Review Q&A

COMP 524: Programming Language Concepts
Björn B. Brandenburg

The University of North Carolina at Chapel Hill

Suppose you are stranded on a desert island with a computer (with an assembler and a basic OS) and need a compiler for a **high-level language** such as Haskell. How do you obtain one?

By **bootstrapping**. Start by building a minimal working interpreter for a subset of Haskell (e.g., no type checking/inference, no polymorphism, no module support, etc.) in assembly.

Write a compiler for Haskell in the chosen subset of Haskell.

Interpret compiler#1 with the basic interpreter to produce a self-hosting compiler (compiler#1 can compile compiler#1).

Now iterate by copying the source of compiler#1 to make a compiler#2 with some features added. Compile compiler#2 with compiler#1.

Iterate until compiler#n is a fully self-hosting, feature-complete Haskell compiler (might take a while...).

Still on the desert island.
Suppose we have a working, self-hosting Haskell compiler on our rescue pod computer (an Intel x86 machine), but found a PowerPC computer in a satellite wreck, and would like to have a working, self-hosting Haskell compiler on the satellite computer.

By **cross-compiling**.

1) Start by modifying the existing Haskell compiler that can generate x86 machine code to also be able to generate PowerPC machine code.

2) Now, **compile the modified compiler with itself** to produce a compiler that runs on x86 but produces PowerPC machine code.

3) Use the compiler produced in step 2), which runs on x86 but produces PowerPC code, and compile itself again. This time, the result is a compiler that runs on PowerPC and that produces PowerPC machine code.

We went from a x86->x86 compiler to a PowerPC->PowerPC compiler.

What are the first two phases of a compiler?

Lexical analysis and syntax analysis.

What's the purpose of lexical analysis?
What's the purpose of syntax analysis?

Lexical analysis: turn stream of characters into stream of tokens (group characters by meaning)
Syntax analysis: infer structure of program from token stream.

Why have a separate lexical analysis phase?

Because tokens can be described with regular grammars, which can be recognized much more efficiently than more flexible grammars.

(Regular grammars cannot describe arbitrary recursive structures.)

How can we recognize regular grammars?

With DFAs (deterministic finite automata).

How can we construct a DFA from a regular expression (regular grammar)?

RE->NFA->DFA (unoptimized)->DFA (optimized)

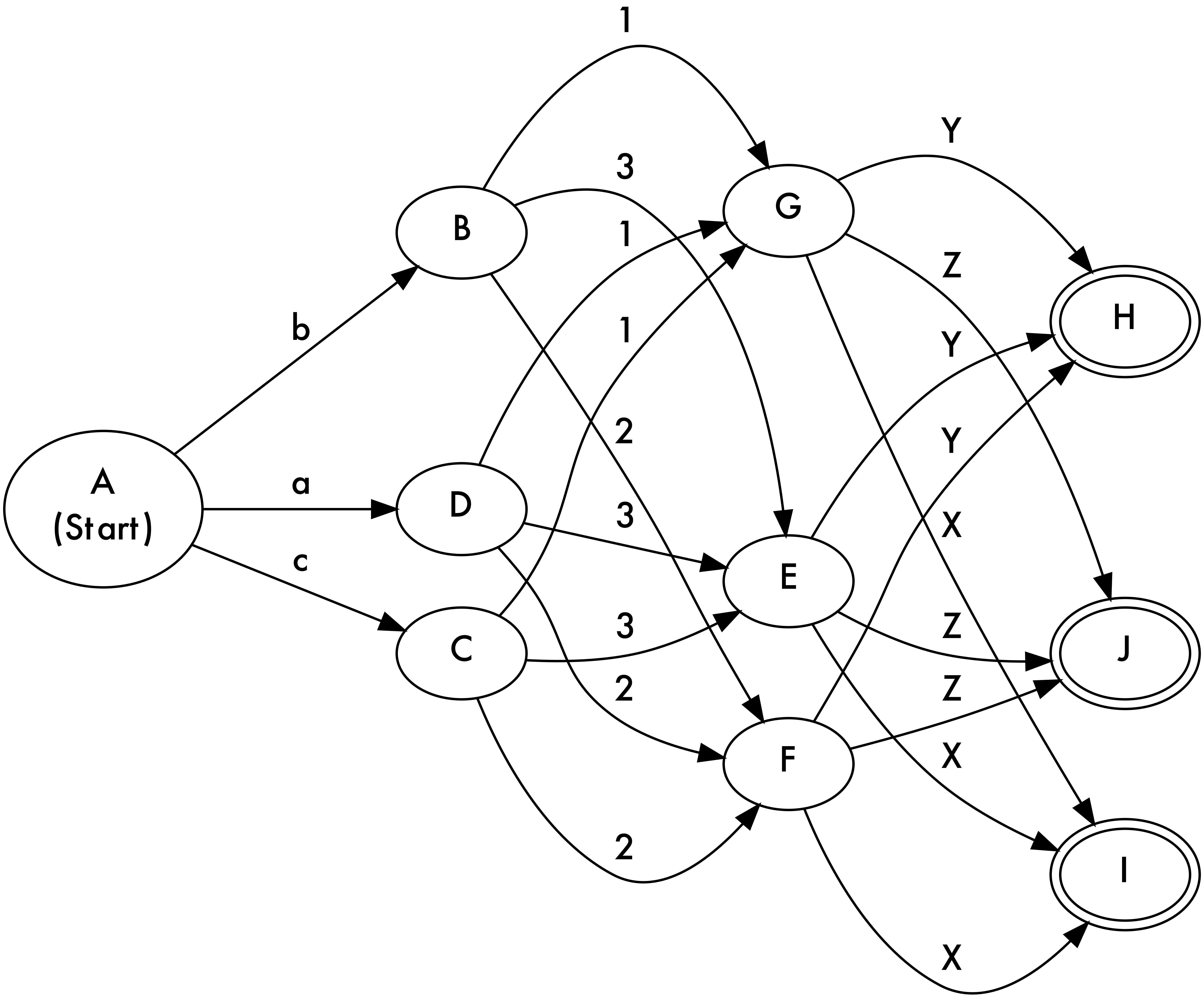
Why do we need DFA optimization?

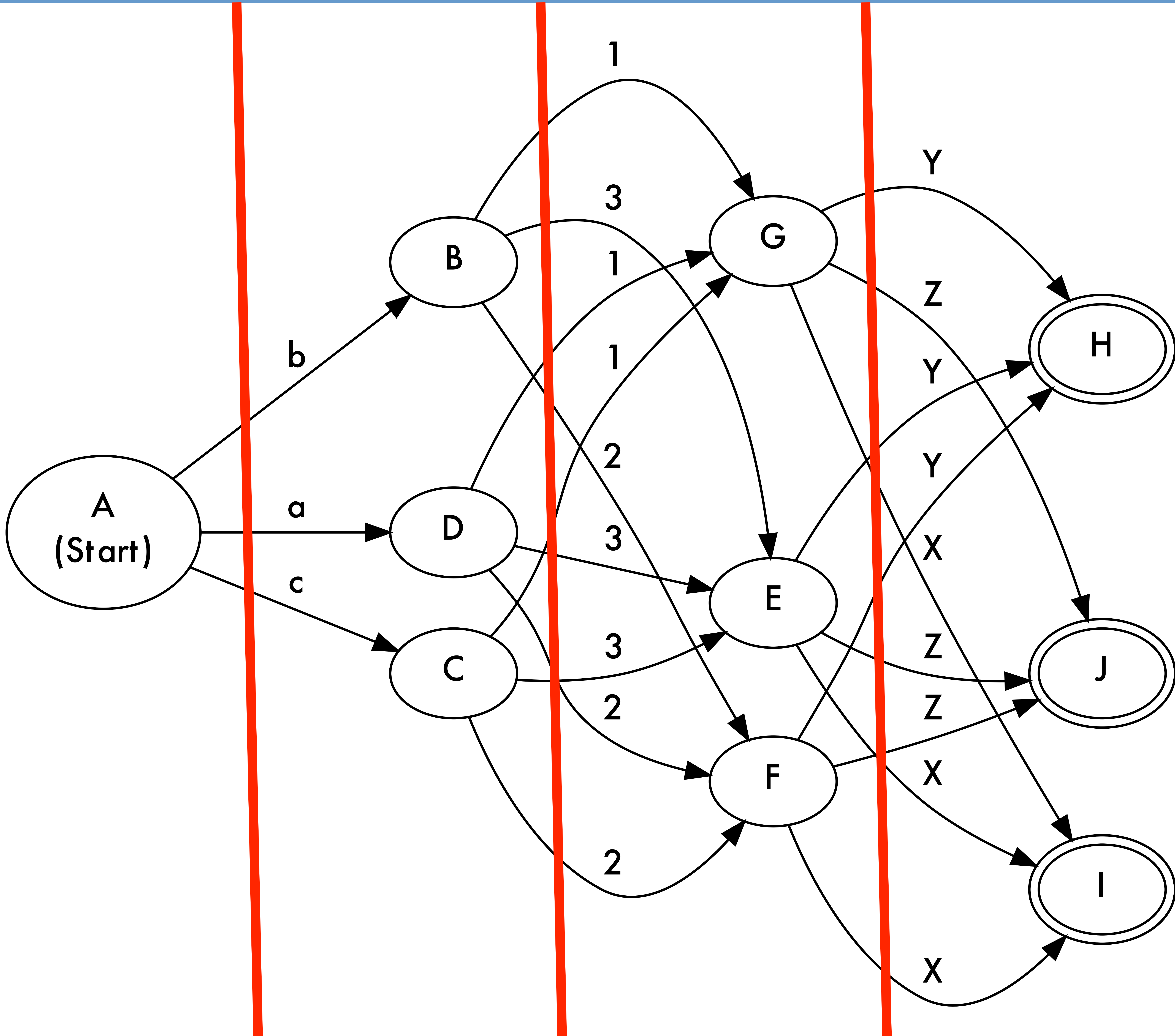
Because unoptimized DFAs produced by the NFA->DFA conversion can have a large number of redundant states.

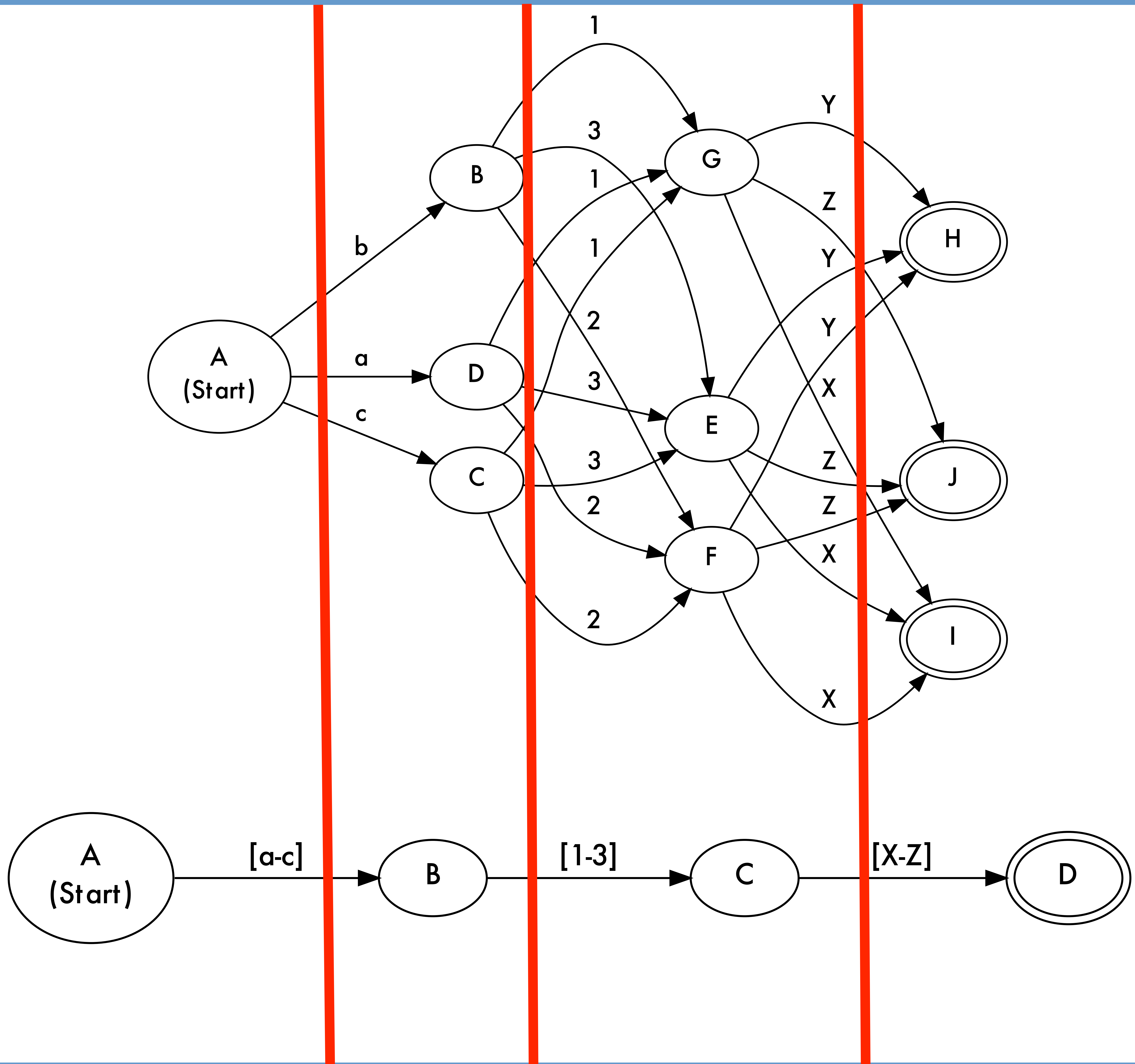
How does DFA optimization work?

By dividing non-equivalent states into partitions.

1) start with a partition for all non-final states and one partition for all final states (for each token type, if there are multiple).
2) Sub-divide partitions while they are not equivalent.
3) The DFA has been optimized when none of the partitions has to be split anymore (contains only equivalent states).
4) Create an equivalent DFA by mapping each partition to a state.


A
(Start)
b
a
c
B
D
C
1
3
1
1
2
3
3
2
2
G
E
F
Y
Z
Y
Y
X
Z
Z
X
X
H
J
I


1
Y
3
G
B
1
Z
H
b
1
Y
2
Y
A
(Start)
a
D
3
X
c
E
Z
C
3
J
2
Z
F
X
2
I
X


A
(Start)
b
a
c
B
D
C
1
3
1
1
2
3
3
2
2
G
E
F
Y
Z
Y
Y
X
Z
Z
X
X
H
J
I
A
(Start)
[a-c]
B
[1-3]
C
[X-Z]
D

In a nested subroutine, what is a free variable?

Any variable that is neither a formal parameter (passed to the function) nor a local declaration.

What is a closure?

A nested subroutine in which the free variables are bound to entities (objects) residing in the lexical scope in which the nested subroutine was defined.

```
def outer_function():
 a_list = [1, 2, 3]

 def nested_function(msg):
  print 'hello', msg, 'the list is', a_list

 return nested_function


a_list = [99, 100, 101]

f = outer_function()

f("world")
```

```
def outer_function():
 a_list = [1, 2, 3]

 def nested_function(msg):
  print 'hello', msg, 'the list is', a_list

 return nested_function


a_list = [99, 100, 101]

f = outer_function()

f("world")
```

A closure in Python: the free variable a_list remains bound to [1,2,3], which was in scope at the time of definition, not at the time of call.

What is an anonymous function?

Simply a function that is not bound to a name.

Anonymous functions are not necessarily nested, and not necessarily a function parameter.

```
>>> (lambda x: x * x)(2)
4
```

An anonymous function in Python.
(not nested, not a parameter)

What's the difference between call-by-name and inlining?

Call-by-name is a function call semantics (that could be implemented in different ways). It defines how parameters are used.

Inlining is a compiler optimization (that may not change the calling semantics). It applies to how a function is called.

Inlining can apply to any function call semantic.

How many students did actually submit review questions?

Two; less than ten percent of the students…