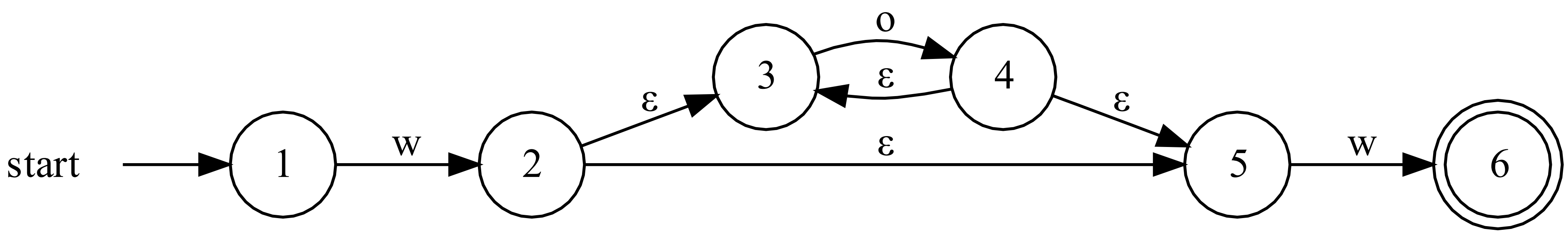
Little Languages

Lecture 21:

Modeling NFAs

We will be using 590-materials on VM today.

Graphs of Pointers in Unmanaged Languages



- Suppose each of the edges in the transition diagram above is a pointer
- In Rust ownership terminology, who *owns* nodes 3 and 4?
- Graphs of pointers with cycles require careful consideration when freeing memory after the data structure is no longer needed. Why?

Region-based Memory Management "Arenas"

- Rather than thinking about allocating and deallocating in terms of individual *nodes*, region-based management reframes the problem.

- Key Applicability Questions
  - Once initialized, does your data structure need to be able to expand/contract?
  - Do elements of the data structure need lifetimes independent of the structure's?

- If no to those questions, then memory management is simplified by allocating a contiguous region for the entire structure rather than individually per node.
  - This region or "arena" can then be deallocated all at once in one step.

- Our Motivation: To implement a regular expression engine, once our NFA graph is initialized it's final and we do not need nodes for longer than the graph.

Vectors as a simple Arena Allocator

- Rust has libraries for assisting with Region-based / Arena Allocation
  - Rather than learning their nuances, we'll employ a rudimentary approach: a Vector.

- Our memory "Arena" will be a Vector of States
  - Thus, each state has an identifier ("id") that is its index in the vector.
  - States will refer to each other via this identifier rather than by memory address.

- This added level of indirection has trade-offs. Fundamental ones:
  1. Cost: Indirection. Lookups must compute address with vector start + id offset * size.
  2. Benefit: Locality. All states are located nearby each other in a region of memory.

State Node Primitives in Thompson's Construction

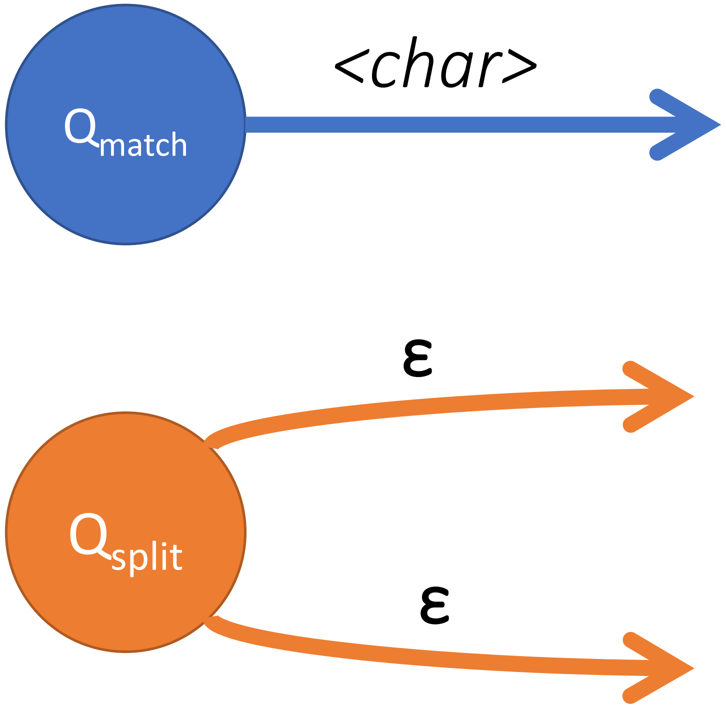
- Theoretical NFAs have *no constraints* on the numbers of edges relating nodes nor the use of ε-transitions.

- **Big CS Idea: Representing abstract concepts with no constraints is made tractable by designing highly constrained, yet easily composed primitives.**

- The genius of Thompson's Construction is its distillation of the representation: Only two kinds of primitive States are needed to composed *any* NFA.

- Source: https://www.fing.edu.uy/inco/cursos/intropln/material/p419-thompson.pdf

State Primitives in Thompson's NFA Construction



A **Match** State matches a single char.

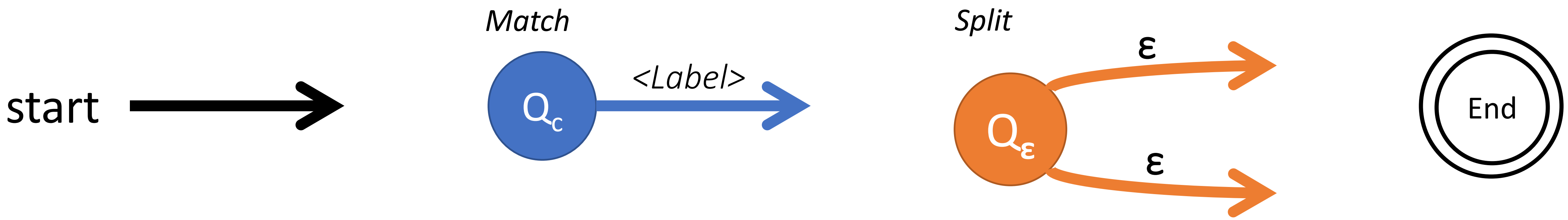
(Thompson's "NNODE")

A **Split** State splits the search path.

(Thompson's "CNODE")

- From these 2 fundamental states you can compose any NFA!
  - We'll also have trivial sentinel states for start/end.

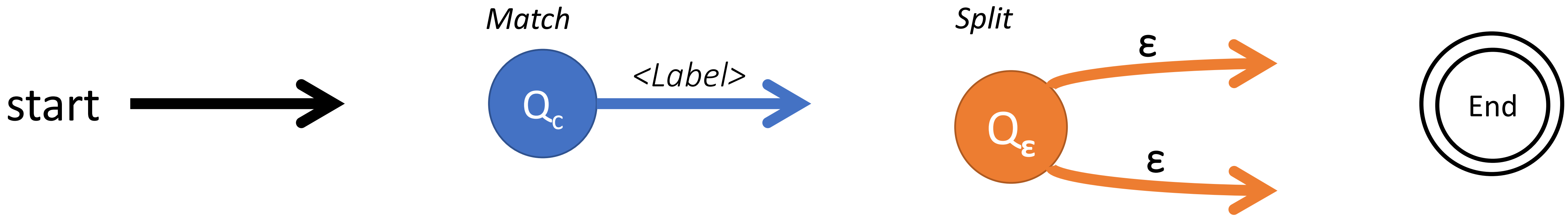
Hands-on: Draw a transition diagram for **a(b|c)*d** using only these State primitives:



Modeling in Rust

```rust
type StateId = usize;

enum State {
    Start(Option<StateId>),
    Match(Label, Option<StateId>),
    Split(Option<StateId>, Option<StateId>),
    End,
}
```



- Note **StateId** is simply a type alias for the vector index of any State in our arena.

Code Walk: Let's Explore the Skeleton Code

- NFA struct is simply a Vec<State> and starting StateId (0)
  - It has an **add(s: State)** method that takes ownership of the State, pushes it into the Vec, and returns its StateId
  - It also has a **join(from: StateId, to: StateId)** method that replaces a dangling **None** edge of ***states[from]*** with ***Some(to)***.
  - When joining a **Split** state, it only joins the 2nd StateId tuple member and assumes the 1st is always known (and in Thompson's construction, it is).

- State Enum (shown previously)
- Char Enum (either Literal(char) or Any)
- Helper functions for debugging:
  - nfa_dump generates a string representation of the NFA's States
  - nfa_dot generates a dot GraphViz representation of the NFA

Figure 1 shows the functions of the third stage of the compiler in translating the example regular expression. The first three characters of the example a, b, c, each create a stack entry, $S[i]$, and an NNODE box.

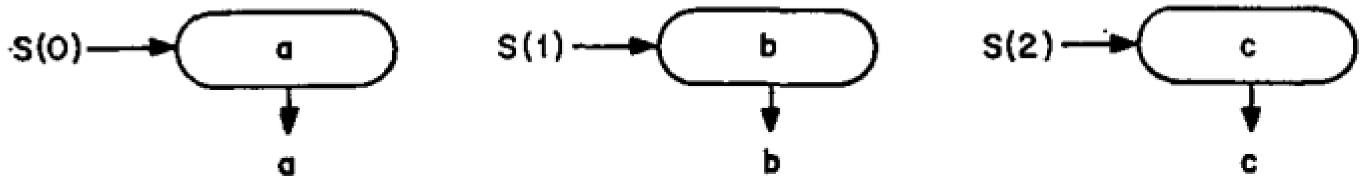


FIG. 1

Source: Thompson '68. Communications of the ACM.
Programming Techniques: Regular expression search algorithm.
https://www.fing.edu.uy/inco/cursos/intropln/material/p419-thompson.pdf

The next character "*" combines the operands b and c with a CNODE to form $b|c$ as an operand. (See Figure 2.)

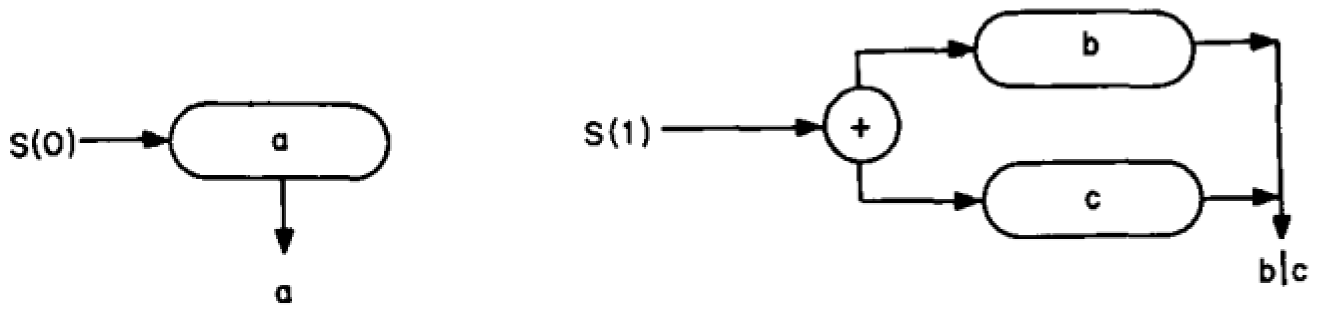


FIG. 2

Source: Thompson '68. Communications of the ACM.
Programming Techniques: Regular expression search algorithm.
https://www.fing.edu.uy/inco/cursos/intropln/material/p419-thompson.pdf

The next character "$*$" operates on the top entry on the stack. The closure operator is realized with a CNODE by noting the identity $X* = \lambda|XX*$, where X is any regular expression (operand) and λ is the null regular expression. (See Figure 3.)

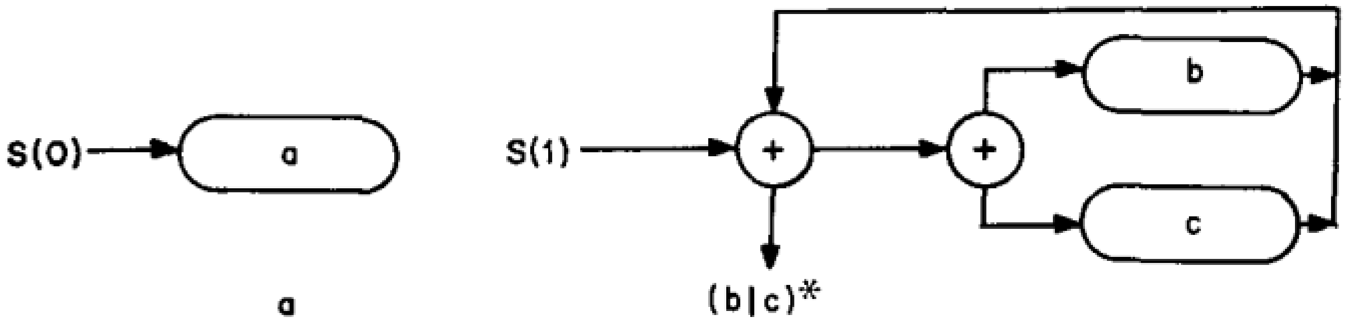


FIG. 3

Source: Thompson '68. Communications of the ACM.
Programming Techniques: Regular expression search algorithm.
https://www.fing.edu.uy/inco/cursos/intropln/material/p419-thompson.pdf

The next character "·" compiles no code, but just combines the top two entries on the stack to be executed sequentially. The stack now points to the single operand $a \cdot (b|c)*$. (See Figure 4.)

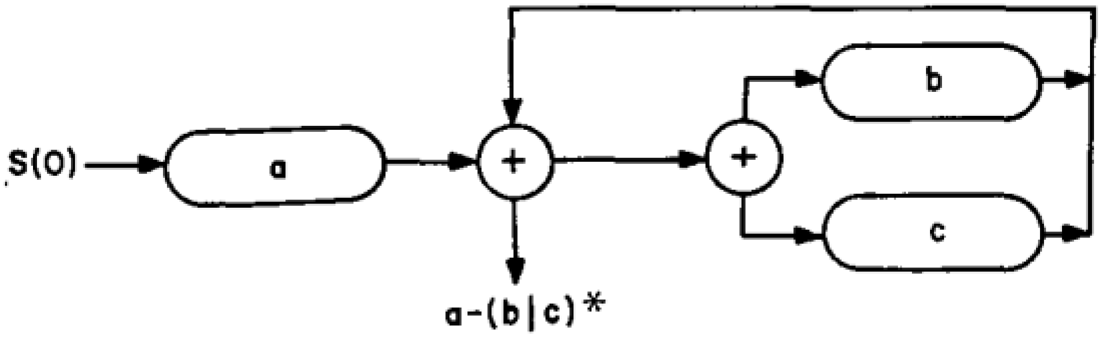


FIG. 4

Source: Thompson '68. Communications of the ACM.
Programming Techniques: Regular expression search algorithm.
https://www.fing.edu.uy/inco/cursos/intropln/material/p419-thompson.pdf

The final two characters $d\cdot$ compile and connect an NNODE onto the existing code to produce the final regular expression in the only stack entry. (See Figure 5.)

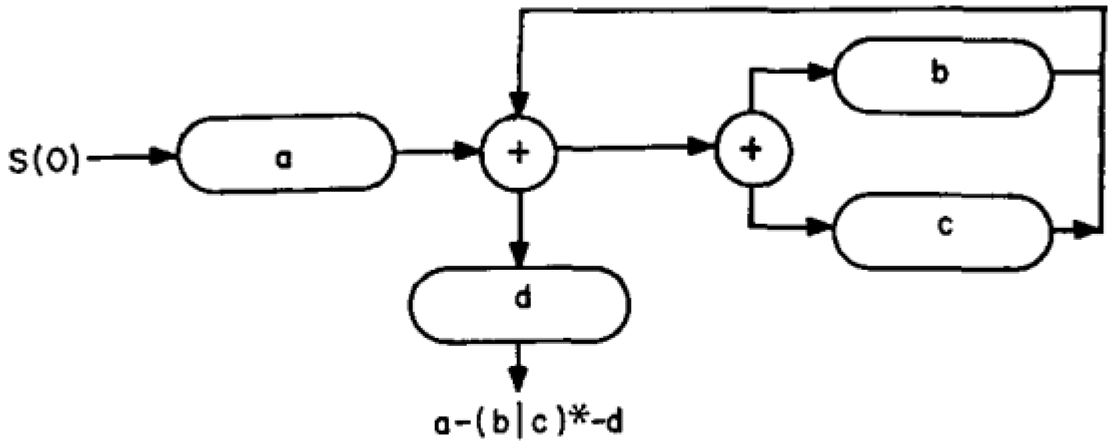


FIG. 5

Source: Thompson '68. Communications of the ACM.
Programming Techniques: Regular expression search algorithm.
https://www.fing.edu.uy/inco/cursos/intropln/material/p419-thompson.pdf

```
// a
let a = m.add(Match(Char::Literal('a'), None));
// b
let b = m.add(Match(Char::Literal('b'), None));
// c
let c = m.add(Match(Char::Literal('c'), None));
// |
let b_or_c = m.add(Split(Some(b), Some(c)));
// *
let star = m.add(Split(Some(b_or_c), None));
m.join(b, star);
m.join(c, star);
// ·
m.join(a, star);
// d
let d = m.add(Match(Char::Literal('d'), None));
// ·
m.join(star, d);
// Finalize by connecting start and end
m.join(m.start, a);
let end = m.add(End);
m.join(d, end);
```