Sequential Consistency & `perl` & *Perl*

About

- cPanel / WebPros
- Houston.pm.org – join us for zooms!

History

- Perl “FLAT” - Formal Language/Automata Toolkit
- Uploaded to CPAN in 2006
- Has waited since the for a real use case
- Interest in providing “concurrent” semantics in Perl that is run on a single process (perl) has revived this tool

Overview

- Perl (the language) readily admits concurrent semantics that a Human can use easily to reason – in fact, it ***begs*** for it
- `perl` (the interpreter/runtime) is strictly a *uniprocess* that admits **no** lightweight or shared memory concurrency
- This causes *cognative dissonance*, which is harmful to both indivduals and communities because it causes real conflict
- Reconciliation can happen if the right approach is taken
- The purpose of this talk is to demonstrate how concurrent shared memory semantics can be used for programming in a uniprocess environment (`perl` runtime)

Shared Memory Concurrency

- Communication is implicit since it depends on shared *memory* state; contrast this with explicit massage passing approaches
- Allows very complex interactions among concurrent lines of execution (threads) to be described implicitly, this means less code; less code means less bugs, etc.
- Is there value in being able to write programs for `perl` that assume implicit shared memory communications; but can run "correctly" in a uniprocess? I believe, *yes*.

Motivation

"to have all the fun of traditional shared memory programming, but inside of a single process"

- **Express concurrency naturally in Perl**
- **...that can leverage the strengths of `perl`**
- **...including its implicit memory model**
- **...and `state`ful subroutines**
- **...and do it in a fun and interesting way**

Motivating Example

Using Sub::Genius

```
#
# Implements classic JAPH :-)
#

my $pre = q{
begin
(
  J &
    A &
      P &
        H
)
end
};
```

More on this later ...

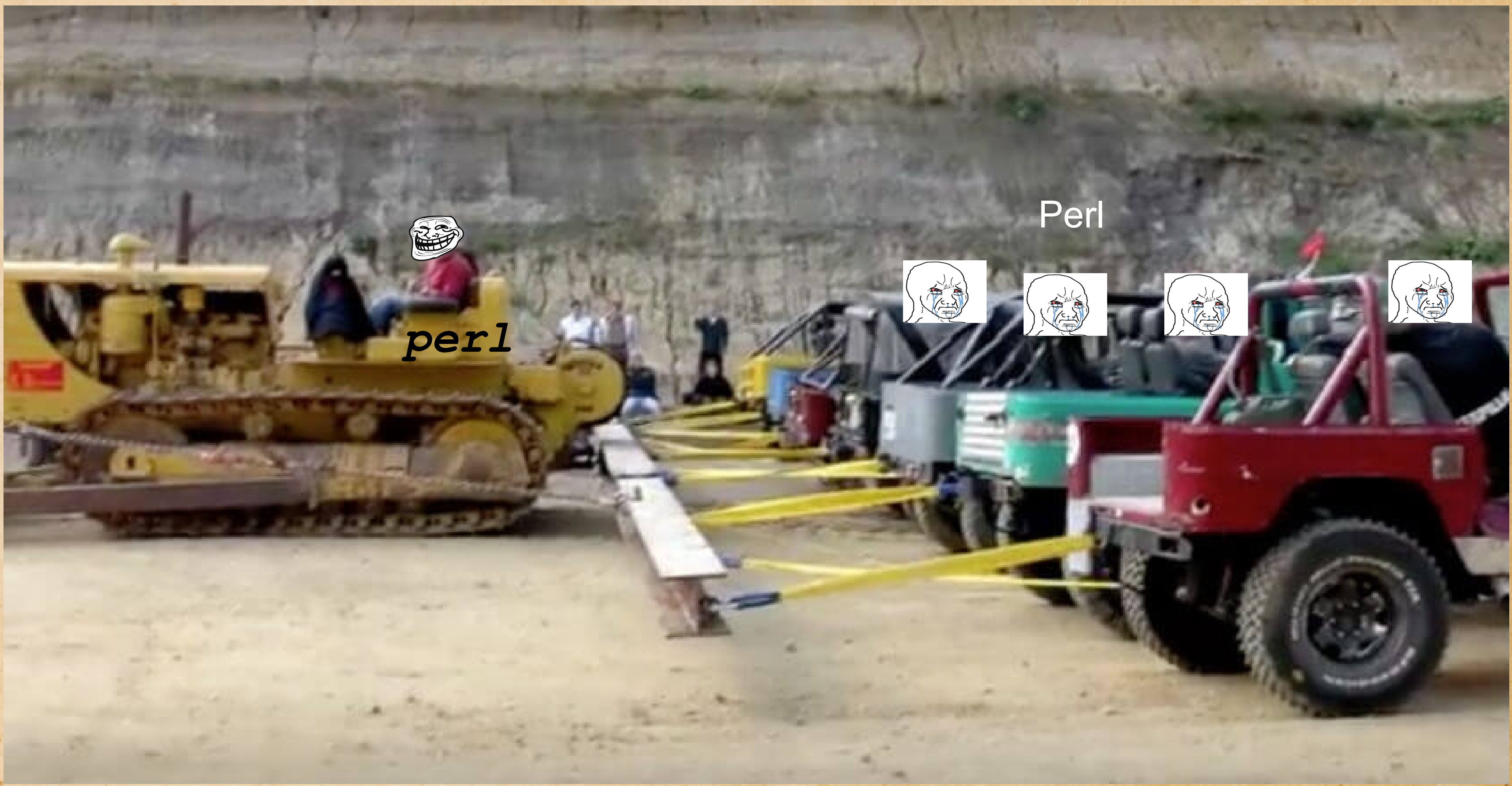
Important Terms to Discuss

- Uniprocess
- Memory model
- Sequential Consistency
- Finite Automata
- Regular Expression
- Shuffle Operator

`perl` is a *uniprocess*

- `perl` 5.000 (*1994*) had no reason to consider executing over more than a single processor
- Linux kernel 2.0 (*1996*) was the first step to OS support for more than 1 CPU (aka, symmetric multi-processing, SMP)
- Many attempts to *work around* this “limitation” in `perl` have been introduced over the years
- Most “solutions” are based on `fork`, which necessarily does't admit SMP
- Other solutions do *weird* things with context switching to *emulate* threads
- **A LOT** of negativity in the community as come from the lack of embracing the *uniprocess* nature of perl
- **BUT** there is a silver lining to this and I believe a path forward
- ...

RECAP: Perl vs `perl`



Sequential Consistency

With respect to "parallel" or "threaded" programs

*"... the result of any execution is the same as if the operations of all the processors were executed in **some sequential order**, and the operations of each individual processor appear in this sequence in the order specified by its program." - Leslie Lamport* [2]

AGAIN, "*all the fun and profits of traditional shared memory programming, but on fewer CPUs than intended – usually, just <u>one</u> (as in the case of this being applied to* `perl`*)*"

Benefits of a Uniprocess

- Inherent memory consistency (no competing *threads*, no corruption)
- No race conditions
- Easy, powerful runtime memory introspection
- `perl` in particular has a *richly* developed set of variable scoping at our disposal (e.g., package, file, block, block-local, stateful subroutines)
- *Some* (me) even claim uniprocess perl has more in common with a uniprocess OS* than other PLs

*albeit one lacks a *decent* programming language; this is **not** true it has at least 6: regexes, XS, FORMAT, s/printf, un/pack, strftime

`perl`'s *Memory* Model

- In computing, a memory model describes: the interactions of threads through memory and their shared use of the data. (wikipedia)
- `perl` is a uniprocess; there are no threads; it is sequential – so what's the "memory model"?
- The closest we have are Perl's *variable scoping*, data isolation, refcounting, etc
- `perl` also can give subroutines memory between calls via the `state` declaration

What about Performance

- This talk doesn't consider this directly
- And when most people discuss “performance” in the context of “concurrency”, what they really care about is “*speed up*”
- Speed up; i.e., the decreasing “*time to solution*” one achieves as more processes are thrown at a particular task – see `fork`
- We shall see that the approach presented offers some interesting opportunities for managing “speed up”

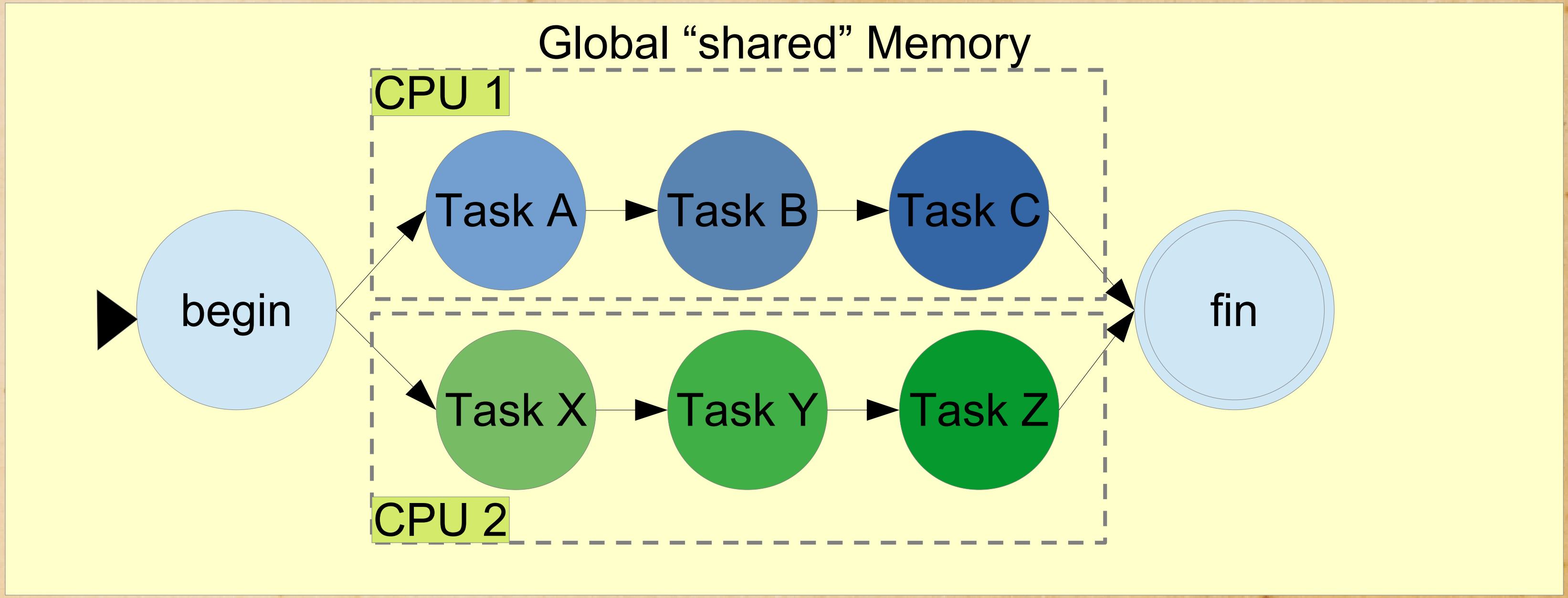
Where this is heading: Concurrent SMP

- Shared memory accomplished through global variables
- Race conditions reduce down to "*safe*" side effects
- No competition for system resources (e.g., disk i/o, network, etc)
- No 'corrupted' memory due to partial writes by 2 or more "threads"
- Though, "overwrites" are absolutely possible; the writes will still be complete
- All the joins of SMP programming and *none* of the speed up! - So what's the benefit?

Sequential Consistency

Example by dependency graph, "A→B→C & X→Y→Z"

- Imagine a program that is written (in a supportive language) such that two independent lines of execution exist.



Sequential Consistency

Example by dependency graph, "A→B→C & X→Y→Z"

- *Total* Ordering?

  A must precede B, B must precede C

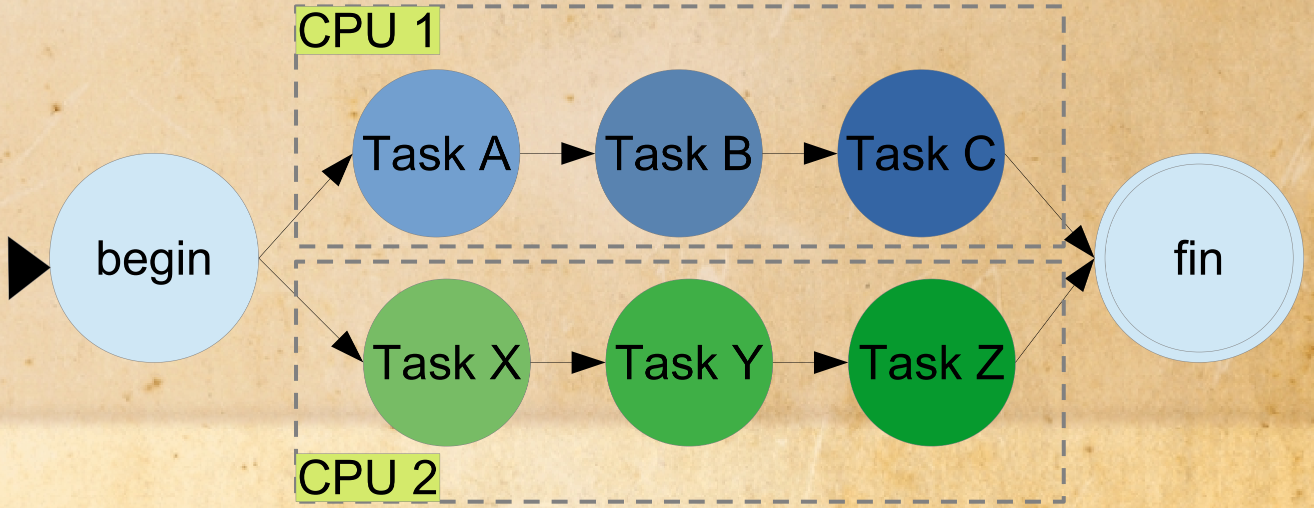
  X must precede Y, Y must precede Z

- Partial Ordering?

  Tasks in {A,B,C} may happen in any order with respect {X,Y,Z}

  "execution" paths A→B→*C* is independent of the path X→Y→*Z*

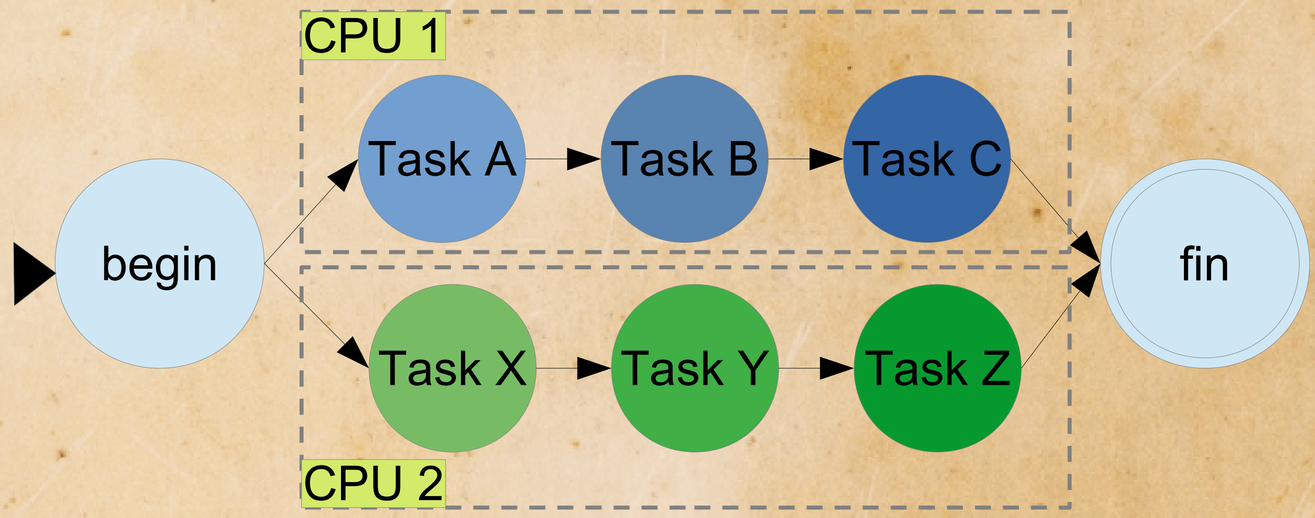
  This independence *implies* concurrency



Sequential Consistency

Example by dependency graph, "A→B→C & X→Y→Z"

- "*ABC*" and "*XYZ*" naturally map to an environment with, e.g., 2 CPUs
- Each independent execution path may happen simultaneously
- Using more than 2 CPU units provides no parallel "*speed up*"

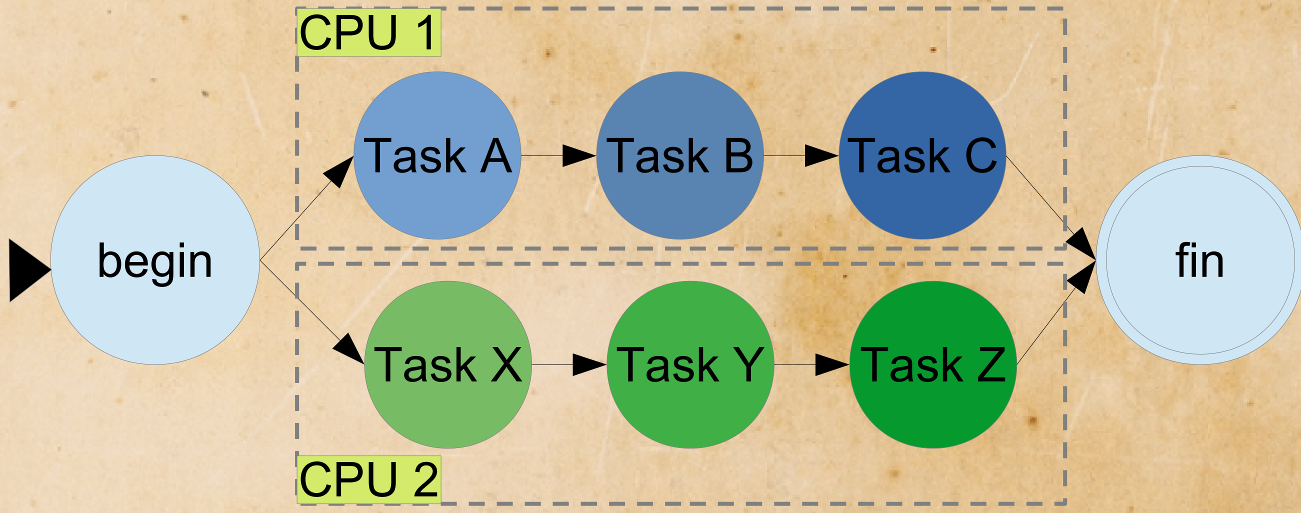


- **What about using 1 processing unit?**

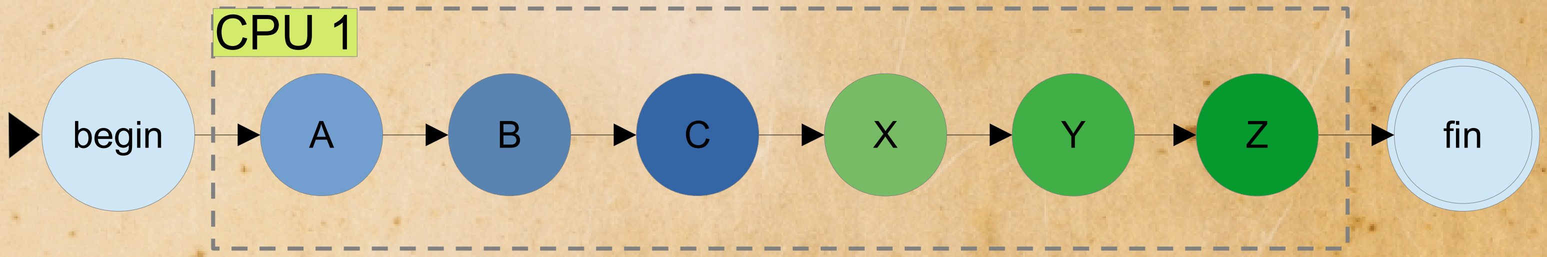
How Sequentialize Consistently?

Example by dependency graph, "A→B→C & X→Y→Z"

- How do we go from 2 CPUs:



- To 1 CPU such that the serialized execution is consistent with a valid ordering using 2 CPUs?

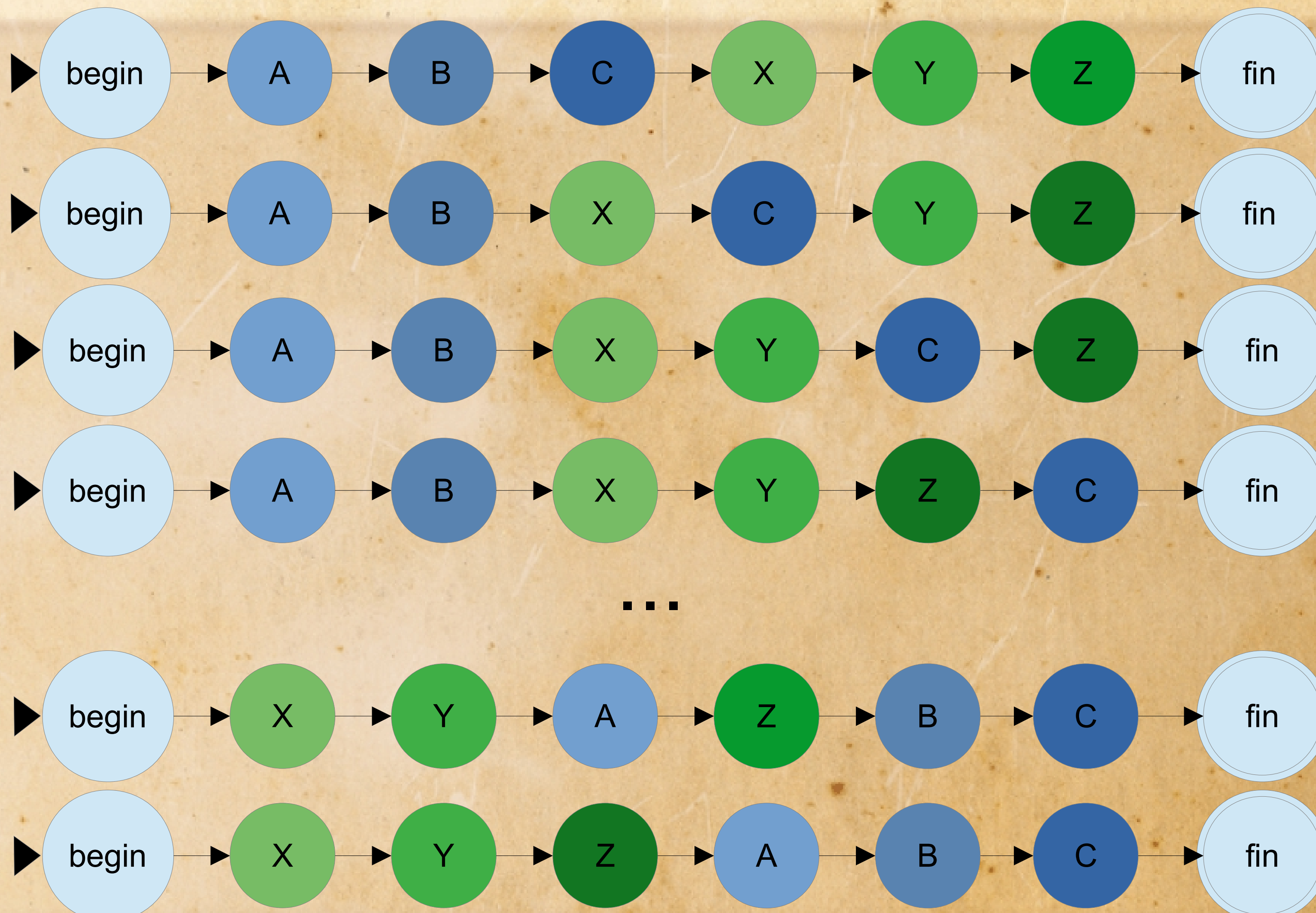


`Execution order: ABCXYZ`

Sequential Consistency

Example by dependency graph, "A→B→C & X→Y→Z"

ABCXYZ
ABXCYZ
ABXYCZ
ABXYZC
AXBCYZ
AXBYCZ
AXBYZC
AXYBCZ
AXYBZC
AXYZBC
XABCYZ
XABYCZ
XABYZC
XAYBCZ
XAYBZC
XAYZBC
XYABCZ
XYABZC
XYAZBC
XYZABC

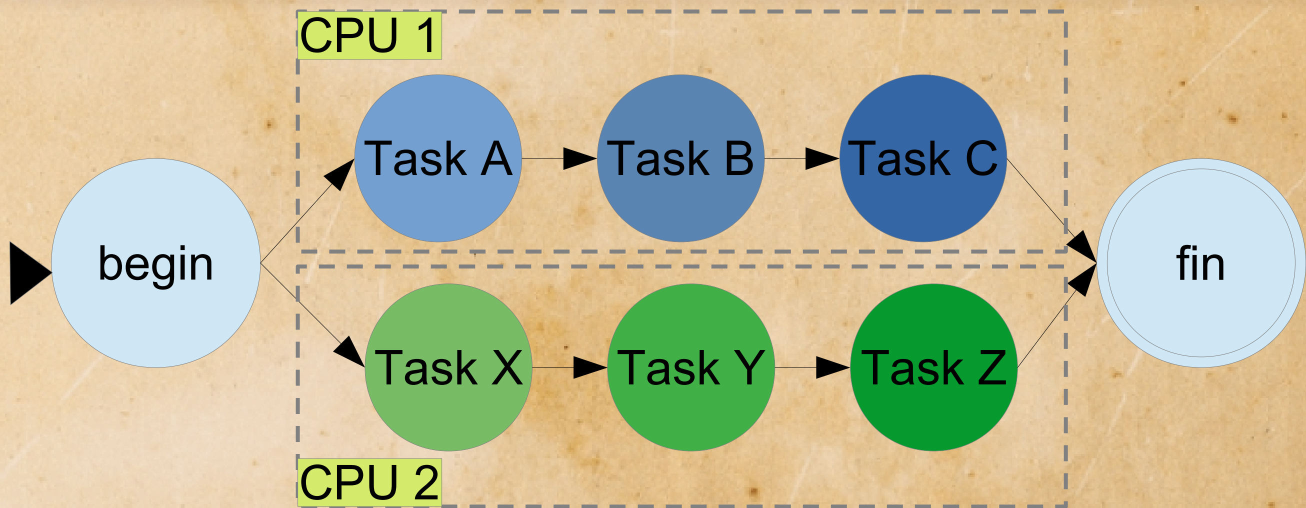


However, there are **20** valid sequential orderings implied by the concurrent "version" executing in parallel!

Sequential Consistency

Example by dependency graph, "A→B→C & X→Y→Z"

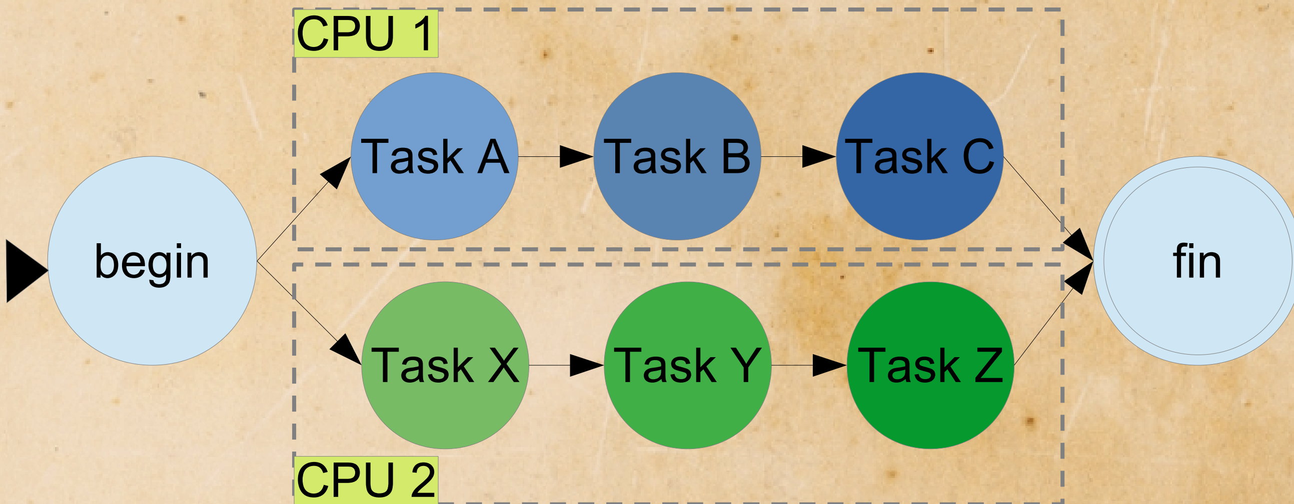
ABCXYZ
ABXCYZ
ABXYCZ
ABXYZC
AXBCYZ
AXBYCZ
AXBYZC
AXYBCZ
AXYBZC
AXYZBC
XABCYZ
XABYCZ
XABYZC
XAYBCZ
XAYBZC
XAYZBC
XYABCZ
XYABZC
XYAZBC
XYZABC



- All *sequentially consistent* (i.e., valid) execution orderings maintain the total orderings: **`"A→B→C" & "X→Y→Z"`**
- Yet, admit the partial orderings permitted by the lack of any dependency between **`{A,B,C} and {X,Y,Z}`**.

How Can We Serialize *Arbitrary Dependencies*?

- **Step 1** - recognize a dependency graph is just a directed *graph*



- **Step 2** – recognize that the serialization of a graph consisting of nodes and labels has been studied a lot over the decades

- **Step 3** – recall that day in your CS class on Computer Theory they covered the *shuffle* operator

Spoiler! We can do so using well known concepts and algorithms from the study of Finite Automata and Regular Language.

Finite Automata

- **Deterministic Finite Automata** (DFA)

  A state machine that can be described as a regular expression (not the Perl kind!)

  Describes a set of valid strings; also known as a *Regular Language*

- **Non-deterministic Finite Automata** (NFA)

  A DFA + an ε-transition ("epsilon") that admits non-determinism

- **Parallel Finite Automata** (PFA)

  A NFA + a λ-transition that adds additional constraints to the language it describes

  Equivalent to binary PetriNets (*Stotts, Pugh 1994 – Parallel Regular Expressions)*

Describing Regular Languages

- **Deterministic Finite Automata** (DFA)

  Concatenation, Union, Kleene Star

- **Non-deterministic Finite Automata** (NFA)

  Concatenation, Union, Kleene Star

- **Parallel Finite Automata (**PFA)

  Concatenation, Union, Kleene Star

  + the *Shuffle* operator

  A PFA is equivalent to a *binary* PetriNet

What is the Shuffle *Operator* [1]?

- Most are familiar with operations over regular languages:

  Concatenation (*usually* implied, ".")

  Union ("|")

  Kleene Star ("*")

- Under these operations, the results are "closed" under RLs; i.e., their application results in another RE
- Shuffle is a less well know, but properly closed under REs.
- Is the "and" to Union's "or"
- Effectively *interleaves* valid strings from two regular languages
- A RE with a shuffle is called a, *Parallel Regular Expression*
- *Operator is often represented as an ampersand, "&"*

What is the Shuffle Operator?

Examples

- Language 1 (L_1) – the string "**ab**"
- Language 2 (L_2) – the string "cd"
- Shuffle, L_1 & L_2:

  "**ab**&cd"

  Requires a valid string from <u>BOTH</u> languages are *interleaved* in any valid string

  Valid strings are:

  abcd

  ac**b**d

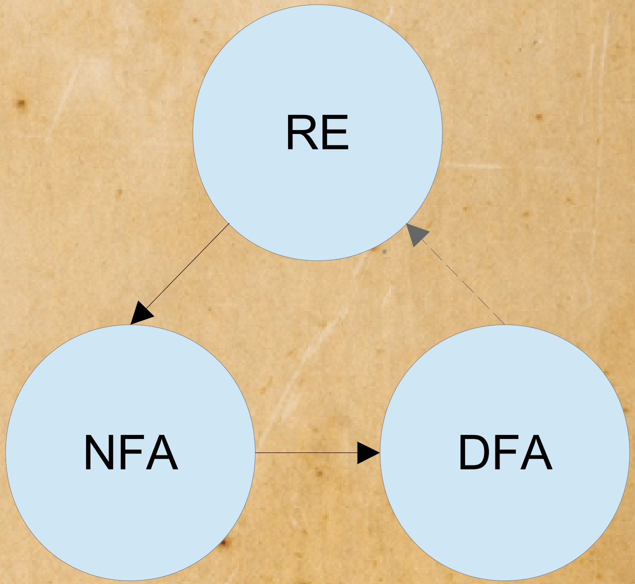
  acd**b**

  c**a**d**b**

  cd**ab**

Converting Non-Shuffled REs to DFA

- RE → NFA
  - Recursive descent parse to create an AST
  - *Thompson Construction* to construct a NFA from a traversal of the AST
  - Introduces ε-transitions
- NFA → DFA
  - Subset Construction Algorithm
- DFA → valid strings
  - Depth-first graph traversal



Converting Shuffled PREs to DFA

- PRE → PFA

  Recursive descent parse to create an AST

  "*modified*" Thompson Construction to construct a PFA from a traversal of the AST

  Introduces both ε-transitions and λ-transitions

- PFA → NFA

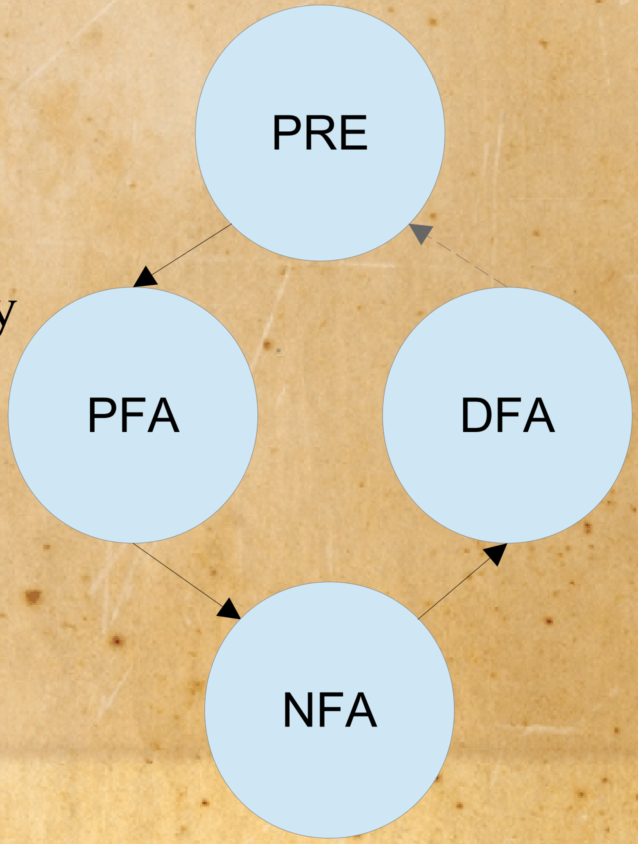
  Algo replaces λ-transitions with ε-transitions by enumerating valid partial orderings

- NFA → DFA

  Subset Construction Algorithm

- DFA → valid strings

  Depth-first graph traversal



Strings from a *finite* Language

- A *finite* language is any language described by a Regular Expression that does **NOT** include a **Kleene Star**; otherwise there are an *infinite* set of valid strings
- *Given* a DFA, generate a set of valid strings in the language it describes
- Solution:

    traverse graph from start to an accept node, collect labels (e.g., letters or symbols) along the way

    Print set of collect symbols collected along the path

Strings from an in*finite* Language

- *(slide added for completeness, no examples in this talk use infinite FAs to to produce execution schedules)*
- An in*finite* language is any language described by a Regular Expression that **does** include a **Kleene Star**; otherwise there are an *finite* set of valid strings
- *Given* a DFA, generate a set of valid strings in the language it describes
- Solution:

  traverse graph from start to an accept node, collect labels (e.g., letters or symbols) along the way

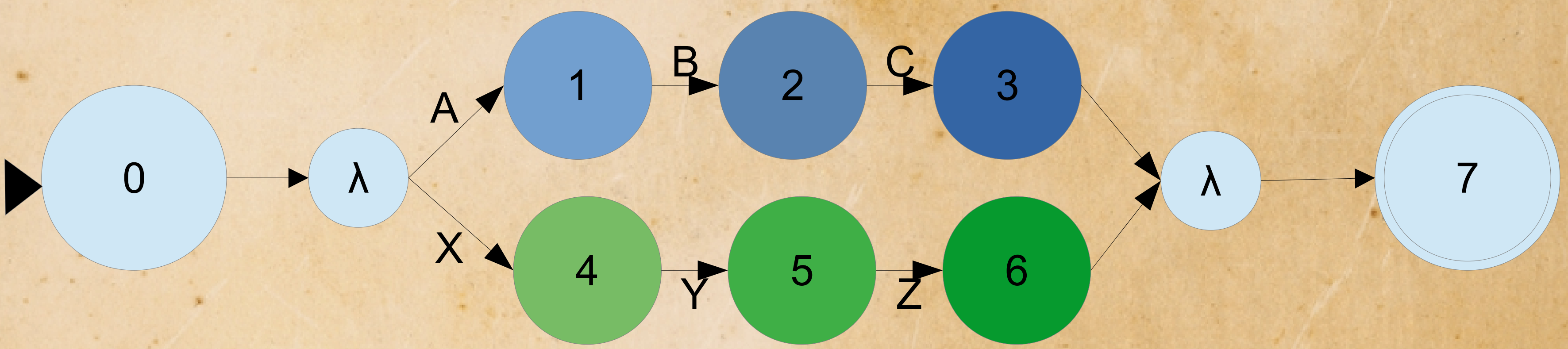
  Manage how "deep" into an infinite cycle the traversal is, back out once a maximum is met

  Print set of collect symbols collected along the path

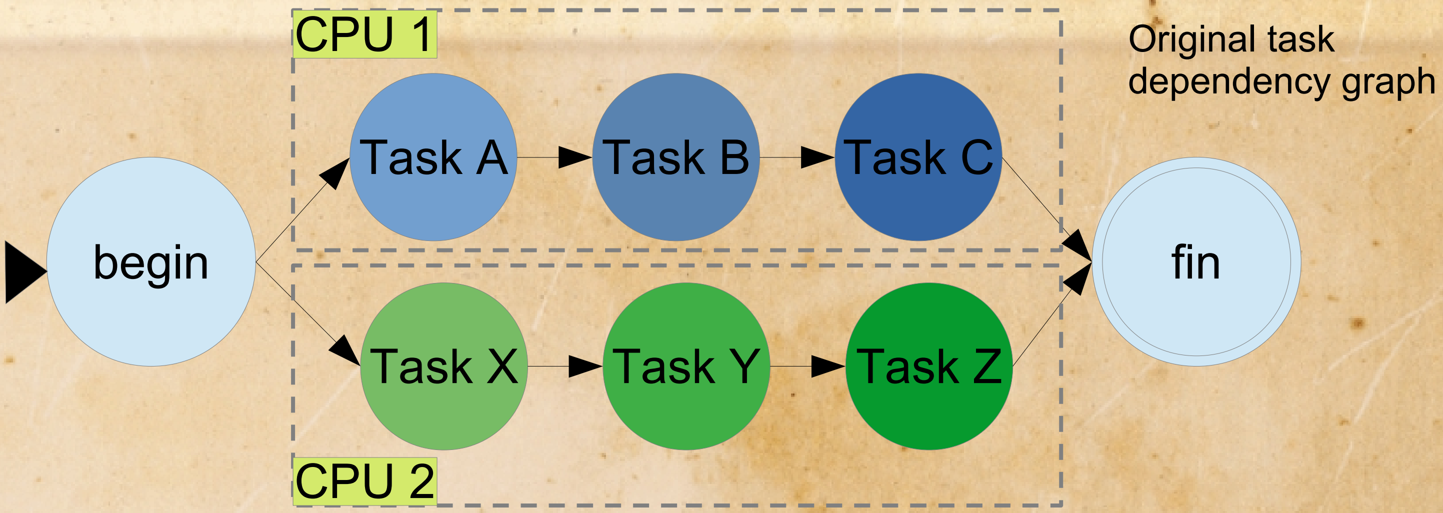
Shuffle Example

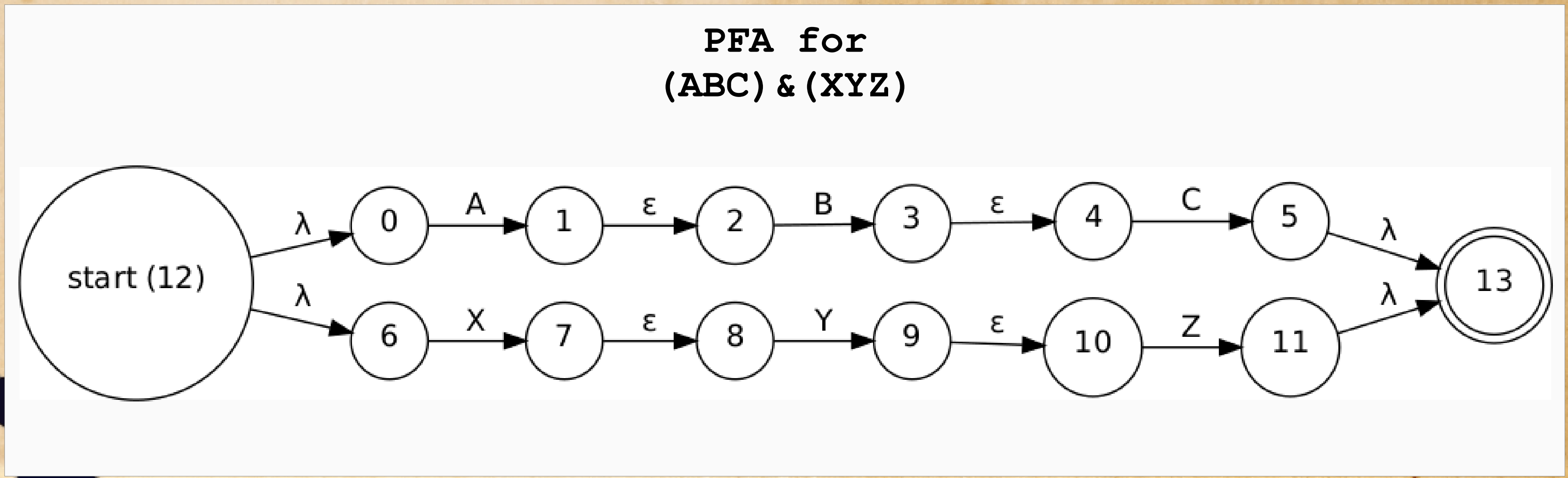
Example of "parallel" Regular Expression, "<u>ABC&XYZ</u>"

- "ABC" & "XYZ"
- More formal (labeled transitions, numbered states) diagram of the **Parallel Finite Automate**:



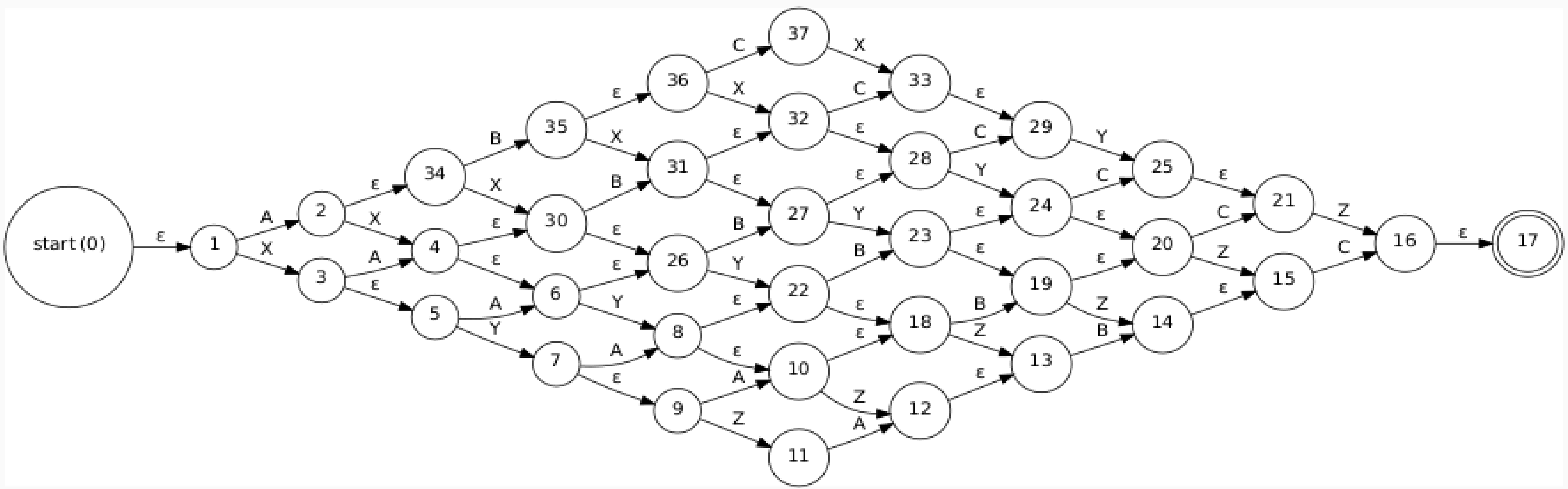
Resulting PFA



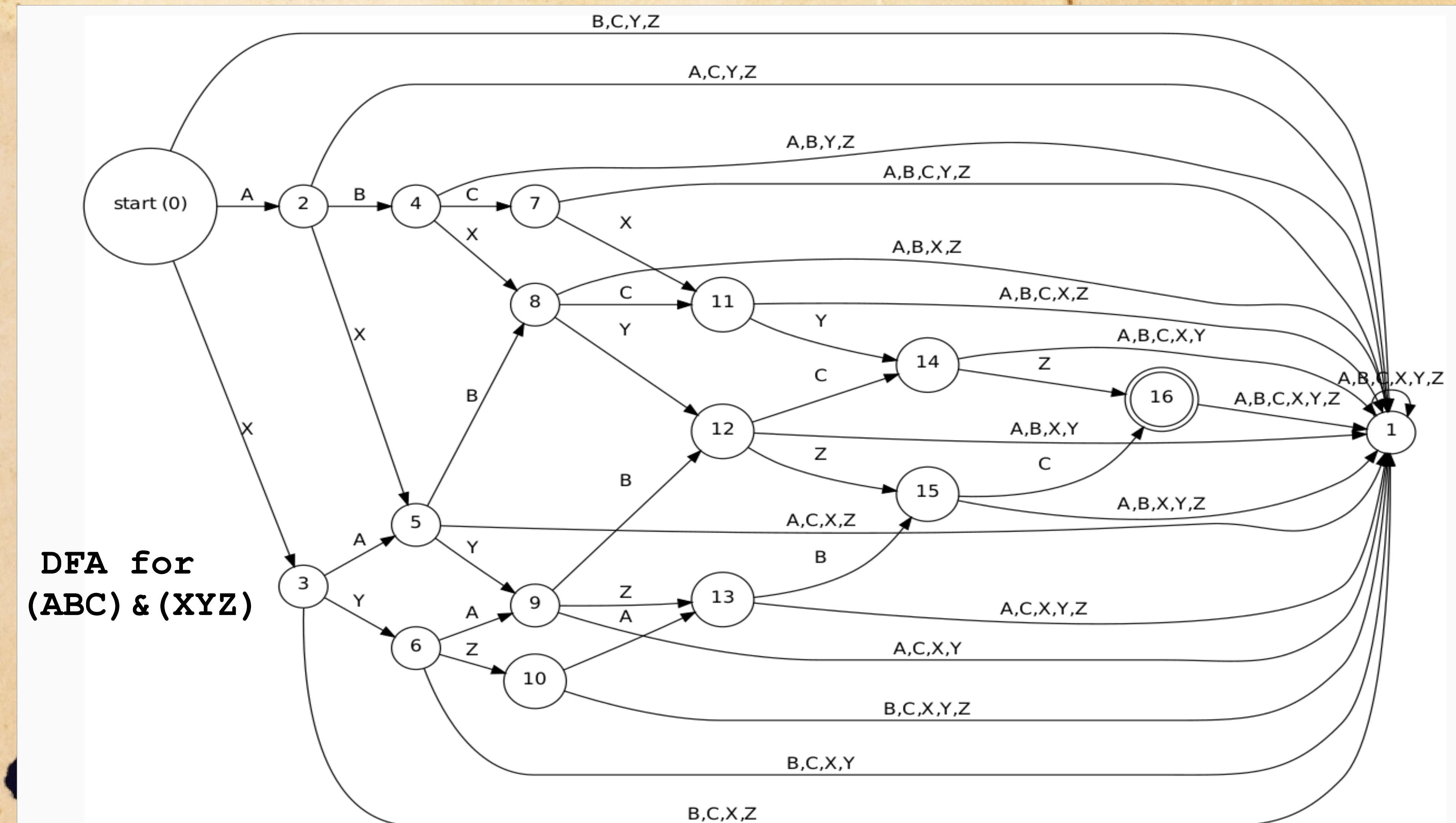


Resulting NFA

NFA for
(ABC)&(XYZ)



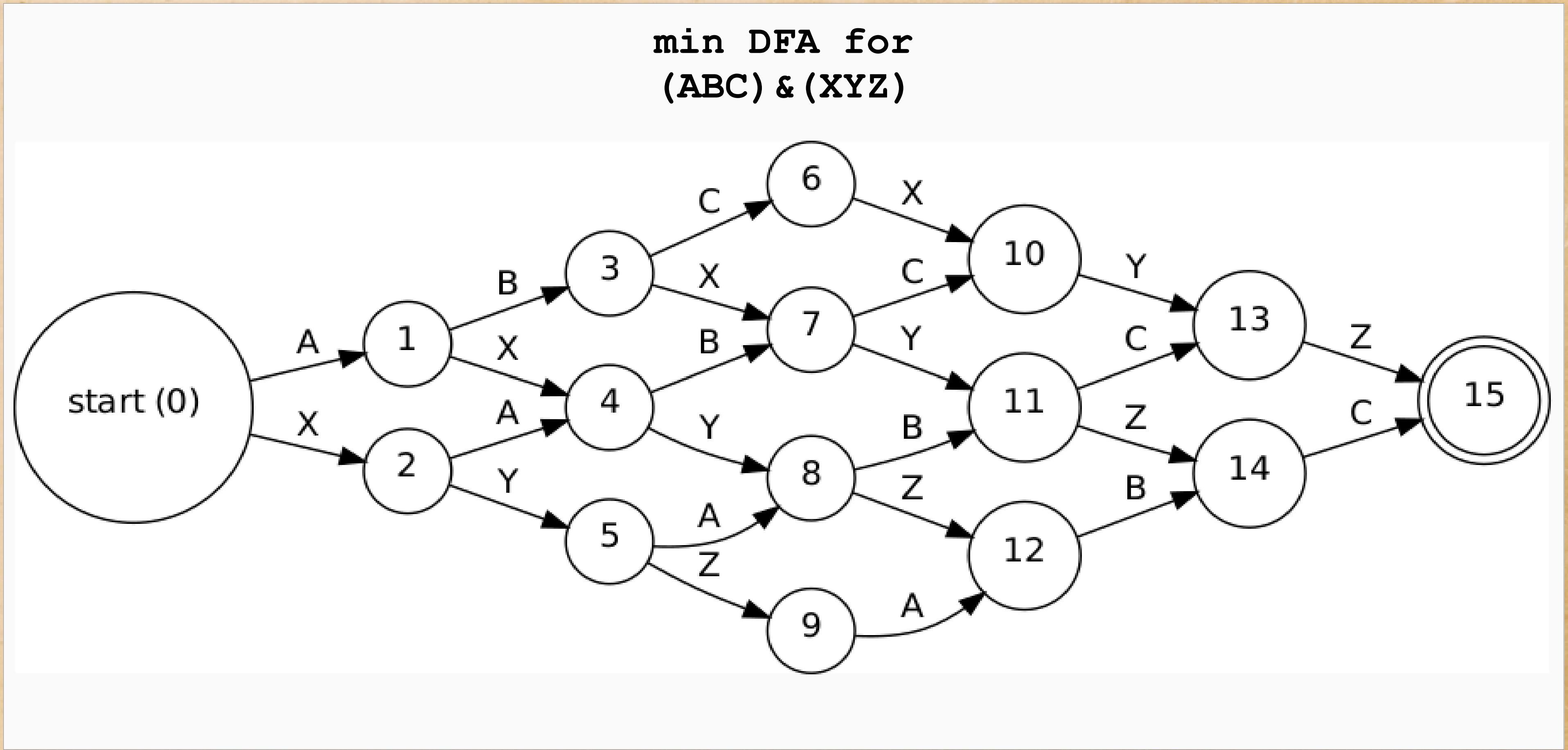
Resulting* DFA



DFA for (ABC)&(XYZ)

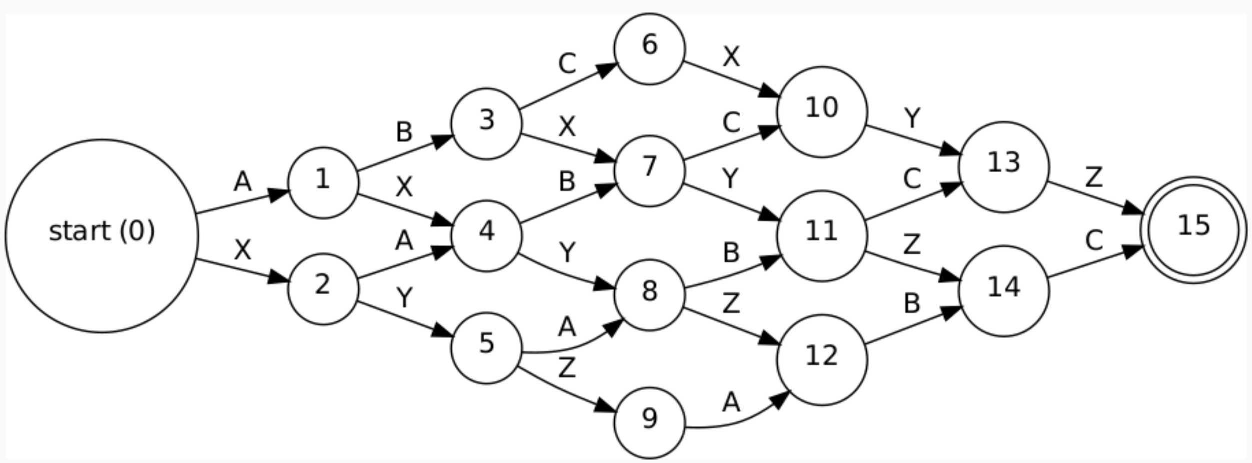
*after subset construction NFA→DFA

Resulting Minimized DFA



Enumeration of all Start → Accept Paths

min DFA for
(ABC)&(XYZ)

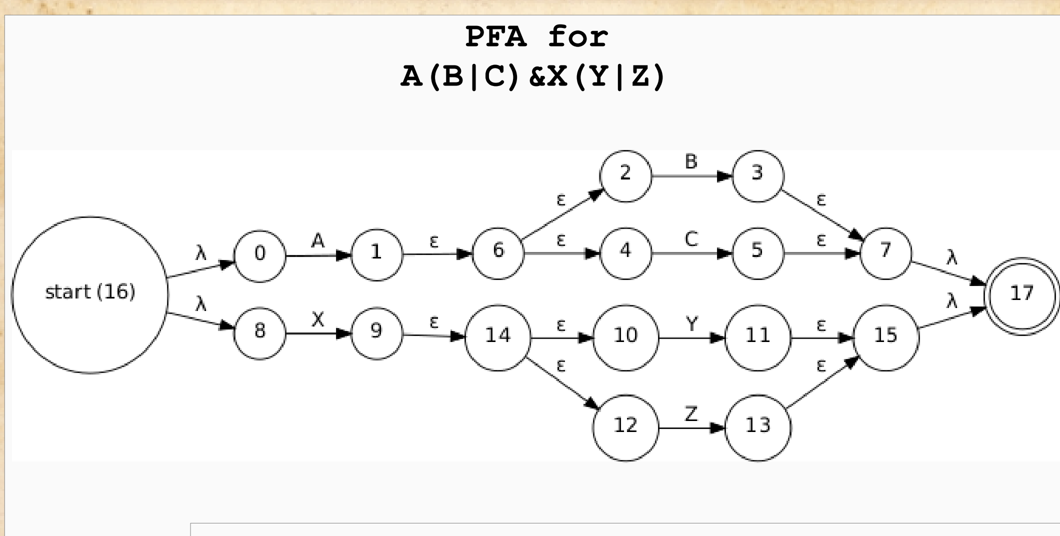


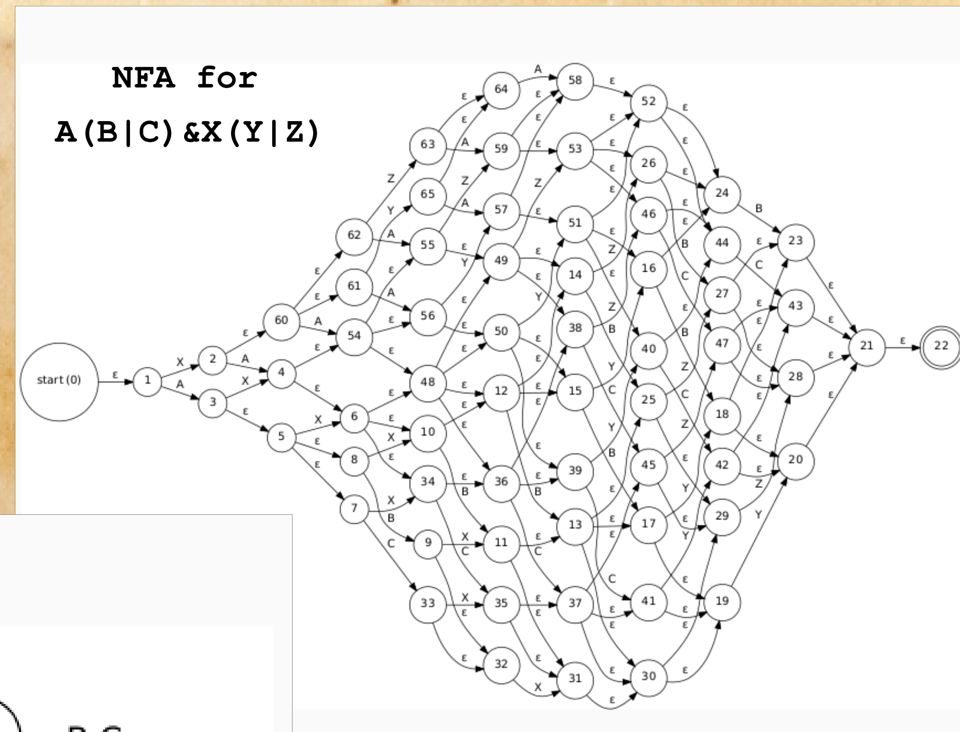
All Start → Accept paths represent valid strings resulting from the interleaving of valid strings from the *shuffled* “languages”.

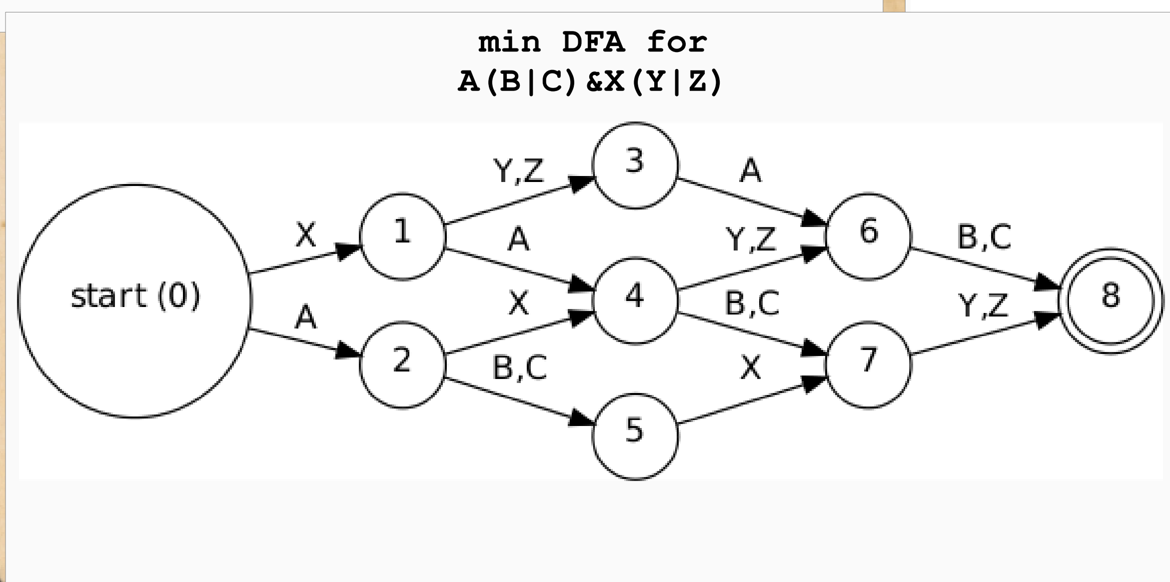
Because this process maintains the partial ordering required by each language in the shuffle, these strings are sequentially consistent, and therefore <u>represent valid execution orders implied by the original concurrent description.</u>

ABCXYZ
ABXCYZ
ABXYCZ
ABXYZC
AXBCYZ
AXBYCZ
AXBYZC
AXYBCZ
AXYBZC
AXYZBC
XABCYZ
XABYCZ
XABYZC
XAYBCZ
XAYBZC
XAYZBC
XYABCZ
XYABZC
XYAZBC
XYZABC

Example with all *Finite* RE Operators







Recall: *Motivation*

- **Express concurrency naturally in Perl**
  (for Humans to reason easily)
- **...that can leverage the strengths of `perl`**
  (uniprocess, memory models)
- **...and do it in a fun and interesting way**
  (primary goal)

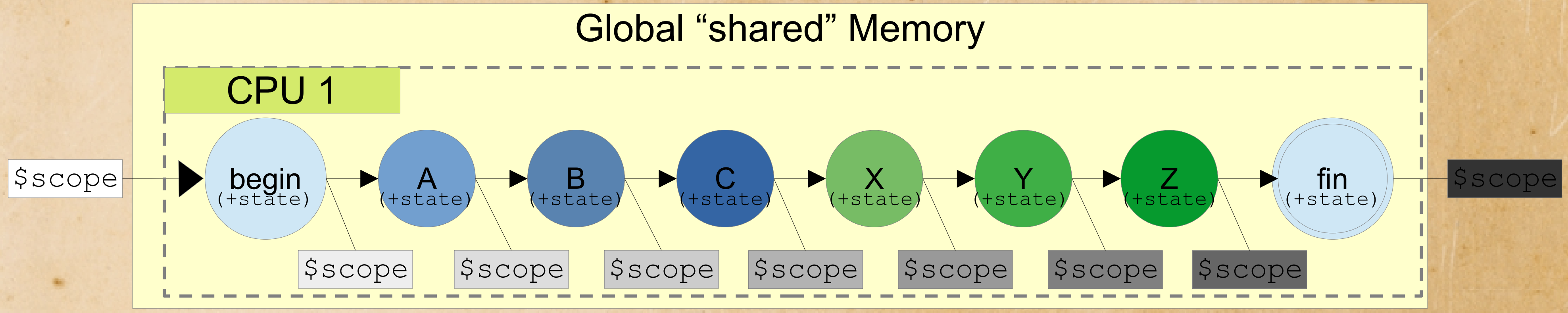
`Sub::Genius`

A Perl module for managing concurrent semantics for serial execution

- Takes a concurrent "plan", i.e., a Parallel Regular Expression
- "dumbs down" SMP for perl, hence the module name (credit: *TEODESIAN*)
- A "plan" describes the execution dependency of subroutines with concurrent semantics (i.e., using a "*shuffle*" operator)
- Serializes it to one of likely *many* serialized forms
- All subroutines are called in the same `perl` process, so we can leverage:
  - Global (*shared*) memory
  - Execution "scope" passed in/out of each subroutine call
  - Subroutines with **`state`** that may change from call to call!

Sub::Genius

A Perl module for managing concurrent semantics for serial execution

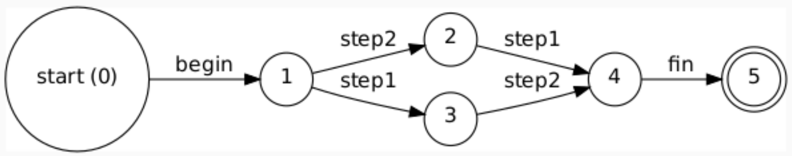


- Share memory space, accessible to all subroutines
- An execution “$scope” that is passed in a *pipeline* fashion
- Subroutines that “remember” between calls – e.g., coroutines

Anatomy of a Sub::Genius program

minimal DFA for

`begin ( step1 & step2 ) fin`



Valid sequential forms:

- `begin → step1 → step2 → fin`
- `begin → step2 → step1 → fin`

```perl
use strict;
use warnings;
use feature 'state';

use Sub::Genius ();

# the plan is a PRE, 'parallel regular expression' in the Formal sense
my $preplan = q{
begin
  ( step1 & step2 )
fin
};

# Load PRE describing concurrent semantics
my $sq = Sub::Genius->new( preplan => $preplan );

# treated as "shared memory"
my $GLOBAL = {};

# run plan
my $final_scope = $sq->run_any( scope => {}, );

# implement sub routines
sub begin {
  my $scope = shift;
  state $sub_state = {};
  # ..
  return $scope;
}

sub step1 {
  my $scope = shift;
  state $sub_state = {};
  # ...
  return $scope;
}

sub step2 {
  my $scope = shift;
  state $sub_state = {};
  # ...
  return $scope;
}

sub fin {
  my $scope = shift;
  state $sub_state = {};
  # ...
  return $scope;
}
```

declare“parallel plan”

shared memory

execution“*scope*”

subroutines with “**`state`**” adds co-routines

subroutine implementations

`Sub::Genius` Tooling & Info

- `stubby`

  General utility for making it easier to create and work with programs utilizing the sequential planning of `Sub::Genius`

- `fash`

  Wrapper around FLAT one-liners; old and crusty, but still useful for learning things about PREs and the FA they define (which can be surprising)

- `S::G` has a lot of information in the POD (`perldoc Sub::Genius`), but more is being written

`Sub::Genius`'s Big Wart

- PREs take exponentially long to convert into a DFA (algo complexity is largely unavoidable)
- Solution:

```
stubby precache -p "a&b&c&d&f"
```

- Basically a "compile" step; saves using `Storable`.
- This programming model for Perl/perl is still embryotic, but it's interesting and powerful
- More work must be done to explore this space

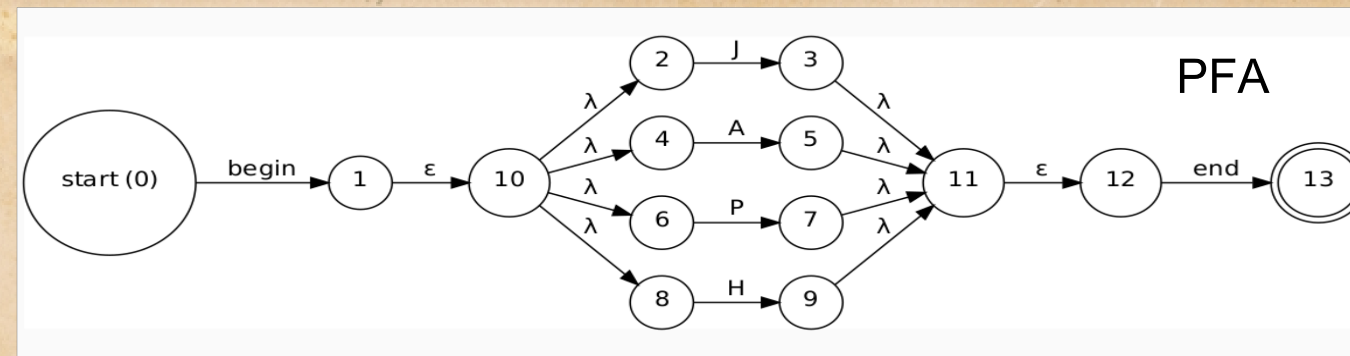

"Now is the time on
Sprockets when we
demo."

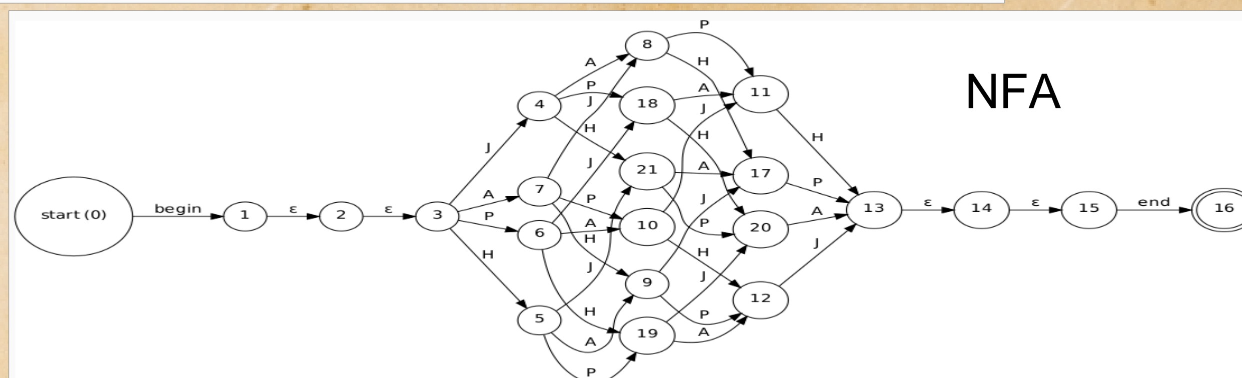
just another Demo

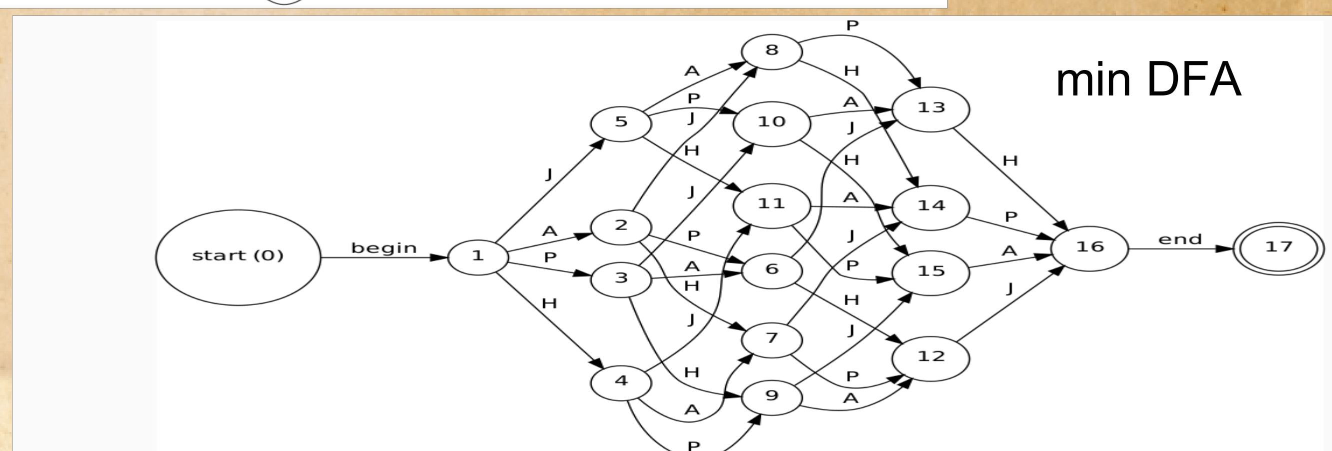
```
#
# Implements classic JAPH :-)
#

my $pre = q{
begin
(
  J &
    A &
      P &
        H
)
end
};z
```

Underlying Automata

Example – "JAPH", stages of serialization using FA conversion algorithms







A Way Forward for...

- Allows for use of concurrent semantics *now*
- Exploitation of `perl`'s uniprocess memory model
- Exploring concurrent semantics in Perl – e.g., “fork/join”, “async/away”, etc
- Other things, I am sure

SEE ALSO

- Pipeworks
- Sub::Pipeline
- Process::Pipeline
- FLAT
- Graph::PetriNet

Good Reads

- 1. https://www.planetmath.org/shuffleoflanguages
- 2. Leslie Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs", IEEE Trans. Comput. C-28,9 (Sept. 1979), 690-691.
- 3. https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-7.pdf
- 4. https://troglodyne.net/video/1615853053
- 5. Introduction to Automata Theory, Languages, and Computation; Hopcroft, Motwani, Ullman. Any year.

Conclusion

- PREs can be used to describe on a useful level, the execution plan of subroutines for the purpose of auto-serialization
- Useful for expressing SMP concurrency for valid executions on single processes
- Lots of work still needs to be done to explore the benefits

Open Questions

- What does it look like when fork is introduced to the supporting subroutines?
- What what is possible when subroutines are treated statefully? (e.g., subs implicitly retain memory)
- How does “async/await”, “futures”, etc translate using this approach
- How can “infinite” languages be used effectively
- What runloop/driver controls would be useful?
- What utilities (e.g., `stubby`, `fash`) would be useful for development and verification of programs written using this approach