PG4200: Algorithms And Data Structures
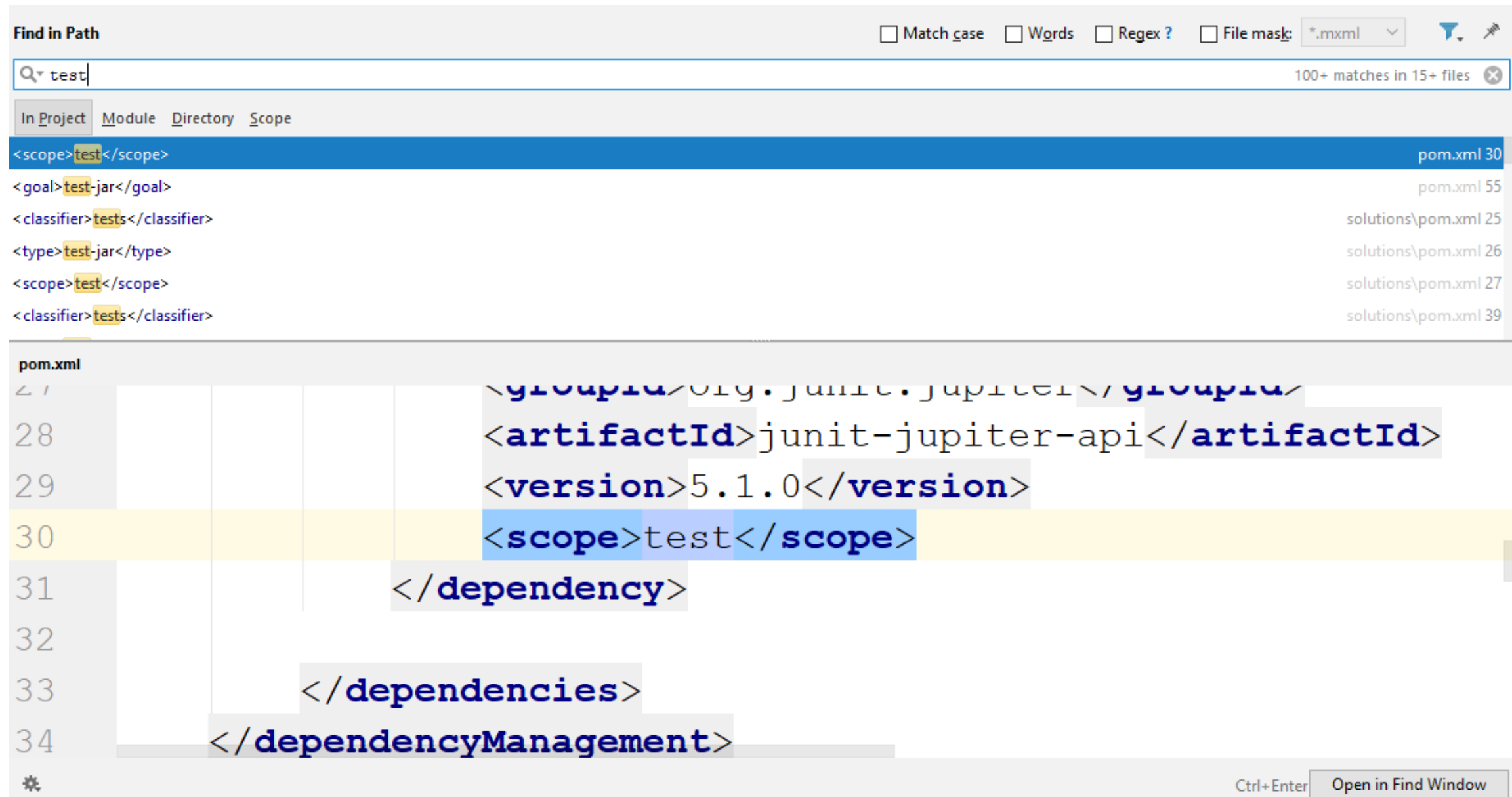
Lesson 09:
Text Search and
Regular Expressions

Dr. Andrea Arcuri

Text Search

Search Words in Text

- Text search is very common, ie in all editors

- But many other applications, eg Web Search Engines and Log Analysis

Definitions

- A *text* of length *N* we want to search in

- A *target* string of length *M* that we want to search, and check its position inside *text*
  - usually a word, but can be any string

- Example: text="*A needle in a haystack*", target="*needle*"
  - recall, we can think of strings like arrays of chars
  - N=22, M=6, matching at position from 2 to 7

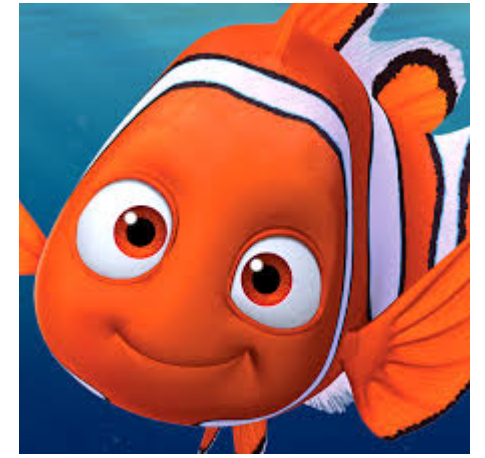| 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A		n	e	e	d	l	e		i	n		a		h	a	y	s	t	a	c	k
		n	e	e	d	l	e														

Brute Force

- For each position "i" in N, check if the next M chars match the values in the target

- If a single letter does not match, continue with "i+1"

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A		n	e	e	d	l	e		i	n		a		h	a	y	s	t	a	c	k

i=0	n							
i=1		n						
i=2			n	e	e	d	l	e

Searching For "Nemo" (case-insensitive)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A	n	e	e	d	l	e	i	n	a	h	a	y	s	t	a	c	k

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
N																					
	N																				
		N	e	m																	
			N																		
				N																	
					N																
						N															
							N														
								N													
									N												
										N	e										
											N										
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

Worst Case

- text="*aaaaaaaaab*", target="*aaac*", N=10, M=4

- For each N-M position, we need to check M chars… can only skip when we see a mismatch on the last char in the target

- Complexity: *O(N * M)*

0	1	2	3	4	5	6	7	8	9
a	a	a	a	a	a	a	a	a	b
a	a	a	c						
	a	a	a	c					
		a	a	a	c				
			a	a	a	c			
				a	a	a	c		
					a	a	a	c	
						a	a	a	c

Can We Do Better?

- text="*aaazaaaaca*", target="*aaac*", N=10, M=4

- When we see the char "z" which is not in the target, there is no point in looking at starting positions i+1 before it, as impossible for them to be correct (as they would contain "z")

- Could just jump over at the position after "z" (i=4 in this case)

0	1	2	3	4	5	6	7	8	9
a	a	a	z	a	a	a	a	c	a
a	a	a	c						
				a	a	a	c		
					a	a	a	c	

Partial Match

- text="*ababacaa*", target="*abac*", N=8, M=4

- For i=0, we have mismatch at position 3

- But we can skip i=1, and go to i=2

- If we had a match for target[1]=b at i=0, then **FOR SURE** we cannot have a match for target[0] at i+1, as we expect [0]=a, whereas we know that it is b

- Even if we have mismatch, ie b!=target[3], then **FOR SURE** we know that text[2] and text[3] would be the beginning ("ab") of a match starting at i=2

0	1	2	3	4	5	6	7
a	b	a	b	a	c	a	a
a	b	a	c				
		a	b	a	c		

Knut-Morris-Pratt Algorithm

- Look at each char in the text *only once* (and not possibly *O(N*M)* like in brute-force)

- Keep track of which element "j" in the target we are matching

- When there is mismatch between target[j] and text[i], need to update "j" before looking at next i+1

- If what read so far would be a partial match, j>0, otherwise we restart from looking at first char in target, ie, j=0

- If there is a match, then j=j+1

KMP Example

- text="*ababacaa*", target="*abac*", N=8, M=4

- At i=3 there is a mismatch, as (target[3]=c) != (b=text[3])

- *But how does KMP knows that next "j" would be a 2?*

	0	1	2	3	4	5	6	7
	a	b	a	b	a	c	a	a
j	0							
j		1						
j			2					
j				3				
j					2			
j						3		

Deterministic Finite-State Automaton (DFA)

- States with *transitions* when reading chars

- Acceptance State (AS): if reached, we have a match

- DFA built for target, eg "*abac*", *regardless of the text to search in*
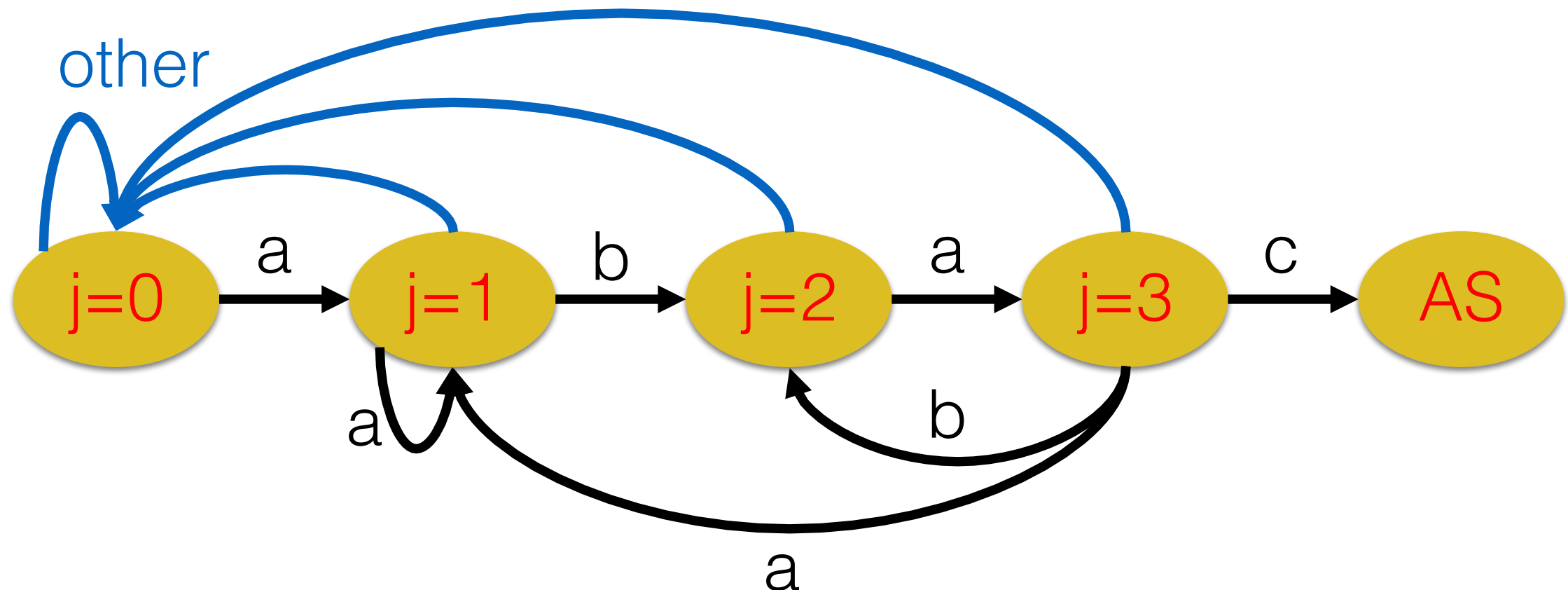
DFA: Matrix Representation

- For each state (columns) we specify what happens when we read a char (rows)

- Ie, cells contain the transitions to next state

"abac"	j=0	j=1	j=2	j=3
a	1	1	3	1
b	0	2	0	2
c	0	0	0	AS=4
"other"	0	0	0	0
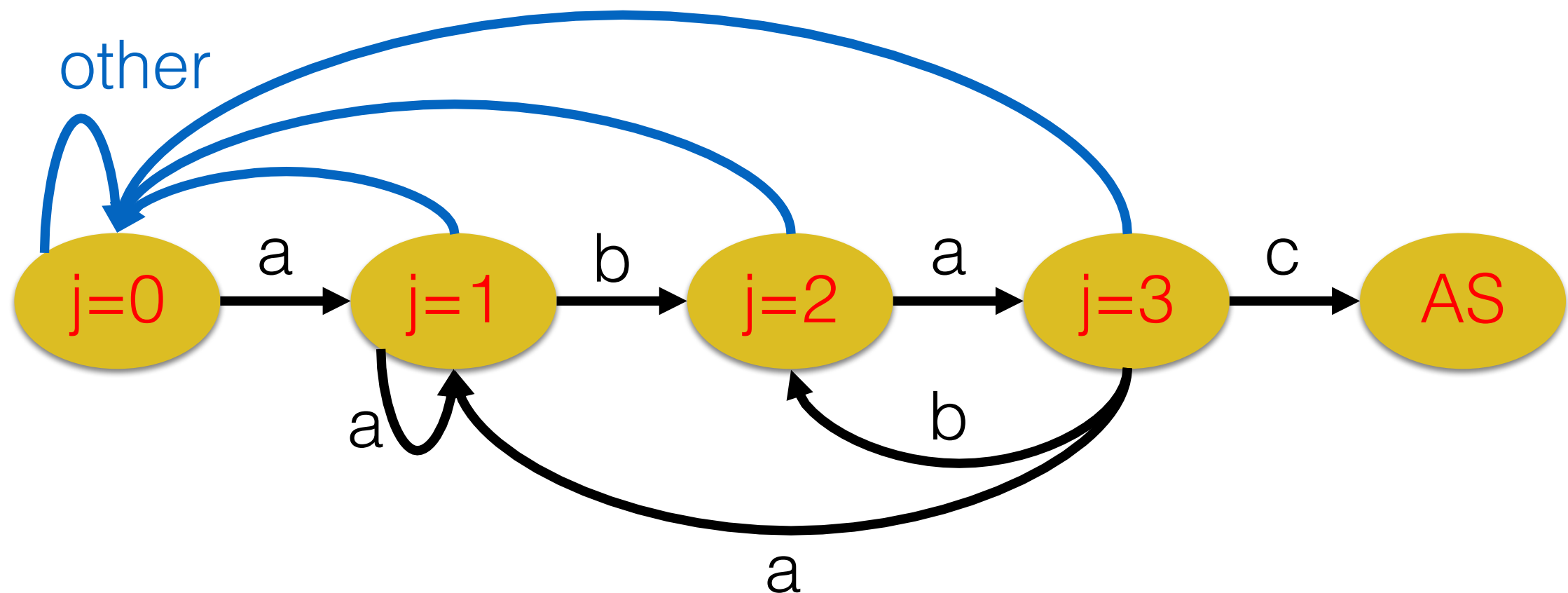
Building The Matrix

- One column at a time

- Starting from left to right

- All 0s, but a 1 for the matching char at position j=0

- Ie, we stay in j=0 unless we read the only matching char "a". As length 0, there cannot be any partial match at this point

"abac"	j=0	j=1	j=2	j=3
a	1			
b	0			
c	0			
"other"	0			
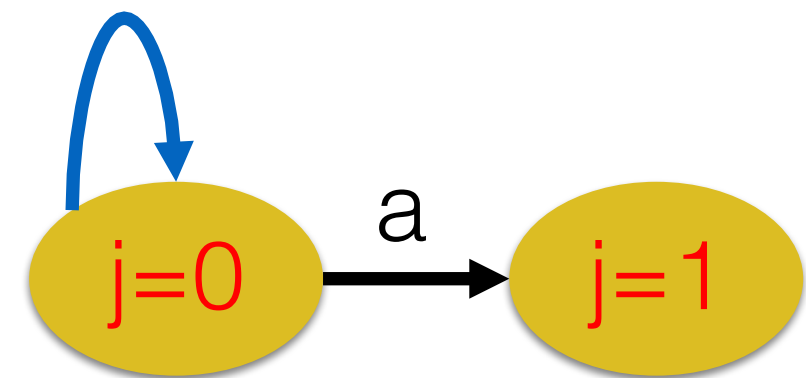
Second Step

- We keep a counter x, initialized to 0

- x represents the "restart" index for j when there is a mismatch based on current partial match

- So, we need to re-evaluate the current char as it was a restart (to check partial matches), ie a partial match starting at a smaller j

- So, $m[c][j]=m[c][x]$

"abac"	j=0	j=1	j=2	j=3
a	1	1		
b	0	0		
c	0	0		
"other"	0	0		

x=0   x=0
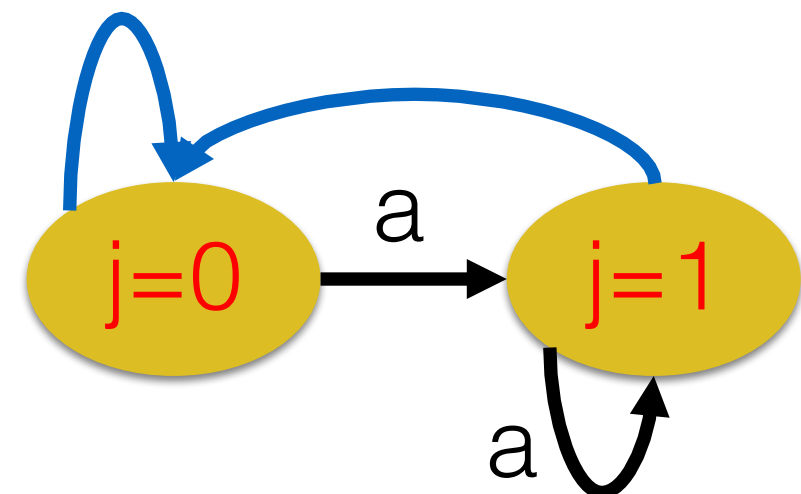
Second Step, Cont. 1

- When we have a mismatch for "a", that is a partial match for j=0,

- So, m["a"][1]=m["a"][x=0]

- Then, we need to update the match transition, which goes forward on next state

- Ie, m[match][j] = j + 1 , which in this case is m["b"][1]=1+1

"abac"	j=0	j=1	j=2	j=3
a	1	1		
b	0	2		
c	0	0		
"other"	0	0		

x=0   x=0
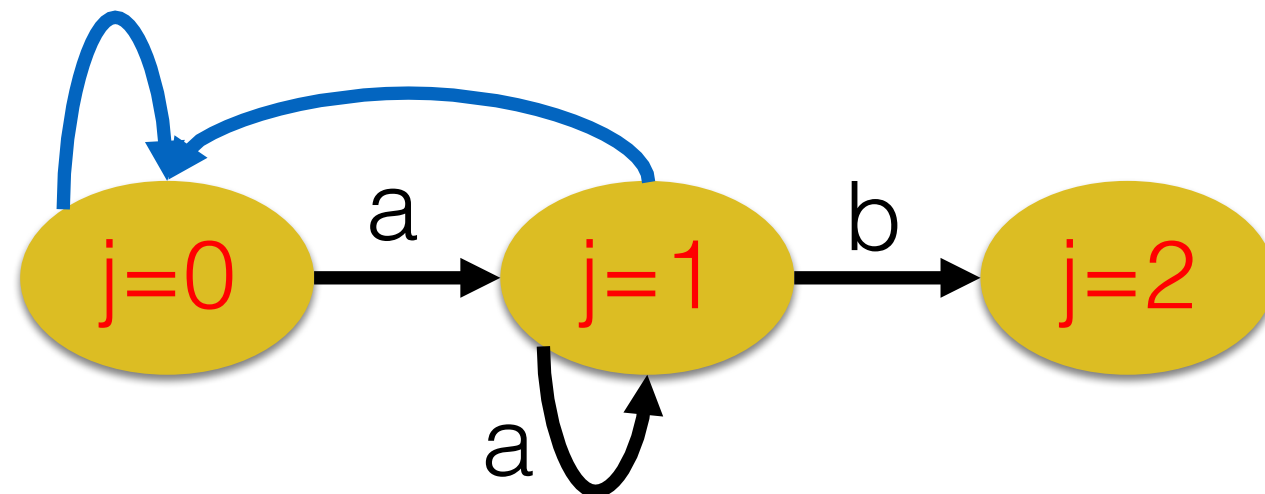
Second Step, Cont. 2

- Matrix is updated for j=1

- But need to update the restart counter x

- x = m[t[j]][x] , ie in our case m["b"][0] = 0

- This means that reading a "b" in the current partial match (which is empty x=0) would not be a valid continuation for that partial match, which so stays empty at x=0

"abac"	j=0	j=1	j=2	j=3
a	1	1		
b	0	2		
c	0	0		
"other"	0	0		

x=0   x=0   x=0

Third Step

- Like 2nd (and all following), using same rules

- m["a"][2] gets first a 1 due to m["a"][2]=m["a"][x] (to handle mismatch), but then a 3 due to m[match][j] = j + 1, as "a" is actually the matching at j=2

- x = m[t[j]][x] , so it gets updated, as m["a"][0] = 1

- Ie, an "a" at position 2 is also a match for starting a new partial match from 0

- So, when looking at next char, we are either continuing match of 4th char, or a partial match on the 2nd

"abac"	j=0	j=1	j=2	j=3
a	1	1	3	
b	0	2	0	
c	0	0	0	
"other"	0	0	0	

x=0   x=0   x=0   x=1
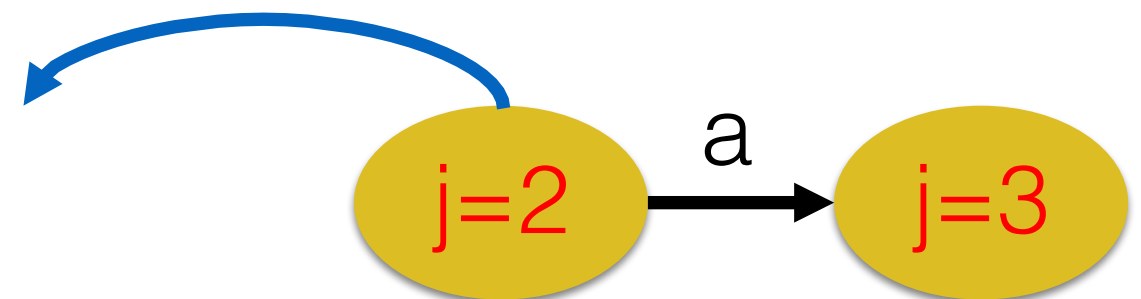
Fourth Step

- Due to x=1, the values from column j=1 are copied over

- As "c" is the matching char, it gets the transition to j=j+1, which is the acceptance state

- "b" would be match for partial "ab.." (as previous in j=2 was an "a")

- "a" would always be match for "a…" (ie, a new starting at j=0)

"abac"	j=0	j=1	j=2	j=3
a	1	1	3	1
b	0	2	0	2
c	0	0	0	AS=4
"other"	0	0	0	0

x=0   x=0   x=0   x=1
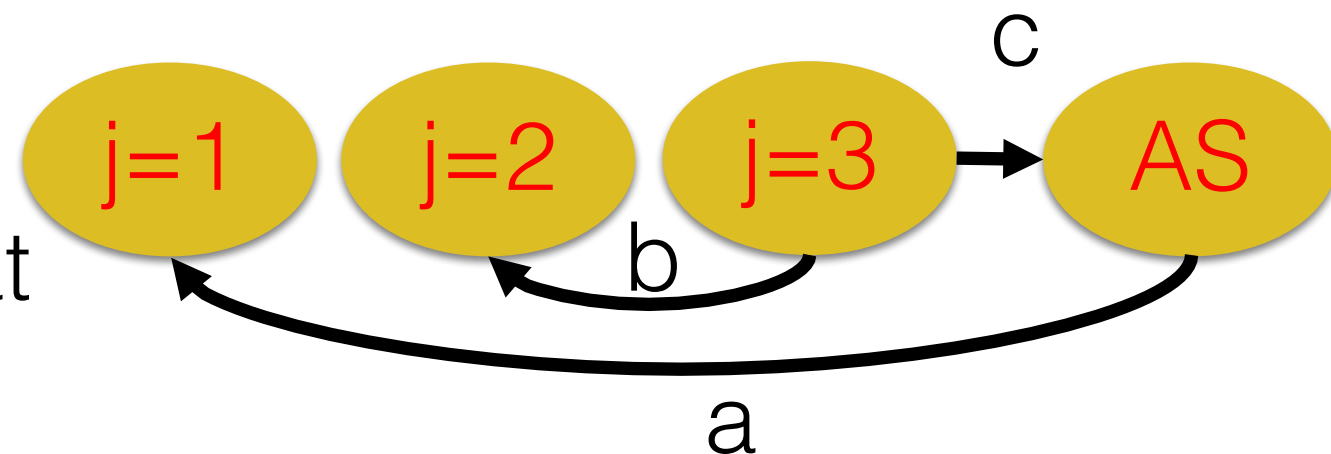
Cost

- Scanning the text is only *O(N),* as each of the N elements is looked up only once

- But need to build the matrix, which is *O(M)*
  - albeit possibly large, the number of rows is at most a constant, as there is only a finite alphabet of symbols

- So, total cost is *O(M+N)*, which is better than *O(M*N)* of brute force

- Furthermore, if we search for same target in many texts, we need to build the matrix *only once*

Regular Expressions

Regex ?

Regular **Ex**pression

- Besides searching for a specific string, perfectly matching char by char, you can define some *rules*

- Example: find all text within "<>" brackets

    - eg: lore ipsum **<*foo*>** lore ipsum

- Rules are defined with a *Regex,* which is a string itself

- Not just searching, but also checking if whole text does satisfy the regex

Constraints

- Is "*loreipsum.com*" a valid email address?

  - no, it does not have the symbol "@"

- Is "*3fenfrje35ddsre123345*" a valid telephone number?

  - no, it contains non-digits, and it is likely too long

- When a String represents a specify type of value (eg, emails), we can use a regex to specify its *constraints*

  - eg, the subset of all possible strings representing a valid email

Definition, Regex is either:

1. An empty set

2. An empty string

3. A single character

4. A regex enclosed in parentheses **()**

5. Two or more concatenated regexs

6. Two or more regexs separated by *or* operator **|**

7. A regex followed by the the closure operator *****

Matching Characters

- Wildcard "**.**" matches any character

  - eg, "a.b" match any 3-letter word starting with "a" and ending with "b"

- Set **[]** matches any single character in the set

  - eg, [abc] does match "a", "b" and "c", but not "d", nor "ab"

- Range **[-]** matches in the range

  - eg, [a-z] matches any lower case letter, [a-zA-Z] any letter, [0-8] any digit but 9

(), | and *

- **()** use to define boundary of a regex

- **|** is an or between two expressions

- ***** applies the previous regex 0 or more times

- Eg, "ab*" does match "a", "ab" and "abbbbbbb"

- Eg, "(ab)*" does match "", "ab", "abab" and "abababab"

- Eg, "(ab)|c" does match "ab" and "c"

Shortcuts

- "**+**" at least once
  - "x+" equivalent to "xx*"

- "**?**" zero or one time
  - "x?" equivalent to "emptyString | x"

- "**{}**" specific number of times
  - "x{5}" equivalent to "xxxxx"
  - "x{2,4}" equivalent to "(xx)|(xxx)|(xxxx)"

Limitations of Regex

- Regex are very useful in many contexts to check the validity of strings that are supposed to have constraints

- However, they are not fully expressive, as there are constraints you cannot express with a regex

- Example: you cannot use a regex to check if a string is a valid Java code

  - for that, you need a Context-Free Grammar, but we will not see them in this course

Do Not Use Regex for HTML!!!

… Regex-based HTML parsers are the cancer that is killing StackOverflow it is too late it is too late we cannot be saved the trangession of a child ensures regex will consume all living tissue (except for HTML which it cannot, as previously prophesied) dear lord help us how can anyone survive this scourge using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes using regex as a tool to process HTML establishes a breach between this world and the dread realm of corrupt entities (like SGML entities, but more corrupt) a mere glimpse of the world of regex parsers for HTML will instantly transport a programmer's consciousness into a world of ceaseless screaming, he comes, the pestilent slithy regex-infection will devour your HTML parser, application and existence for all time like Visual Basic only worse he comes he comes do not fight he comes, 'his unholy radiańcé destroy͘ing all enlightenment, HTML tags leaking from your eyes͟ l'ike liquid pain, the song of regular expression parsing will extinguish the voices of mortal man from the sphere I can see it can you see it it is beautiful the final snuffing of the lies of Man ALL IS LOST ALL IS LOST the pony he comes he comes he comes the ichor permeates all MY FACE MY FACE ͡oh god no NO NOO͡OO NΘ stop the angles are not real ZALGO IS TONY THE PONY HE COMES

Building A Regex

- Each Java String has a method *".matches()"* that takes as input a regex

- However, important to understand how regexs are implemented

- We will not see whole implementation considering all operators, but just a minimal working subset

Example

- Regex: **(a*I(bc)*)z**

- Matching: "z", "az", "aaaaaaz", "bcz", "bcbcbcz"

- When we read a first "a", there are two valid alternatives: either we get another "a", or we get a "z" (but only if it was the last char to scan)

- As for KMP, for the matching we need a *finite state machine*

- However, we need a *non-deterministic* one, because at each step several different paths could be taken

- We need to keep track of all possible paths that could be valid

- This is due to operators like **I** and *****

- *Empty-transitions* are transitions on the graph that can be taken without reading data from text, ie transitions related to the meta-characters

- We do not need to *explicitly* represent transitions when parsing and consuming chars from the text, eg transition from "b" to "c", and "z" to AS
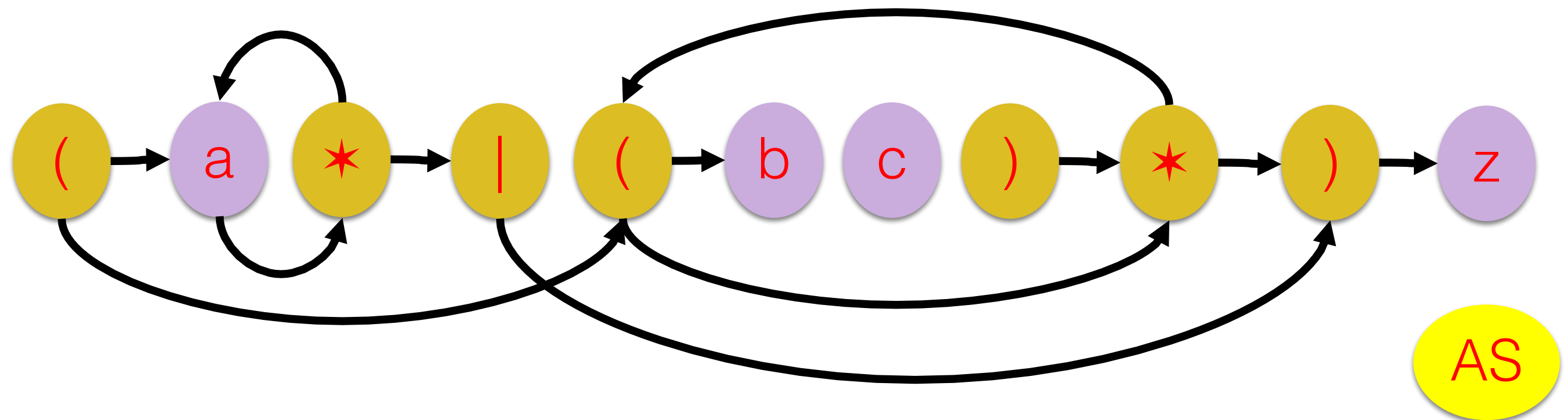
First Step When Matching

- I can navigate the graph to look at all possible states I could be non-deterministically be in

- Ie, expecting to read "a", "b" or "z"

- Paths: "**(-> a**", "**(-> (-> b**", "**(-> (-> * ->) -> z**", "**(-> a -> ✱ -> l ->) -> z**"

- Note: that "z" can be reached with two different paths
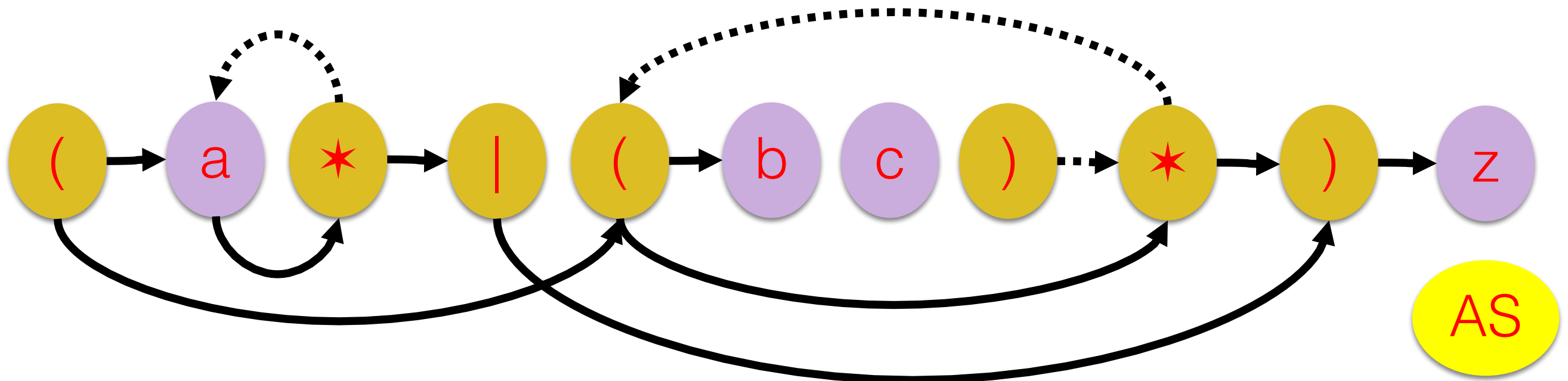
Matching "**aaz**"

- Before reading first char, we are non-deterministically in 3 states, with indices 1 ("a"), 5 ("b") and 10 ("z")

- Reading "a" does match only one of them, which leads to an actual transition (not in the graph) to the next element 2 (✱)
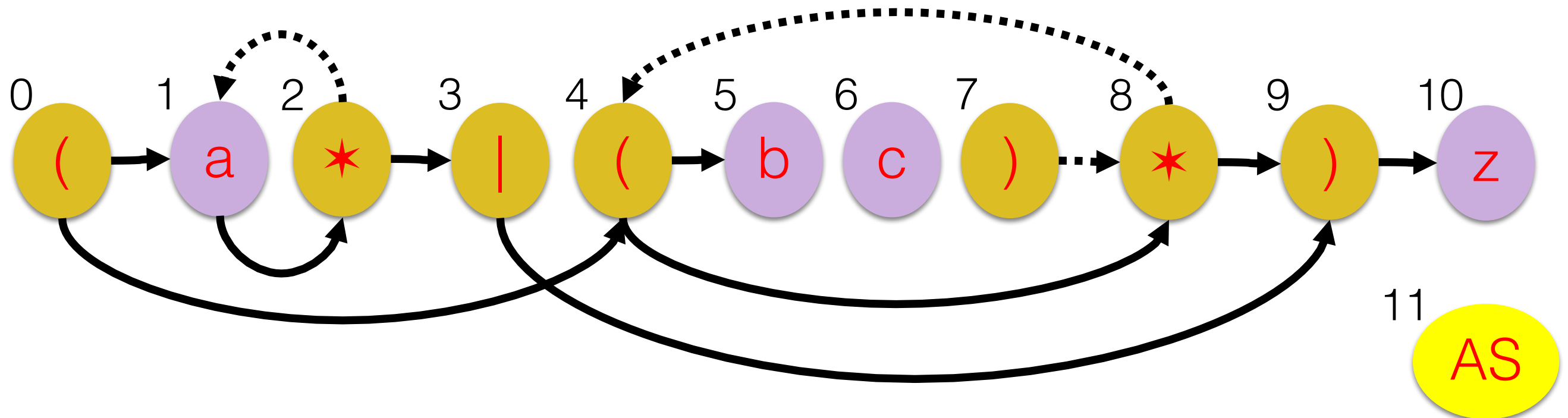
Matching "**aaz**"

- From state 2 (✱) we can only reach "a" again, and "z", but not "b" nor "c"

- Reading "a" again will leave us in the same condition, ie being in state 1 ("a") and 10 ("z")

- Reading finally "z" is match only for 10, not 1, and so we have a transition to AS

- As we have read the whole text and we have at least one path in AS, then the input does match the regular expression
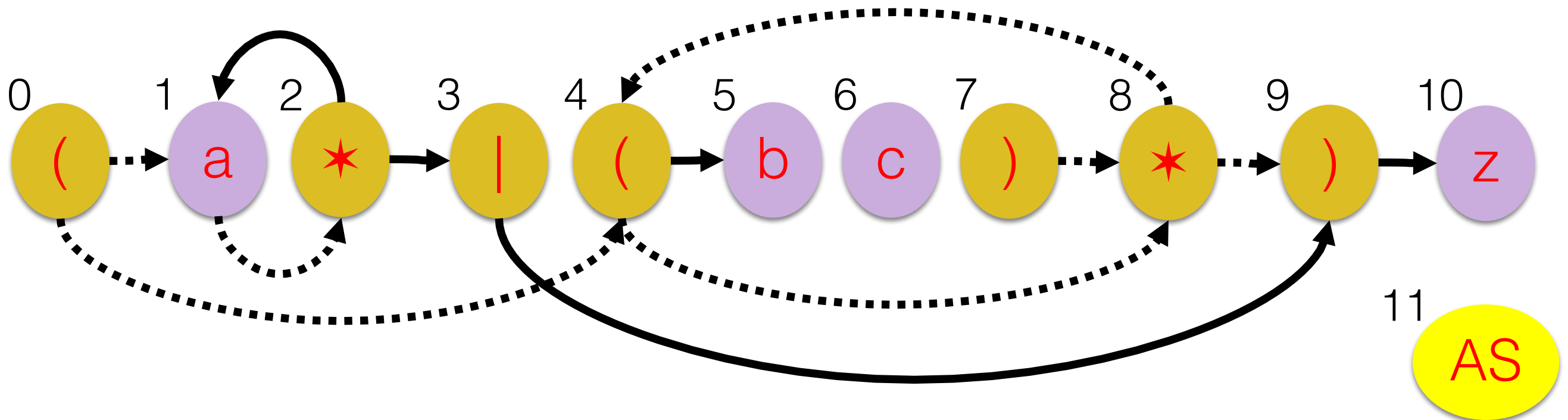
Matching "**bk**"

- Before reading first char, we are non-deterministically in 3 states, with indices 1 ("a"), 5 ("b") and 10 ("z")

- Reading "b" does match only one of them, which leads to an actual transition (not in the graph) to the next element 6 ("c")
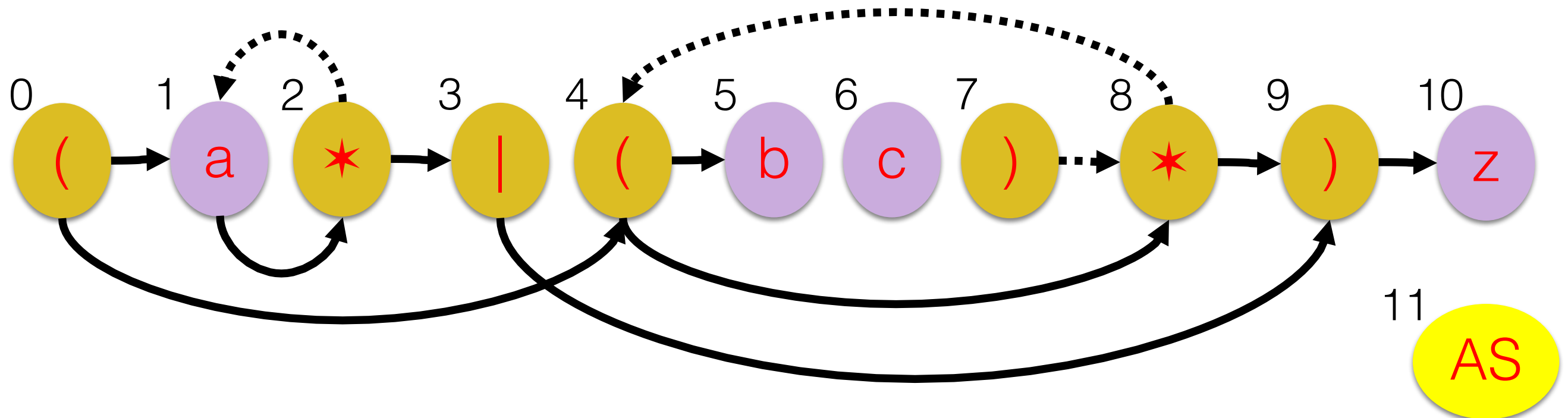
Matching "**bk**"

- From state 6 ("c") there is no empty-transition we can traverse for free

- The only way to exit is to read a "c" as next char

- But as we read a "k" next, there is no match, and path is abrupted

- As we are not non-deterministically in any other state, we know that the input is not a valid match
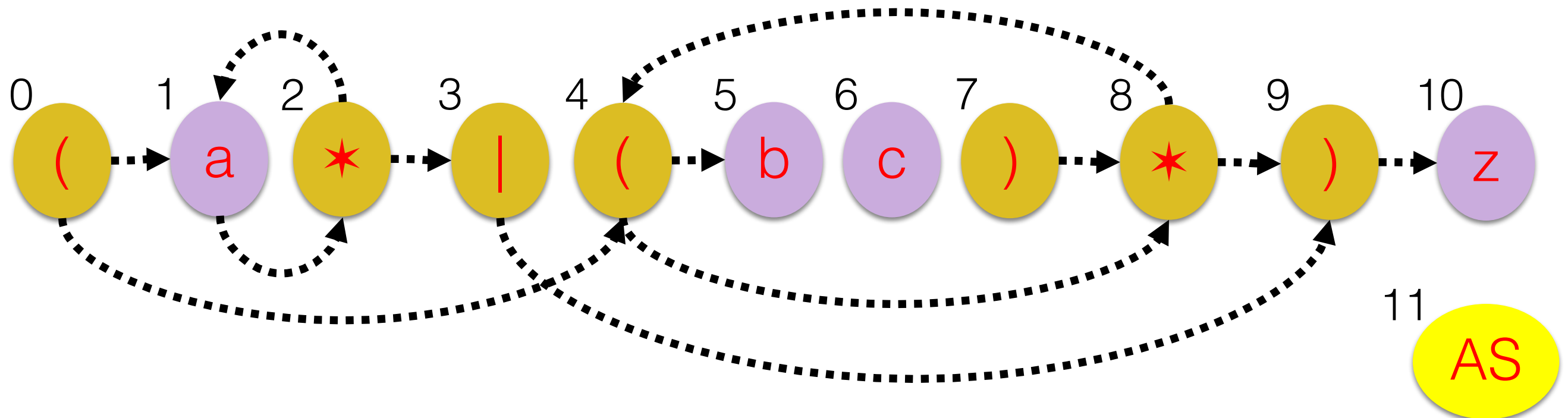
Implementation Phases

1. Build a non-deterministic finite state machine for the empty-transitions using a direct graph

2. Read one char at a time from the input, and calculate all possible valid states in which we could possibly be

- Build one state per char in the regex, plus Acceptance State

- "**(**", "✱" and "**)**" will have a empty-transition forward

(→ a ✱ → | (→ b c) → ✱ →) → z

AS
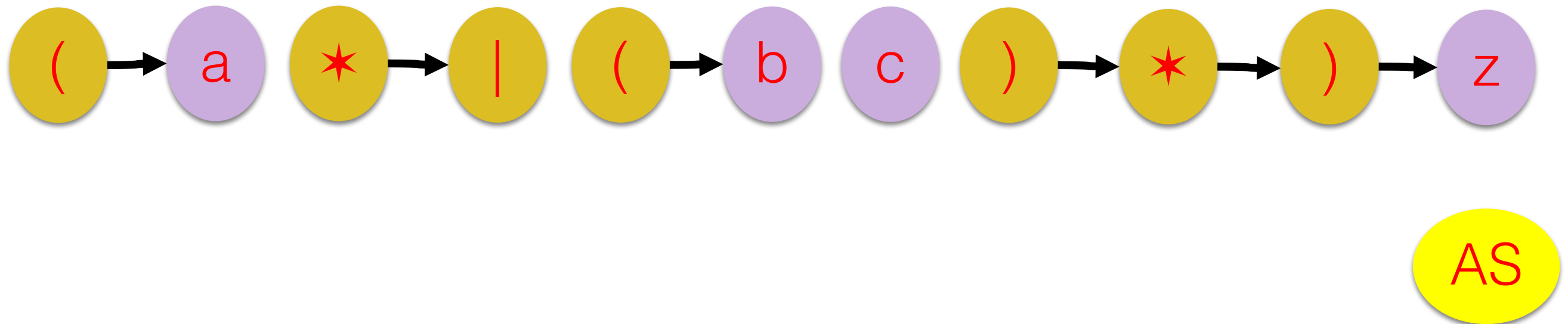
- When we have a "**|**", the enclosing "**(**" will have an empty-transition to the state after "**|**", whereas "**|**" will an empty-transition to the closing "**)**"
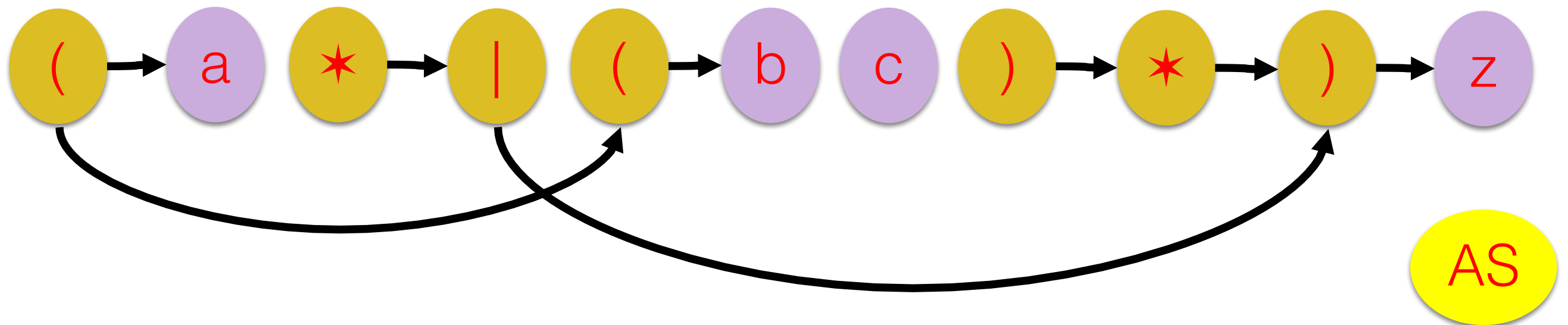
- Handling "∗" is tricky, we need a transition back to previous *block,* and the *beginning* of previous block need transition to "∗"

- Why *tricky?* Because previous block could be composed of arbitrary number of nested **()** parentheses, and we could be inside a block not closed yet

- We have to use a **STACK** as *temporary* data structure to keep track of how many "**(**" are still open
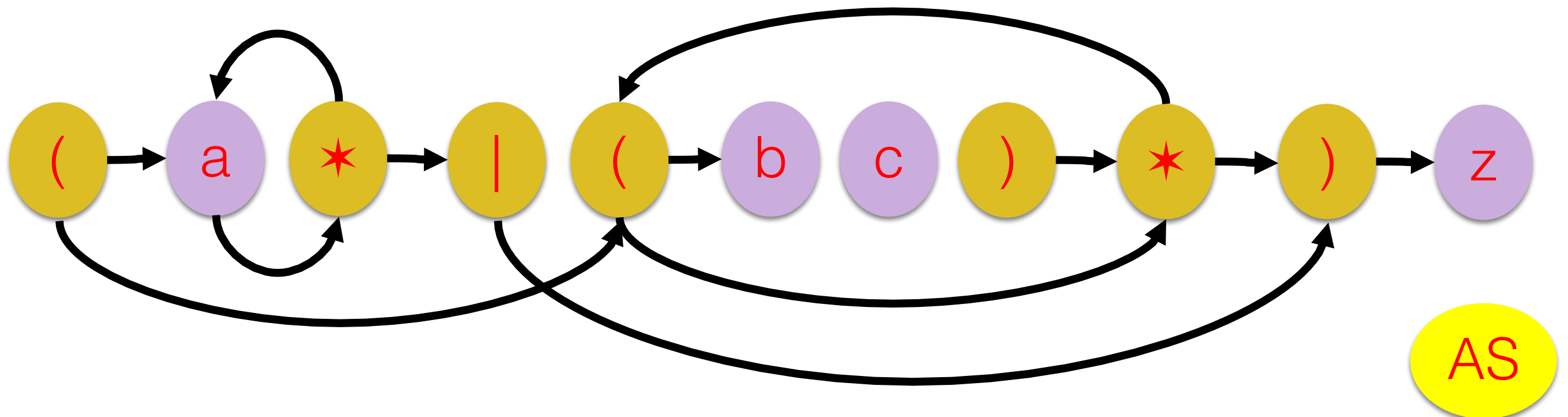
Traversing The Graph

- From state 0, find all connected states on the graph

- Read a char, and see if it matches any of these states x
  - If yes, add x+1 to the next states to consider in the path
  - If no, exclude such state

- From all the x+1 matched states, add all connected states

- Repeat until either 0 matches, or read whole input

- If read whole input and in AS, then there is a match

Homework

- Study Book Chapter 5.3 and 5.4

  - but not Boyer-Moore, nor Rabin-Karp

- Study code in the *org.pg4200.les09* package

- Do exercises in *exercises/ex09*

- Extra: do exercises in the book