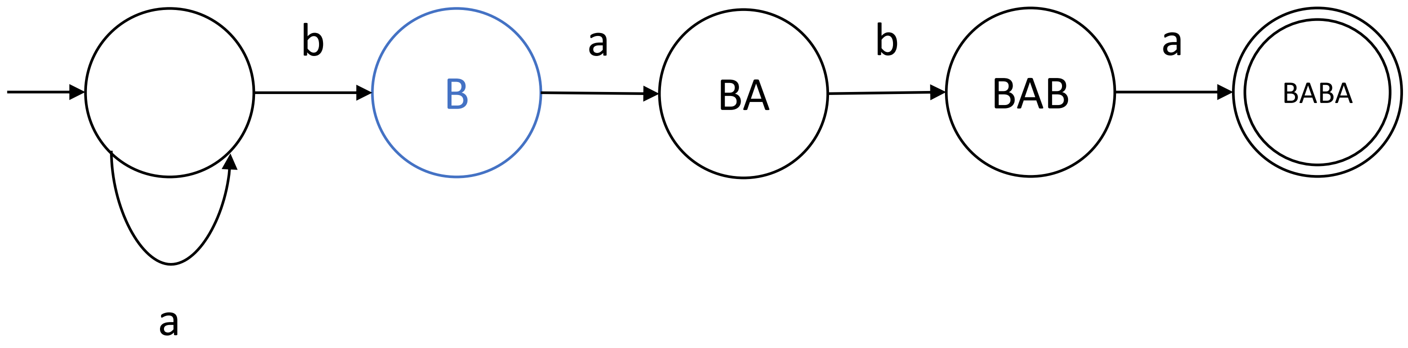
DFA: Strings Ending in "baba"

- Instead, for each state, let's carefully think about what we should do when we see a certain letter.
- From the empty state, looping back on "a" is okay.



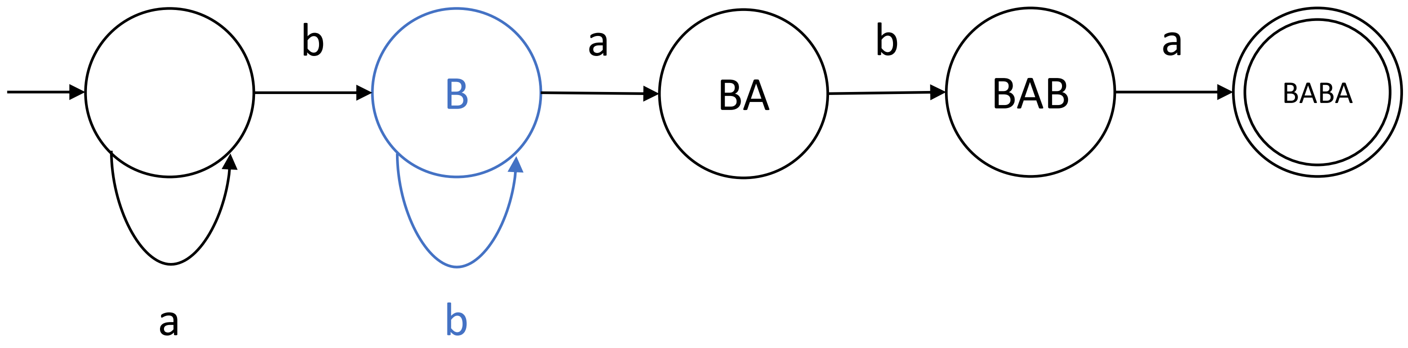
DFA: Strings Ending in "baba"

- Let's say we're in state B. Then the string currently ends with "b".



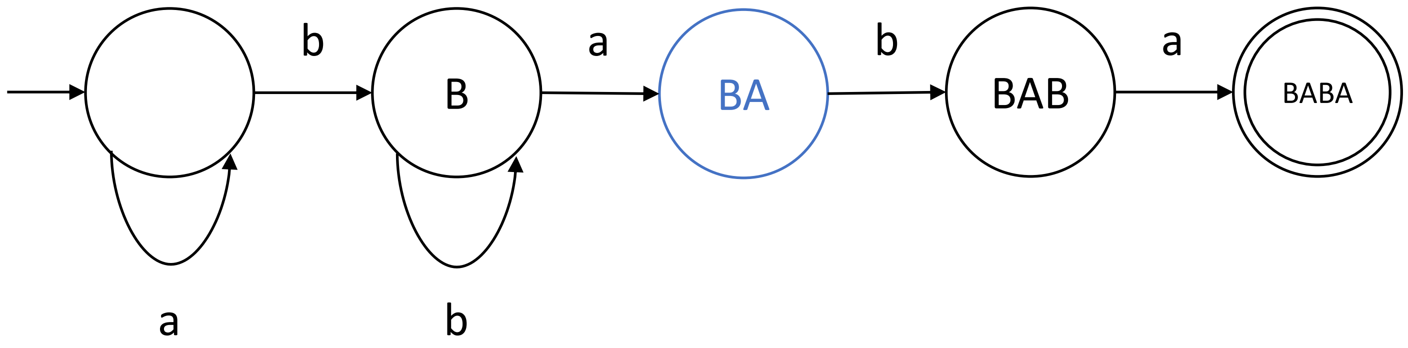
DFA: Strings Ending in "baba"

- Let's say we're in state B. Then the string currently ends with "b".
- If we see another "b", the string still ends with "b", and we are still waiting for "aba". Nothing has changed, so we can loop.



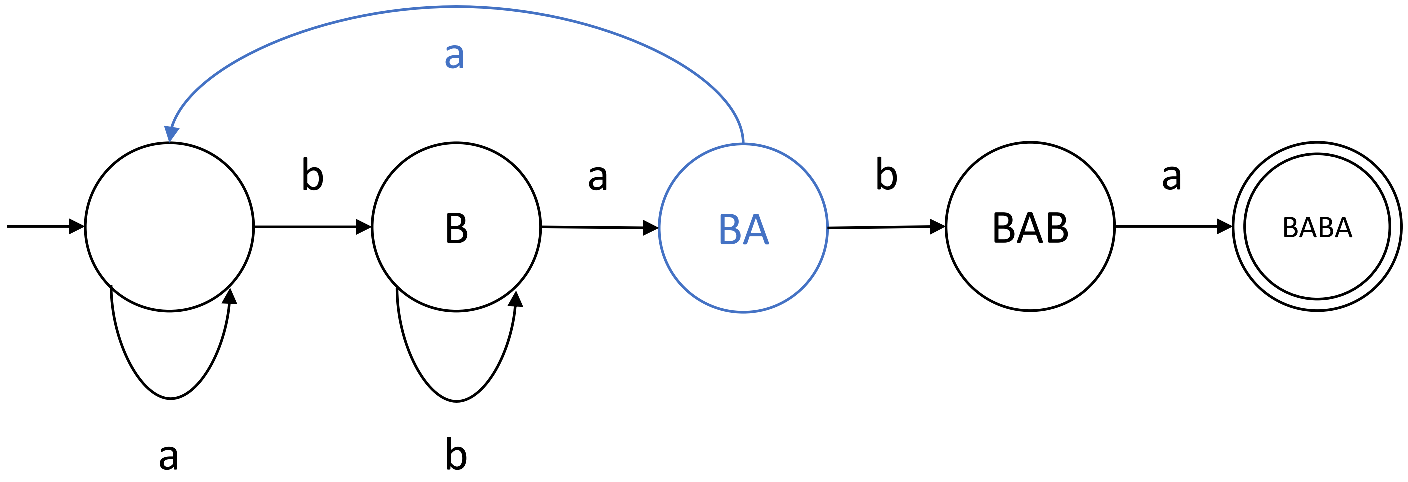
DFA: Strings Ending in "baba"

- Let's say we're in state BA. Then the string currently ends with "ba".
- If we see another "a", the string now ends with "baa". We lost our progress and now need to see the entire suffix "baba".



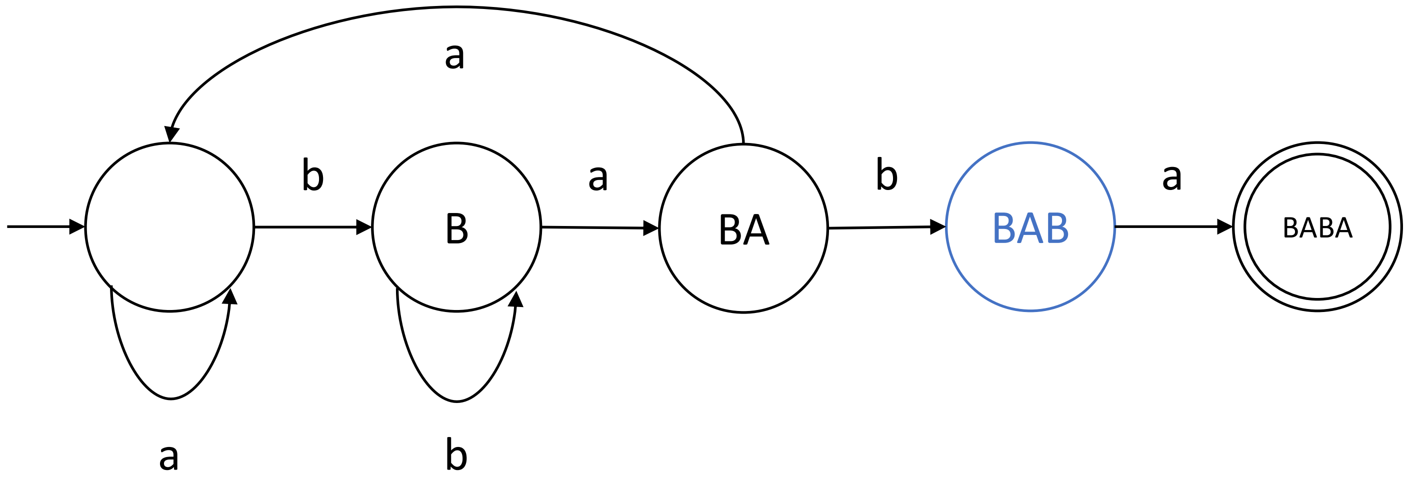
DFA: Strings Ending in "baba"

- Let's say we're in state BA. Then the string currently ends with "ba".
- If we see another "a", the string now ends with "baa". We lost our progress and now need to see the entire suffix "baba".
- We actually need to *backtrack* to the empty state if we see "a".



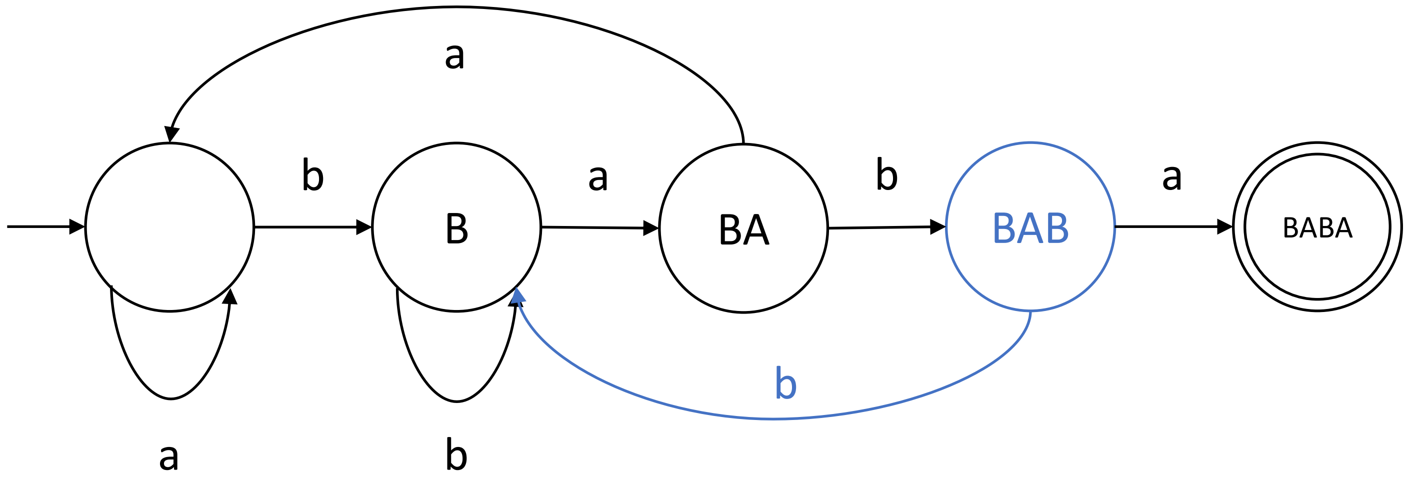
DFA: Strings Ending in "baba"

- Let's say we're in state BAB. The string currently ends with "bab".
- If we see another "b", the string ends with "babb". We didn't lose all progress, but we still need to see "aba".



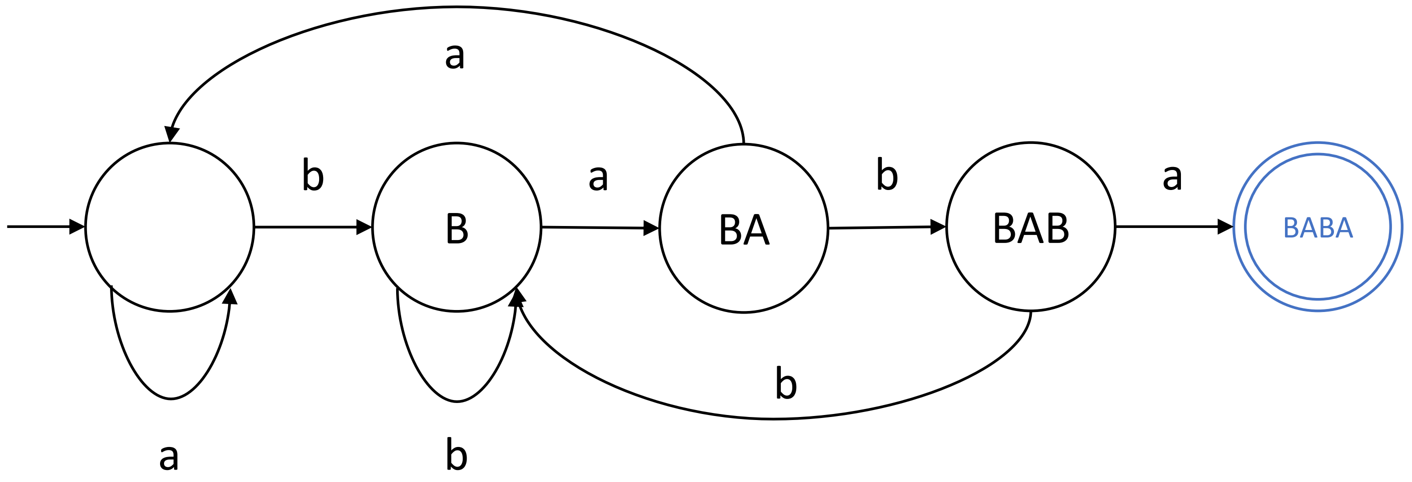
DFA: Strings Ending in "baba"

- Let's say we're in state BAB. The string currently ends with "bab".
- If we see another "b", the string ends with "babb". We didn't lose all progress, but we still need to see "aba".
- Backtrack to state B.



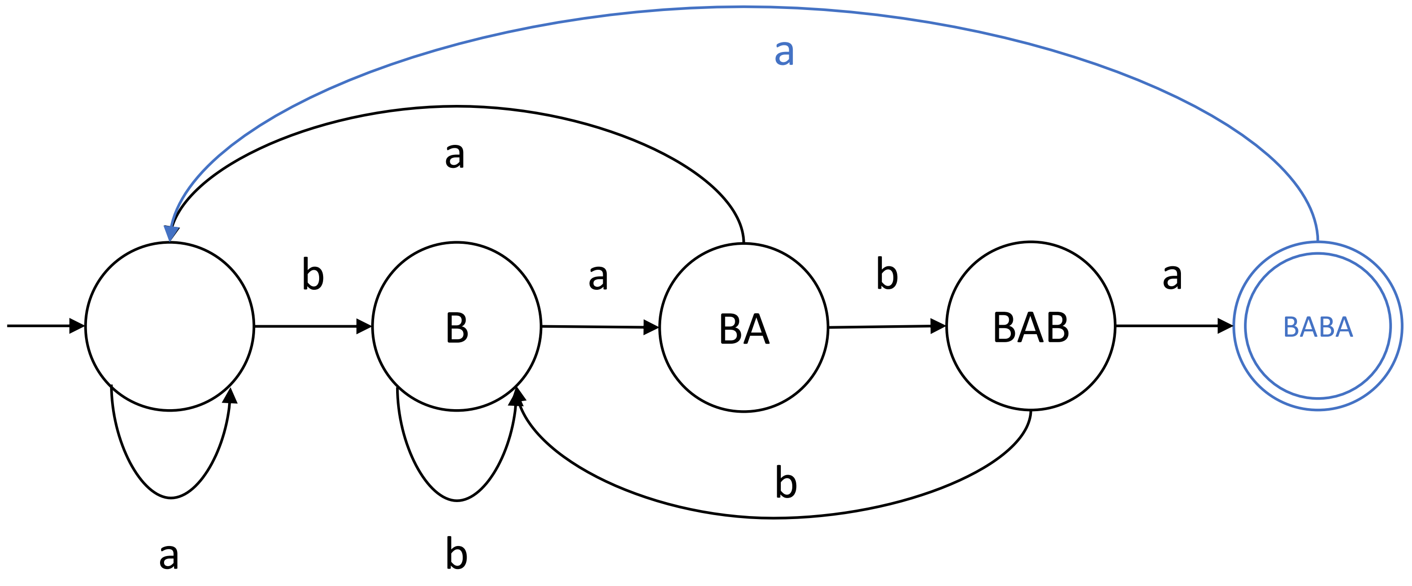
DFA: Strings Ending in "baba"

- Finally, state BABA. The string currently ends with "baba".
- If we see another "a", the string ends in "babaa" and we lost all our progress again.



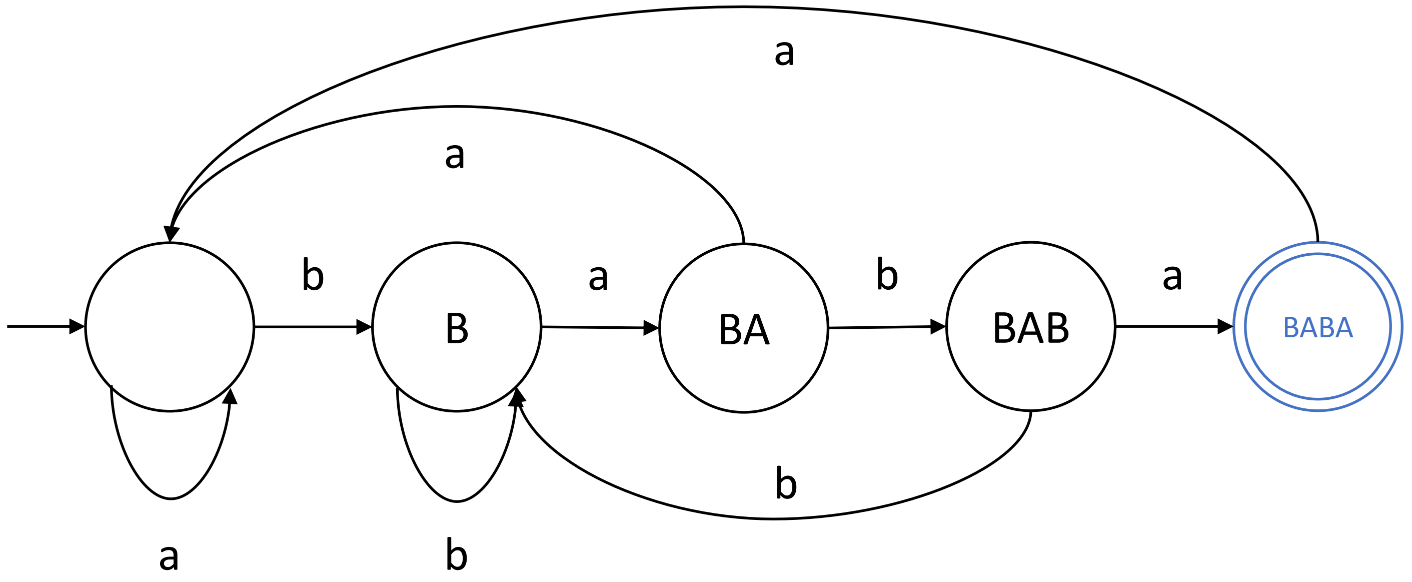
DFA: Strings Ending in "baba"

- Finally, state BABA. The string currently ends with "baba".
- If we see another "a", the string ends in "babaa" and we lost all our progress again. Go back to the empty state.



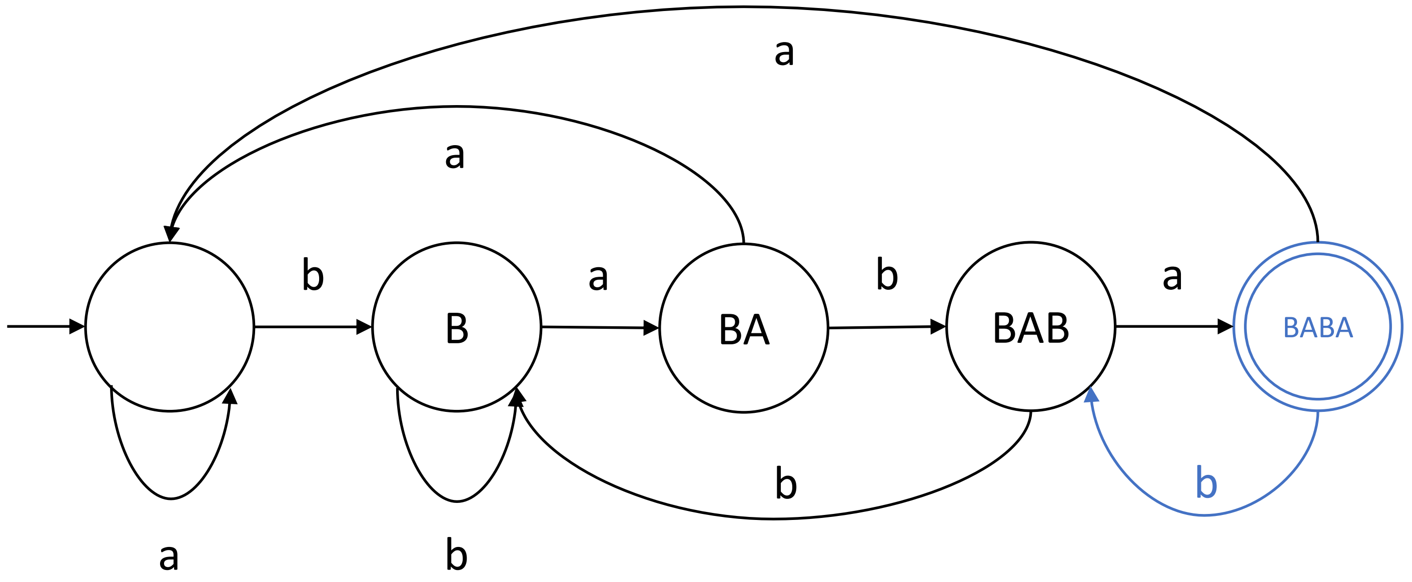
DFA: Strings Ending in "baba"

- Finally, state BABA. The string currently ends with "baba".
- If we see "b", the string ends in "babab". We only lost one letter of progress in this case (waiting for "a").



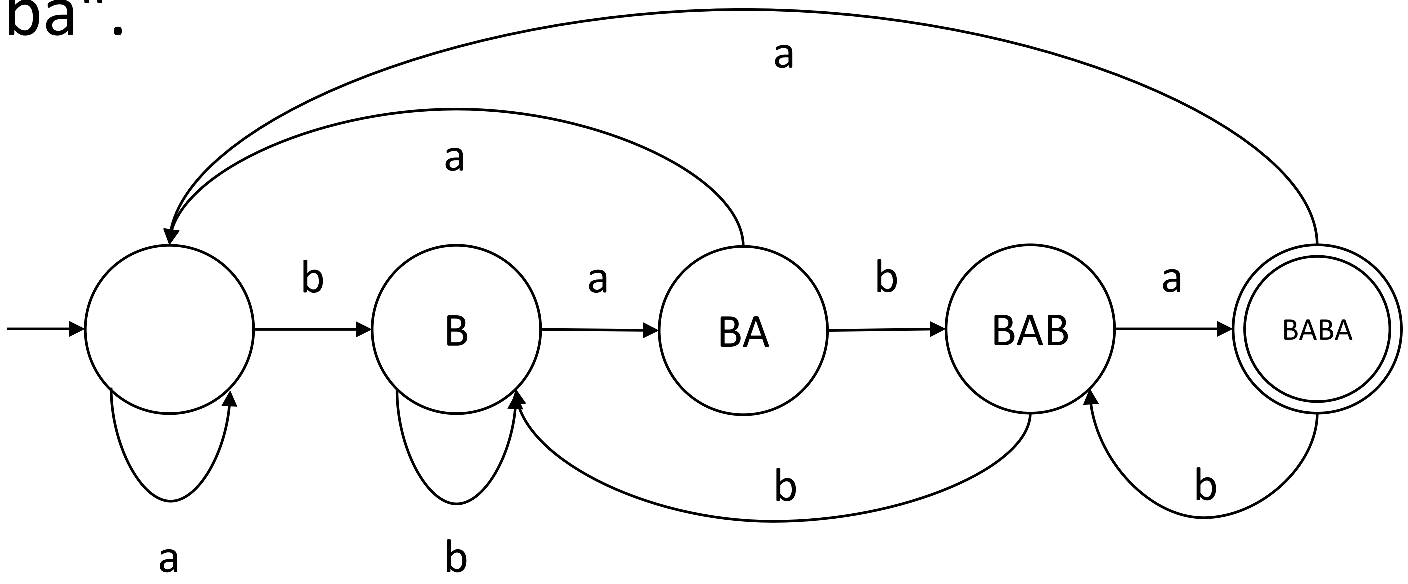
DFA: Strings Ending in "baba"

- Finally, state BABA. The string currently ends with "baba".
- If we see "b", the string ends in "babab". We only lost one letter of progress in this case (waiting for "a"). Go to state BAB.



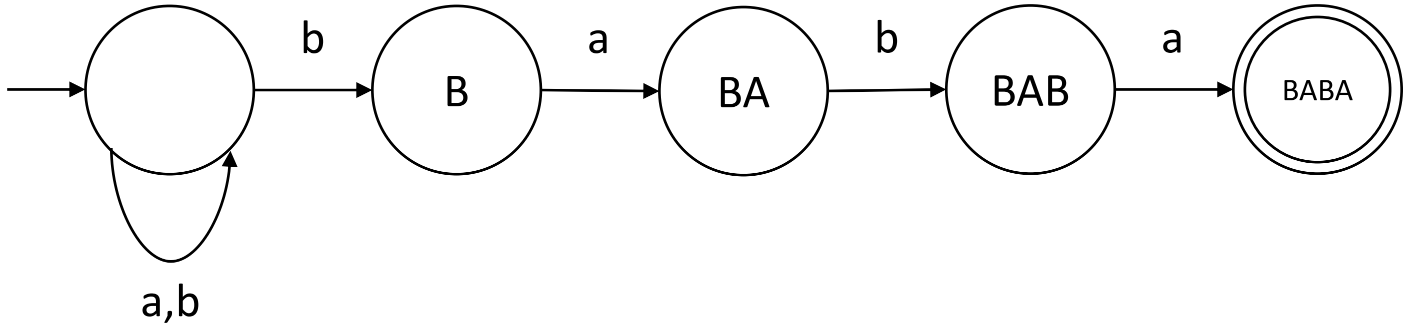
DFA: Strings Ending in "baba"

- We are done, because for each state, there is an arrow leading out on each symbol. The behaviour is totally specified.
- This was much more complicated than the regular expression "(a|b)*baba".



Nondeterministic Finite Automata?

- The "deterministic" aspect of DFAs might now seem inconvenient. What's wrong with just doing this?



- There is actually such a thing as a **nondeterministic finite automaton (NFA)** where this is valid**.** We will discuss them later in the course.
- NFAs are sometimes easier to come up with than DFAs. However, implementing the recognition program for a DFA is easier.

DFA Recognition Algorithm

- We can develop a general algorithm that takes both a string *and a DFA* as input, and determines if the DFA accepts a string.
- This algorithm is very efficient. It can be implemented in linear time in the length of the string, and linear space in the number of DFA states.
- The idea of the algorithm is just to "follow the arrows".
  - Start at the initial state and read characters from the string one at a time.
  - If there is an arrow on the current character, follow it to the next state.
  - If there is no arrow on the current character, reject the string.
  - After reading the entire string, if we ended up in an accepting state, accept the string.

Representing a DFA in Code

- How do we actually represent a DFA in a computer program?
- The fundamental components are:
  1. A list of states
  2. A specification of which states are accepting
  3. A specification of the "arrows" between states
- If the states are specified as strings, we can use *maps* for (2) and (3).
  - A [*state* → *boolean*] map that tells us whether a state is accepting.
  - A [(*state, character*) → *state*] map that encodes arrows. If there is an entry (X, a) → Y in the map, there is an arrow from state X to state Y labelled with a.
  - There's only one possible new state for each (state, character) pair!

Representing a DFA in Code

- How do we actually represent a DFA in a computer program?
- The fundamental components are:
  1. A list of states
  2. A specification of which states are accepting
  3. A specification of the "arrows" between states
- If the states are specified as integers, we can be more efficient and use arrays for (2) and (3).
  - Accepting[i] is true if state i is accepting, false otherwise.
  - Arrows[i][a] = j if there is an arrow from state i to state j on character a. (This assumes characters are encoded as small numbers, like in ASCII!)

DFA Recognition Algorithm: Pseudocode

- Let's assume we have the following two helper functions:
  - Accepting(X): Returns true if X is an accepting state, false otherwise.
  - Arrow(X, a): If there is an arrow leading out of state X labelled with character a going to state Y, return Y. Otherwise, return *undefined*.

```
state = initial state of DFA
for each character a in the input string:
  nextState = Arrow(state, a) (look for arrow to new state)
  if nextState is undefined:  (if there is no arrow…)
    return False              (reject input string)
  state = nextState           (otherwise go to new state)
return Accepting(state) (after reading the whole string…)
(accept if current state is accepting, otherwise reject)
```

DFAs and Regular Languages

- Every regular language can be recognized by a DFA, and every language recognized by a DFA is regular. (Kleene's Theorem)
- This fact is not obvious. We won't prove it in this course, but we will give some of the intuition later on.
- While some languages might be easier to describe with regular expressions than DFAs (and vice versa), we ultimately don't lose any expressive power by using DFAs.
- In fact, one common approach to implementing regular expression engines is to convert regular expressions to DFAs, then use the recognition algorithm we just saw!

Implementing Maximal Munch

- Our goal this whole time has been to implement the maximal munch algorithm:

1. Find the longest prefix of the input that is a valid token lexeme.
   If no such prefix exists, halt with an error (scanning failed).
2. If a prefix was found, remove it from the front of the input and generate a token for this prefix.
3. Repeat the above steps until either an error occurs, or the input becomes empty (scanning successful).

- We can do it if we represent the set of valid token lexemes as a DFA!

Implementing Maximal Munch

1. Find the longest prefix of the input that is a valid token lexeme.
   If no such prefix exists, halt with an error (scanning failed).

- Suppose we have a DFA for the language of valid token lexemes.
- One way to solve this problem is to take every prefix of the input and run it through the DFA recognition algorithm.
- But this would repeat the same work over and over again.
  - If we run "bab" through DFA recognition, and then next we run "baba" through DFA recognition, the same first three steps are repeated.
- Instead we use a *backtracking* strategy.

Implementing Maximal Munch

1. Find the longest prefix of the input that is a valid token lexeme.
   If no such prefix exists, halt with an error (scanning failed).

- Suppose we have a DFA for the language of valid token lexemes.
- Start running the input string through the DFA.
- Whenever we land on an accepting state, we *remember* two pieces of information: the state we are in, *and* the prefix of input read so far.
  - The prefix read so far is a valid token lexeme, since we reached an accepting state after reading it. We're just not sure if it's the *longest* one.
- Keep reading until the recognition algorithm gets **"stuck"**.

Implementing Maximal Munch

1. Find the longest prefix of the input that is a valid token lexeme.
   If no such prefix exists, halt with an error (scanning failed).

- Suppose we have a DFA for the language of valid token lexemes.
- Run the input string through the DFA, remembering the state and input prefix whenever we pass through an accepting state.
- Keep going until the DFA recognition algorithm gets "stuck":
  - There is no arrow leading out of the current state on the next character.
  - We reached the end of the input.
- The state we're stuck in can be **accepting** or **non-accepting**.

Implementing Maximal Munch

1. Find the longest prefix of the input that is a valid token lexeme.
   If no such prefix exists, halt with an error (scanning failed).

- Suppose we have a DFA for the language of valid token lexemes.
- Run the input string through the DFA, remembering the state and input prefix whenever we pass through an accepting state, until we get "stuck". We can be stuck in an **accepting** or **non-accepting** state.
- If we're stuck in an **accepting** state, the prefix we read so far is a valid token, and no longer prefix will be accepted, so we found the desired prefix and we're done Step 1!

Implementing Maximal Munch

1. Find the longest prefix of the input that is a valid token lexeme.
   If no such prefix exists, halt with an error (scanning failed).

- Suppose we have a DFA for the language of valid token lexemes.
- Run the input string through the DFA, remembering the state and input prefix whenever we pass through an accepting state, until we get "stuck". We can be stuck in an **accepting** or **non-accepting** state.
- If we're stuck in a **non-accepting** state, the prefix we read so far is not a valid token. Fortunately, we **remembered** the last prefix that was valid, and what state we were in, so we can **backtrack** to that point!

Implementing Maximal Munch

1. Find the longest prefix of the input that is a valid token lexeme.
   If no such prefix exists, halt with an error (scanning failed).
   - Run the input string through the DFA for valid token lexemes, using the same process as the DFA recognition algorithm.
   - Whenever we pass through an accepting state, take note of the state itself and the current prefix of the input we have read.
   - If we get "stuck" (no valid next arrow or reached end of input) in a non-accepting state, backtrack **in both the DFA and the input** to the last accepting state and prefix we remembered. This is the *longest prefix* we are looking for.
   - We remember the state since it can tell us information about the token's kind.
   - If we get "stuck" and never passed through an accepting state, that is an error.

Implementing Maximal Munch

- Our goal this whole time has been to implement the maximal munch algorithm:

1. Find the longest prefix of the input that is a valid token lexeme. If no such prefix exists, halt with an error (scanning failed).
2. If a prefix was found, remove it from the front of the input and generate a token for this prefix.
3. Repeat the above steps until either an error occurs, or the input becomes empty (scanning successful).

Implementing Maximal Munch

2. If a prefix was found, remove it from the front of the input and generate a token for this prefix.

- Because of how we implemented Step 1, "removing the prefix" is something that happens implicitly.
- We backtrack in the input to the point right after we read the desired prefix. So if we just go back to Step 1 and continue from where we left off, that's the same as "removing the prefix".
- **Generating a token:** The prefix becomes the lexeme. We can often assign a kind to the lexeme by looking at which state we ended up in, but analysis of the lexeme itself may also be needed.

Implementing Maximal Munch

- Our goal this whole time has been to implement the maximal munch algorithm:

1. Find the longest prefix of the input that is a valid token lexeme.
   If no such prefix exists, halt with an error (scanning failed).
2. If a prefix was found, remove it from the front of the input and generate a token for this prefix.
3. Repeat the above steps until either an error occurs, or the input becomes empty (scanning successful).

- Reset the DFA and repeat Step 1 from where we left off in the input.

Simplified Maximal Munch

- The need for backtracking makes maximal munch a little finicky to implement correctly. The version we presented also has poor performance (quadratic time) in some cases due to the backtracking.
- In the first project, we'll ask you to implement a simplified version with *no backtracking*. This version does not need to "remember" anything.
- When "stuck" in a non-accepting state, simplified maximal munch just gives up and produces an error.
- It is easier to implement and efficient (linear time) but it is naturally more limited in what it can correctly tokenize. Still, it is often "good enough".
- Pseudocode for simplified maximal munch is given in the course notes.

A Scanning Example: Regular Expressions

- The **grep** and **egrep** tools in Unix lets you search for lines in a file matching a regular expression. (egrep has simpler syntax than grep)
- Example: `egrep "a(ba)*|c" file.txt` finds all lines in `file.txt` that contain a substring matching the expression `a(ba)*|c`.
- Part of implementing a tool like this involves breaking down the regular expression into smaller parts to understand its meaning.
- Scanning might not seem necessary for *formal* regular expressions since they are composed by combining individual characters.
- But practical implementations often have extra syntax and features.

A Scanning Example: Regular Expressions

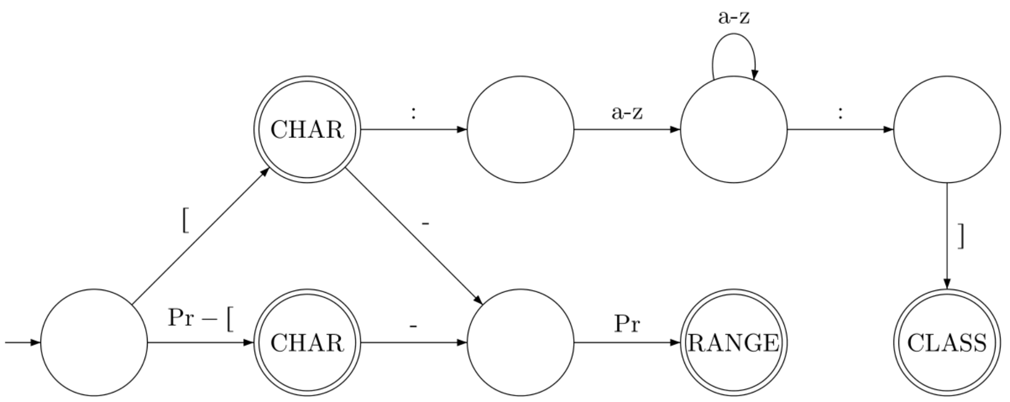
- The grep and egrep tools have additional features not present in the formal version of regular expressions.
- For example, they support **character sets** enclosed in square brackets:
  - [abc] is a shorter way to write (a|b|c).
  - Character ranges are supported: [a-z] means (a|b|c|…|z).
  - These elements can be combined: [241a-z] means (2|4|1|a|b|c|…|z).
  - Special character classes are supported: [[:alnum:]] is short for [a-zA-Z0-9].
  - You can even do something like [_[:alnum:]] (alphanumerics and underscore).
- Let us create a **scanning DFA for character sets** that breaks them down into smaller parts for easier processing.

A Scanning Example: Regular Expressions

- We will formally define a character set as a sequence of **tokens** surrounded by an opening [character and a closing] character.
- The allowed token kinds and their lexemes are:
  - CHAR: A single printable ASCII character.
  - RANGE: A CHAR, followed by a - character, followed by a CHAR.
  - CLASS: A string of the form [:*name*:], where *name* is a non-empty sequence of lowercase alphabetic characters.

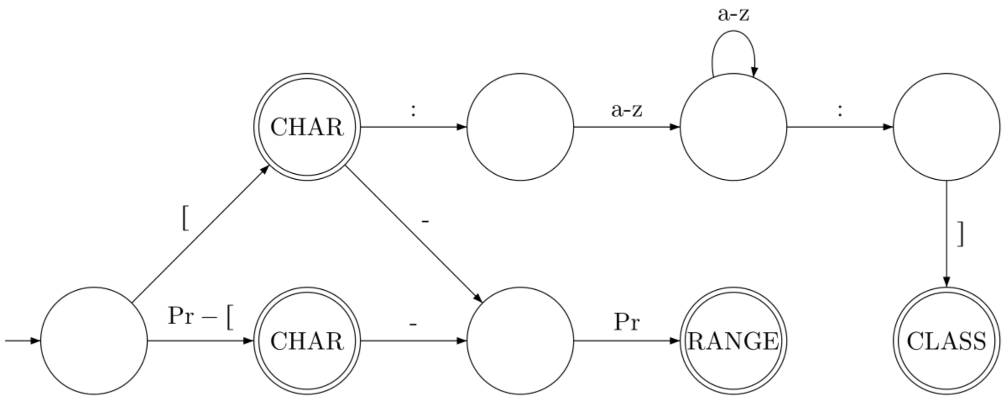
Here is a DFA for these tokens:

- Pr means printable characters
- Pr – [excludes the [character



A Scanning Example: Regular Expressions

- Pr means printable characters
- Pr – [excludes the [character

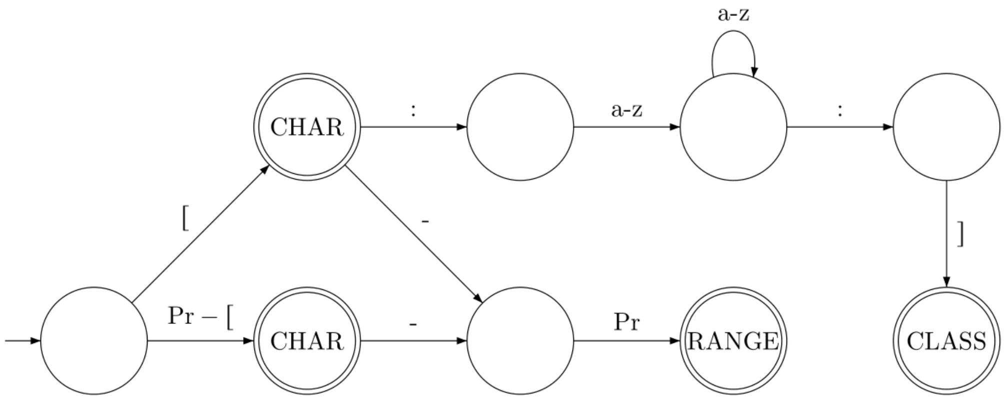


- Let's go through the following examples:

1. Scan **"x[:digit:]a-fA-F"** using simplified maximal munch.
2. Scan **":3-[:[:["** using maximal munch.
3. Scan **"[::"** using simplified maximal munch.

A Scanning Example: Regular Expressions

- Pr means printable characters
- Pr – [excludes the [character

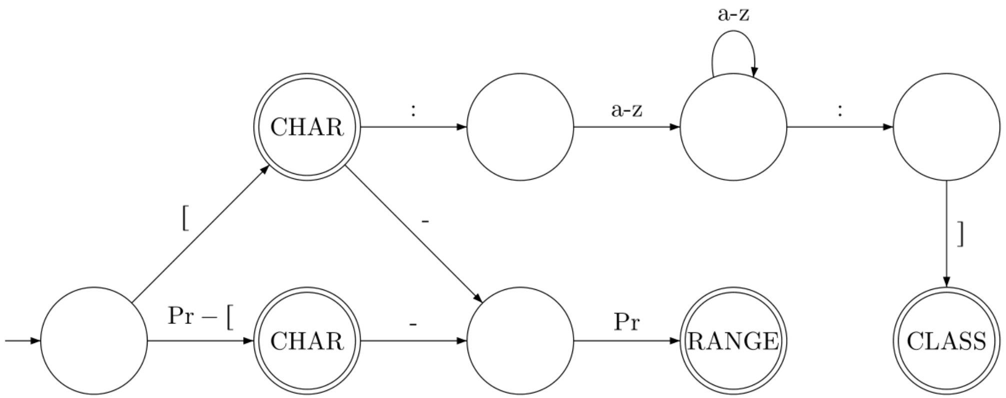


1. Scan **"x[:digit:]a-fA-F"** using simplified maximal munch.
   - **CHAR "x"** (remaining string: **"[:digit:]a-fA-F"**)
   - **CLASS "[:digit:]"** (remaining string: **"a-fA-F"**)
   - **RANGE "a-f"** (remaining string: **"A-F"**)
   - **RANGE "A-F"** (remaining string: **""**)

- The input string is empty, so we output the tokenization successfully.

A Scanning Example: Regular Expressions

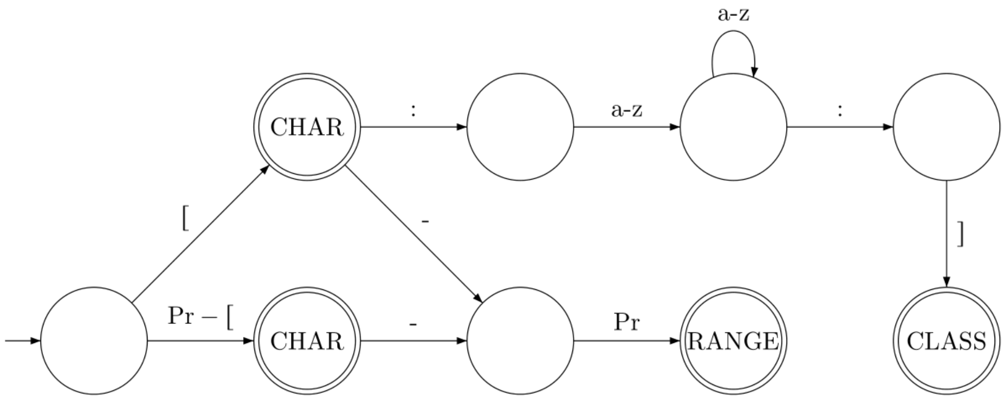
- Pr means printable characters
- Pr – [excludes the [character



2. Scan **":3-[:[:["** using maximal munch.
   - **CHAR ":"** (remaining string: **"3-[:[:["**)
   - **RANGE "3-["** (remaining string: **":[:["**)
   - **CHAR ":"** (remaining string: **"[:["**)
   - After reading **"[:"** we get stuck on **[** in a **non-accepting** state. **Backtrack** and output:
     CHAR "[" (remaining string: **":["**)
   - **CHAR ":"** then **CHAR "["** (remaining string: **""**) Scan successful!

A Scanning Example: Regular Expressions

- Pr means printable characters
- Pr – [excludes the [character



3. Scan **"[::"** using simplified maximal munch.
    - As before, after reading **"[:"** we get stuck on **[** in a **non-accepting** state.
    - In Simplified Maximal Munch, this is an **ERROR** and we stop scanning.
    - Maximal Munch would successfully scan this as:
      CHAR "["   CHAR ":"   CHAR ":"

Limitations of Maximal Munch

- We just saw an example where simplified maximal munch fails but maximal munch works.
- Sometimes a tokenization exists, but maximal munch does not find it.
- Example: token lexemes = {"aa", "aaa"}, input string = "aaaa".
  - Maximal munch will find token **["aaa"]**, then produce an error.
  - But this can be tokenized as **["aa"] ["aa"]**.
- Sometimes even if maximal munch produces a tokenization, it might not be the "best" or "expected" one.
- Example: C++ template parameters. Consider **vector<pair<int,int>>**.
- Prior to C++11, the C++ scanner interpreted **>>** as an operator and produced an error. Had to write **vector<pair<int,int> >** with a space.

Looking Forward

- Why did we take a break from studying machine language and start studying scanning?
- Machine language is annoying to write, so we wanted to use **assembly language** instead.
- MIPS instructions are a lot easier to read and write if we use assembly language.
- This will allow us to write more complex programs in MIPS. (Or at least make it much easier!)

Instruction	Assembly Syntax	Behaviour
Add	add $d, $s, $t	$d = $s + $t
Subtract	sub $d, $s, $t	$d = $s - $t
Multiply	mult $s, $t	hi:lo = $s * $t
Multiply Unsigned	multu $s, $t	hi:lo = $s * $t
Divide	div $s, $t	lo = $s / $t; hi = $s % $t
Divide Unsigned	divu $s, $t	lo = $s / $t; hi = $s % $t
Move from High	mfhi $d	$d = hi
Move from Low	mflo $d	$d = lo
Load Immediate & Skip	lis $d	$d = MEM[PC]; PC += 4
Set Less Than	slt $d, $s, $t	$d = 1 if $s < $t; 0 otherwise
Set Less Than Unsigned	sltu $d, $s, $t	$d = 1 if $s < $t; 0 otherwise
Jump Register	jr $s	PC = $s
Jump & Link Register	jalr $s	temp = $s; $31 = PC; PC = temp
Branch On Equal	beq $s, $t, i	if ($s == $t) PC += 4 * i
Branch On Not Equal	bne $s, $t, i	if ($s != $t) PC += 4 * i
Load Word	lw $t, i($s)	$t = MEM[$s + i]
Store Word	sw $t, i($s)	MEM[$s + i] = $t

Directive	Assembly Syntax	Behaviour
Encode As Word	.word i	i is encoded in the program as a 32-bit word

Looking Forward

- MIPS assembly language syntax is simple enough that you might be able to get by without a scanner, but it might be awkward to deal with things like whitespace:

  `add $1, $2, $3` and `add$1,$2,  $3` are both valid

- Scanning simplifies the process of understanding the meaning of a program, and using a DFA for scanning often means you just need to figure out how to describe the valid tokens, then implement MM or SMM.
- Next time, we'll start writing an **assembler** for MIPS and look at how to create a scanning DFA for MIPS tokens!

```
add $d, $s, $t
sub $d, $s, $t
mult $s, $t
multu $s, $t
div $s, $t
divu $s, $t
mfhi $d
mflo $d
lis $d
slt $d, $s, $t
sltu $d, $s, $t
jr $s
jalr $s
beq $s, $t, i
bne $s, $t, i
lw $t, i($s)
sw $t, i($s)
```