



# **EECS 483: Compiler Construction**

## **Lecture 26: Exam Review and Course Wrap Up**

**April 20  
Winter Semester 2026**

# Plan for Today

- Most of time: exam review
- Last 10 minutes: course wrap-up.

# Exam Review



# Final Exam Info

- Final Exam on Wednesday 04/29 10:30-12:30pm
- Location: LCSIB 1365 (Leinweber)
- Possible Topics: Assignments 4 and 5, lecture after spring break.
- Exam Review on Monday, 04/20
- 1 page of notes, double sided ok, printed or written ok.
- Practice material:

Last year's final exam on canvas

Appel book chapters 10 (liveness), 11 (register allocation), 17 (dataflow analysis), 3 (lexing) and 4 (parsing)

Dragon book chapters 8 (register allocation), 9 (data-flow analysis/ optimization), 2 (lexing) and 3 (parsing)

# Practice Material

Last year's final exam on canvas

***Modern Compiler Implementation in ML*** by Andrew Appel (free online access through UM library)

chapters 10 (liveness), 11 (register allocation), 17 (dataflow analysis), 3 (lexing) and 4 (parsing)

***Compilers: Principles, Techniques, & Tools 2e*** by Aho, Lam, Sethi and Ullman, aka the Dragon book (not free)

chapters 8 (register allocation), 9 (data-flow analysis/optimization), 2 (lexing) and 3 (parsing)

# Second Half Course Major Topics

## Garbage Collection

what is garbage/automated MM?

mark/sweep vs copying

## Program Analysis and Optimization

- Rice's Theorem, limitations of optimization
- Register Allocation
- Liveness analysis
- Conflict analysis
- Chaitin's algorithm
- Chordal graphs, perfect elimination order, graph coloring SSA programs
  - Possible values analysis
  - assertion removal
  - General lattice-based dataflow analysis
    - forward vs reverse mode

## Compiler Frontend

- Tokens, lexical analysis
- Regular expressions
- Non-deterministic and deterministic finite automata
- limitations of Regex
- what is syntactic analysis/parsing?
- Context-free grammars
- Parse trees vs abstract syntax trees
- Derivations
- Associativity, precedence
- Top-down parsing
  - LL(1)
  - conversion to LL(1), left-factoring, etc
- Bottom-up parsing
  - LR(0) items, shift-reduce parsing
  - First set, follow set, nullability
  - shift/reduce, reduce/reduce conflicts



# Garbage Collection

# Memory Deallocation and Reuse

- Every modern programming language allows programmers to allocate new storage dynamically
  - New records, arrays, tuples, objects, closures, etc.
- Every modern language needs facilities for **reclaiming and recycling** the storage used by programs
- It's usually the most complex aspect of the run-time system for any modern language.



# Memory Deallocation and Reuse

- Memory used for an object can be reused if it is **garbage**
- What is **garbage**?
  - A value is garbage if it **will not be used** in any subsequent computation by the program
- How do we determine which objects are garbage?

# Identifying Garbage

- How do we determine which objects are garbage?
- Stack-allocation:
  - when we return or tail call, all objects in the stack frame are garbage so that memory can be reused.
  - See: our implementation of return, branch with arguments
  - this is an **under-approximation**. Objects allocated on the stack may remain long after they are no longer used.
- Is it easy to determine which objects are garbage?
  - No. It's undecidable. Eg:  
if long-and-tricky-computation then use v  
else don't use v

# Identifying Garbage

- Since determining which objects are garbage is tricky, people have come up with many different techniques
  - It's the programmers problem:
    - Explicit allocation/deallocation
  - Reference counting
  - Tracing garbage collection
    - Mark-sweep, copying collection
    - Generational GC

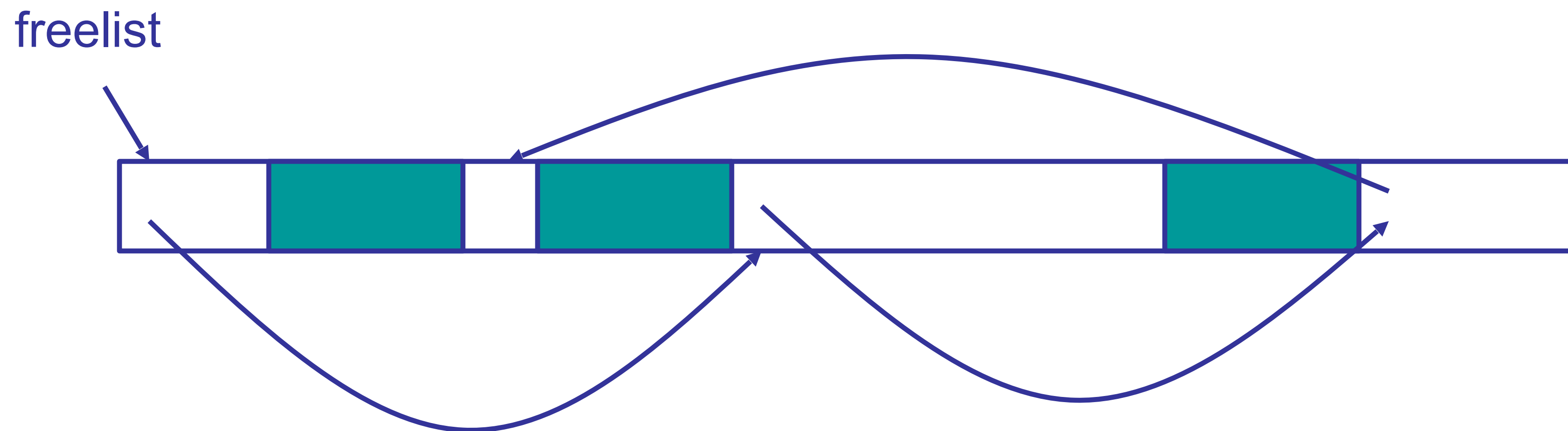


# Explicit MM

- User library manages memory; programmer decides when and where to allocate and deallocate
  - `void* malloc(long n)`
  - `void free(void *addr)`
  - Library calls OS for more pages when necessary
  - Advantage: if you work hard, you can free at the exact right time for their program
  - Disadvantage: people don't want to bother with such details if they can avoid it
  - Disadvantage: difficult to get right, dangerous when wrong

# Explicit MM

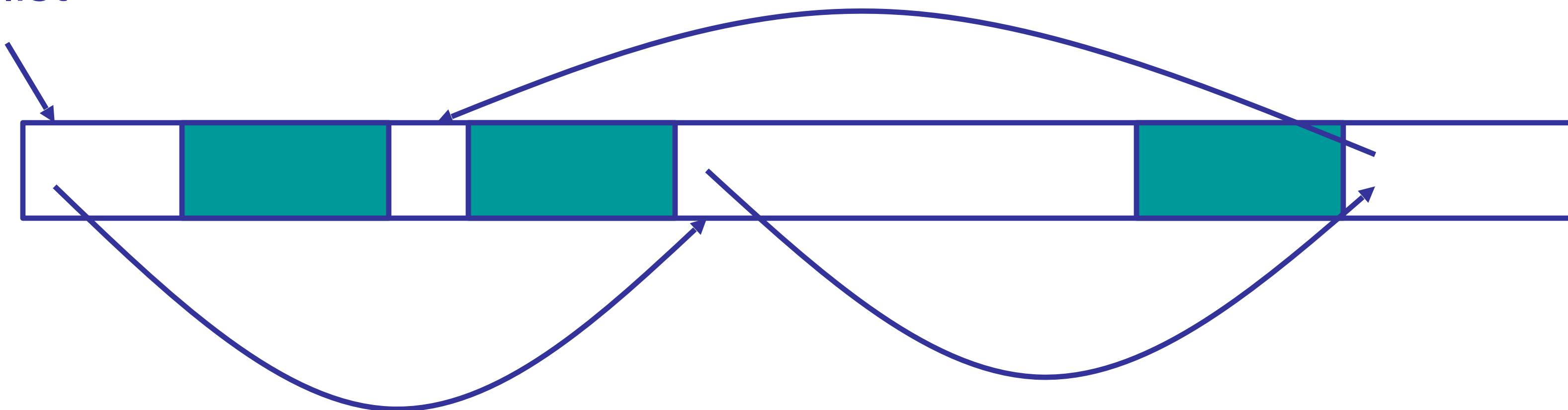
- How does malloc/free work?
  - Blocks of unused memory stored on a **freelist**
  - **malloc**: search free list for usable memory block
  - **free**: put block onto the head of the freelist



# Explicit MM

- Drawbacks
  - **malloc is not free**: we might have to do a significant search to find a big enough block
  - As program runs, the heap **fragments** leaving many small, unusable pieces

freelist





# Explicit MM

- Solutions:
  - Use multiple free lists, one for each block size
    - Malloc and free become  $O(1)$
    - But can run out of size 4 blocks, even though there are many size 6 blocks or size 2 blocks!
  - Blocks are powers of 2
    - Subdivide blocks to get the right size
    - Adjacent free blocks merged into the next biggest size
  - Still doesn't avoid fragmentation
    - 30% wasted space
    - No magic bullet: memory management always has a cost

# Automatic MM

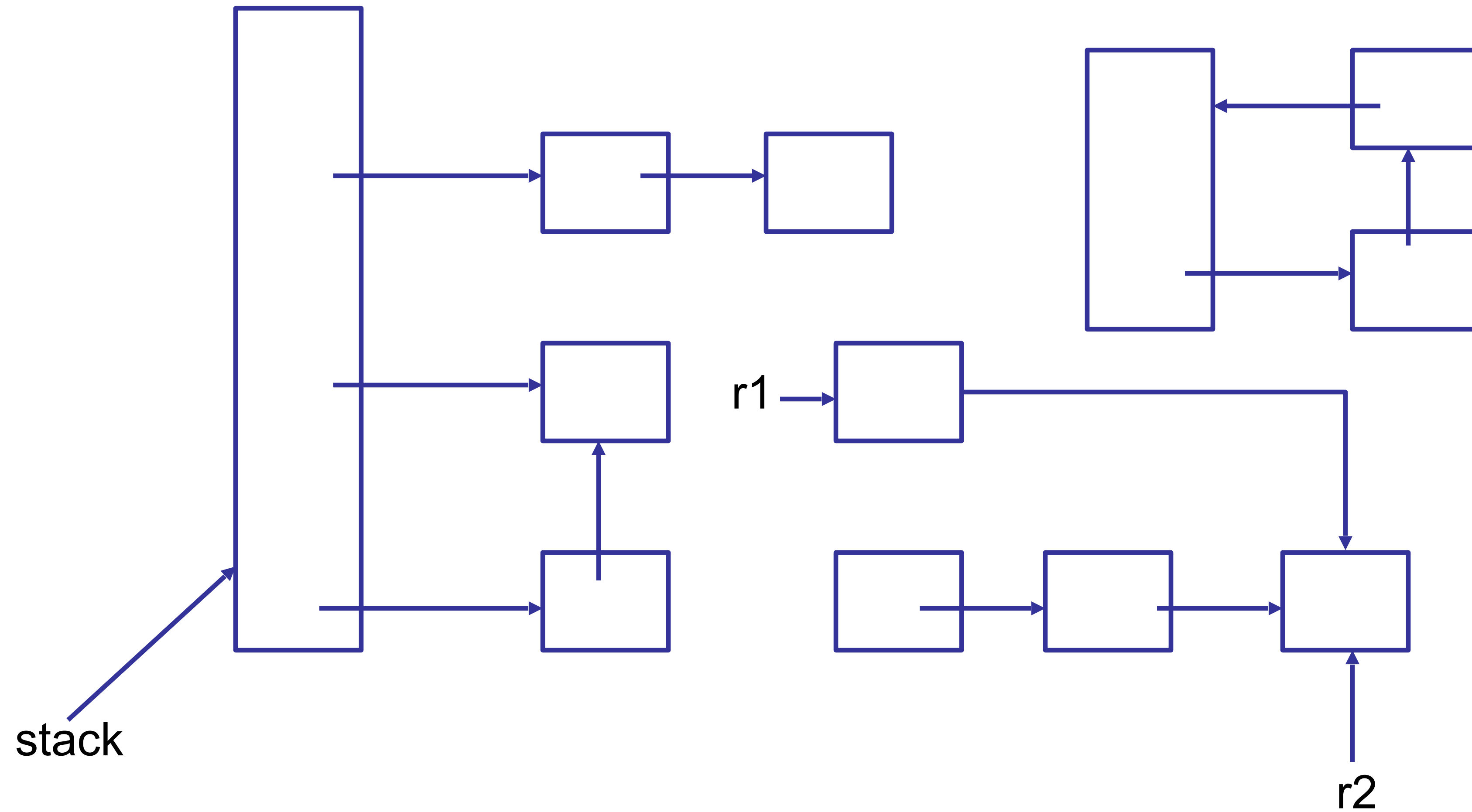
- Languages with explicit MM are much harder to program than languages with automatic MM
  - Always worrying about **dangling pointers**, **memory leaks**: a huge software engineering burden with mistakes often leading to **security vulnerabilities**
  - Languages with unsafe, explicit MM are on the way out. Alternatives like Rust are being incorporated into high-performance settings like Linux kernel and web browsers.
  - Biden administration even instituted an executive order recommending new software be written in memory-safe languages!



# Automatic MM

- Question: how do we decide which objects are garbage?
  - We conservatively approximate
  - Normal solution: an object is garbage when it becomes unreachable from the roots
    - The roots = registers, stack, global static data
    - If there is no path from the roots to an object, it cannot be used later in the computation so we can safely recycle its memory

# Object Graph



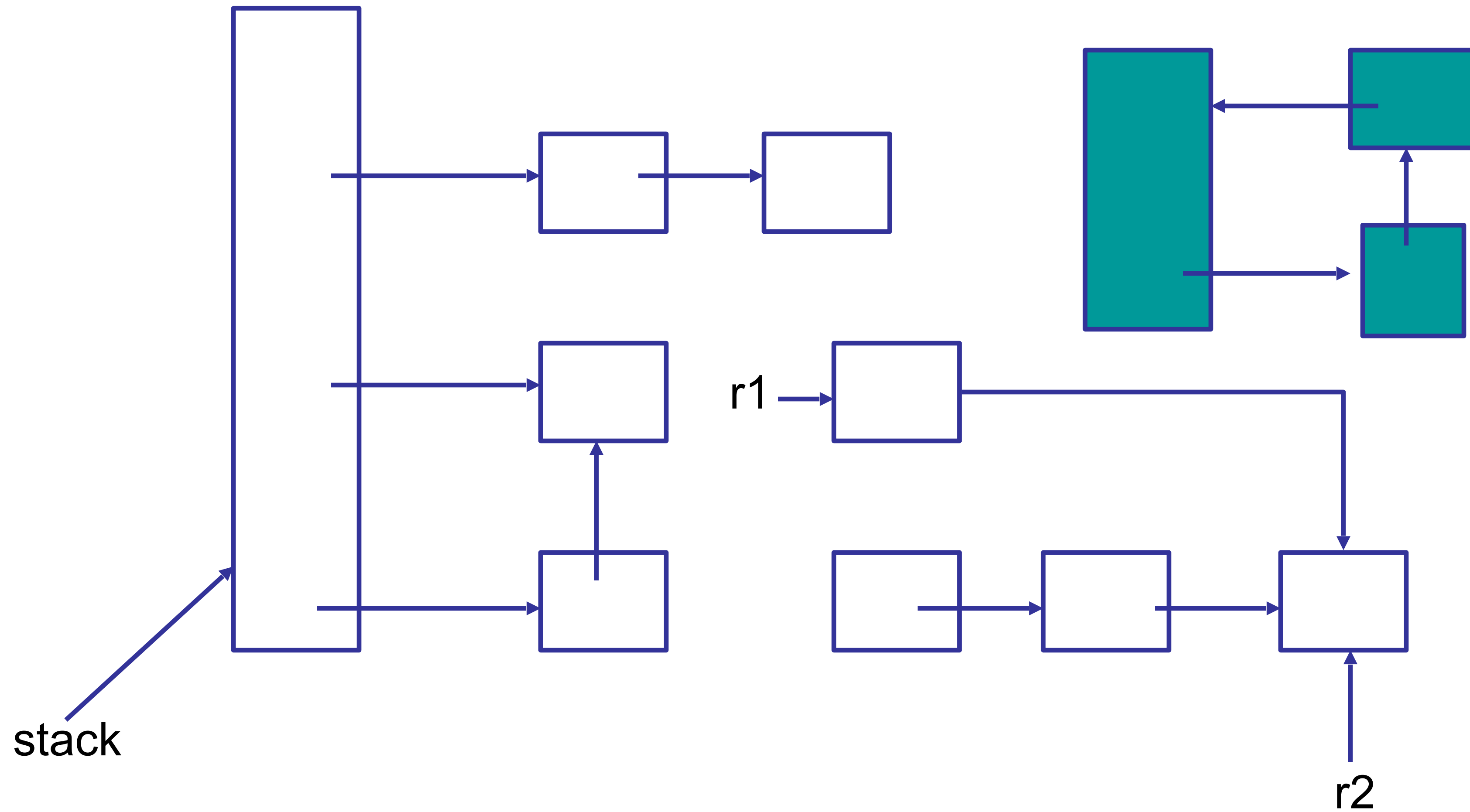
- How should we test reachability?

# Reference Counting

- Keep track of the number of pointers to each object (**the reference count**).
- When the reference count goes to 0, the object is unreachable garbage



# Object Graph



- Reference counting can't detect cycles

# Reference Counting

– In place of a single assignment  $x.f = p$ :

```
z = x.f
c = z.count
c = c - 1
z.count = c
If c = 0 call putOnFreeList(z)
x.f = p
c = p.count
c = c + 1
p.Count = c
```

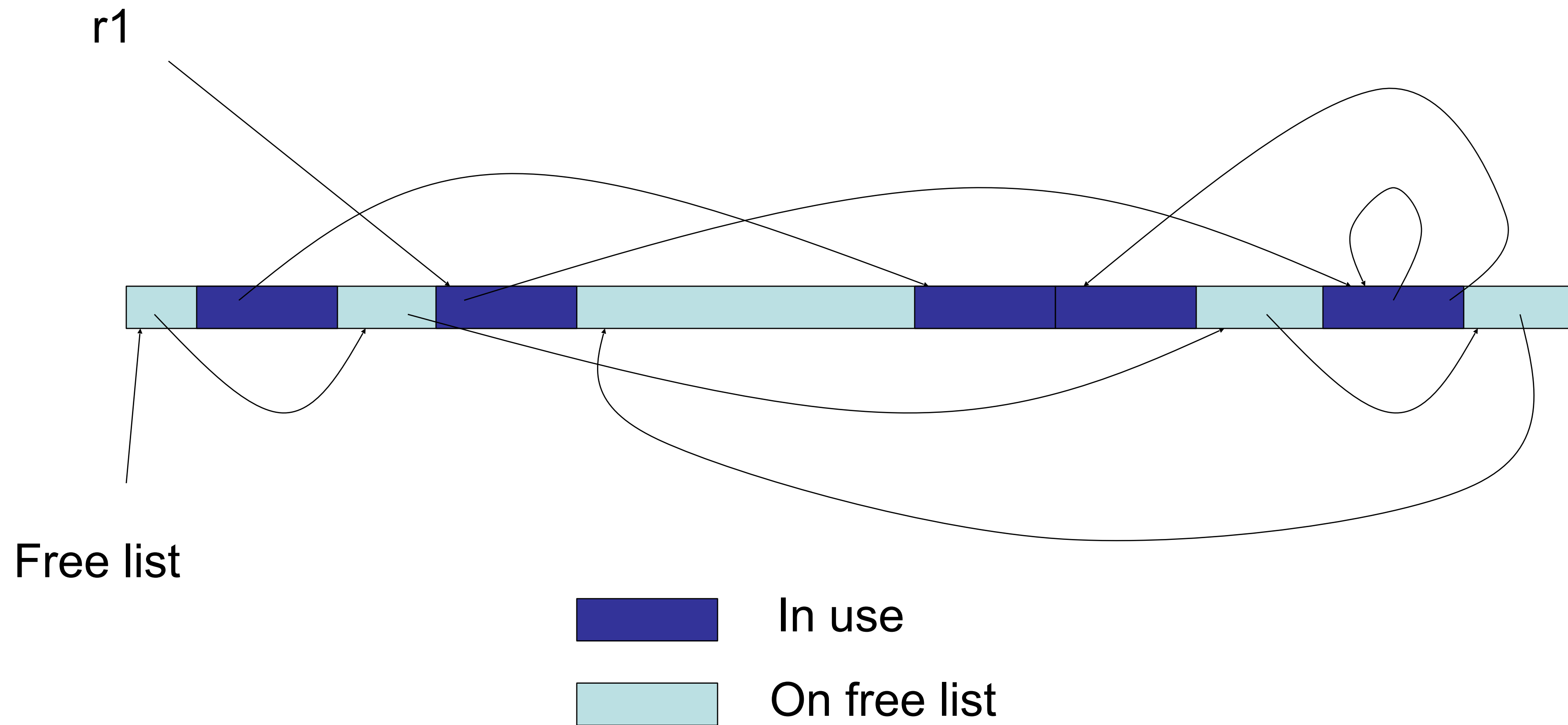
- Ouch, *that hurts* performance-wise!
- *Dataflow analysis* can eliminate some increments and decrements, but many remain
- Reference counting used in some special cases but not usually as the primary GC mechanism in a language implementation

# Mark-sweep

- A two-phase algorithm
  - **Mark phase**: Depth first traversal of object graph from the roots to mark live data
  - **Sweep phase**: iterate over entire heap, adding the unmarked data back onto the free list

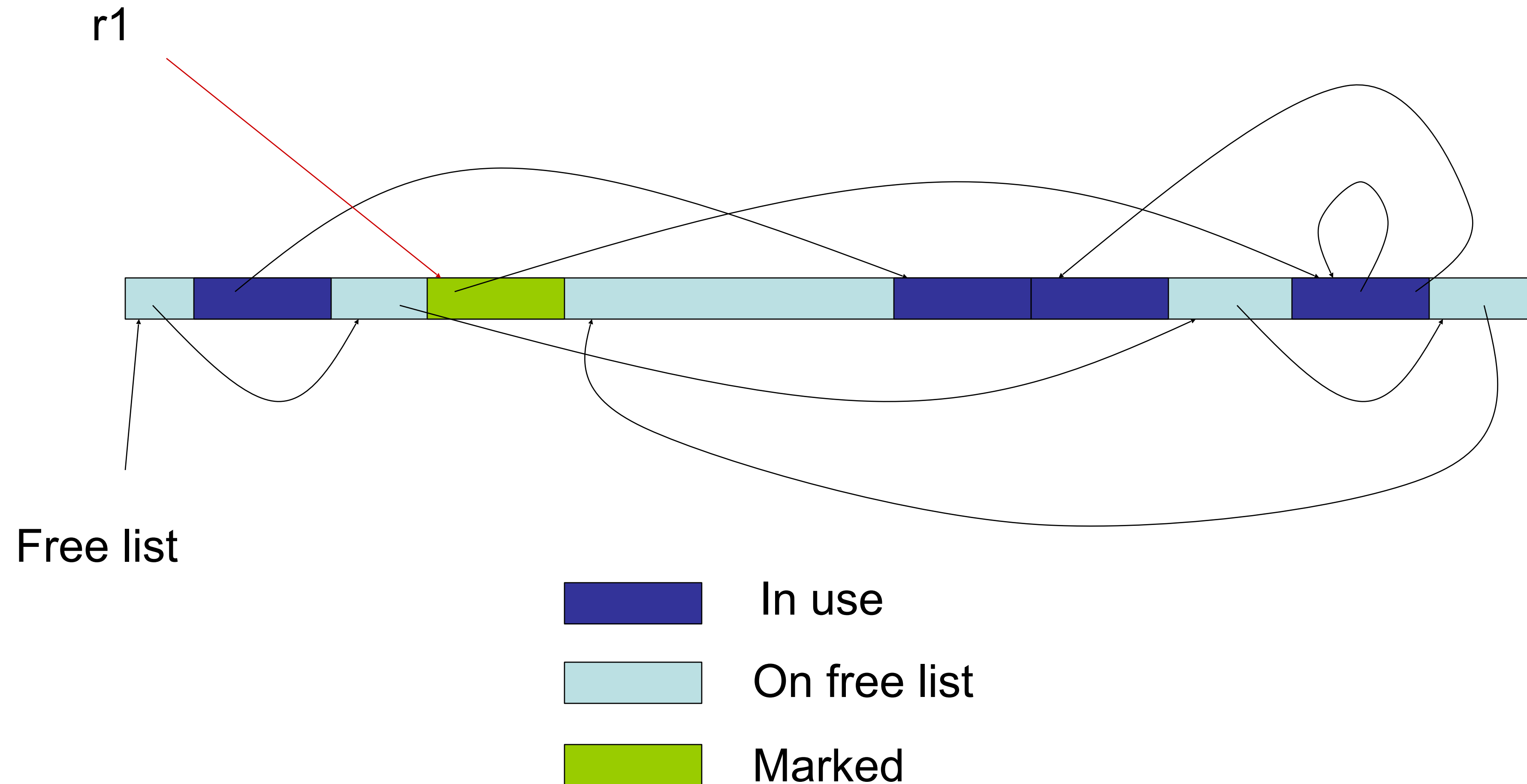


# Example



# Example

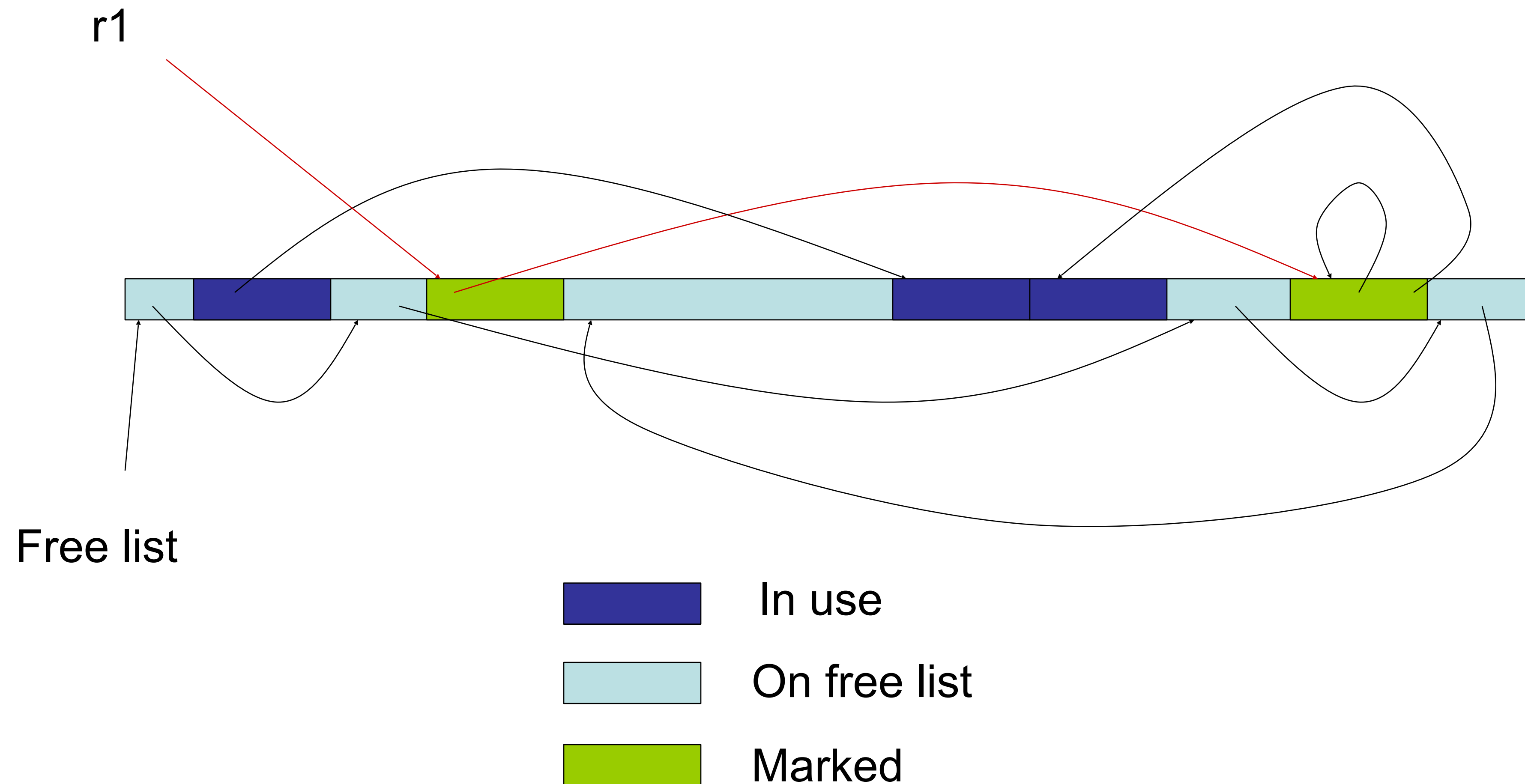
Mark Phase: mark nodes reachable from roots





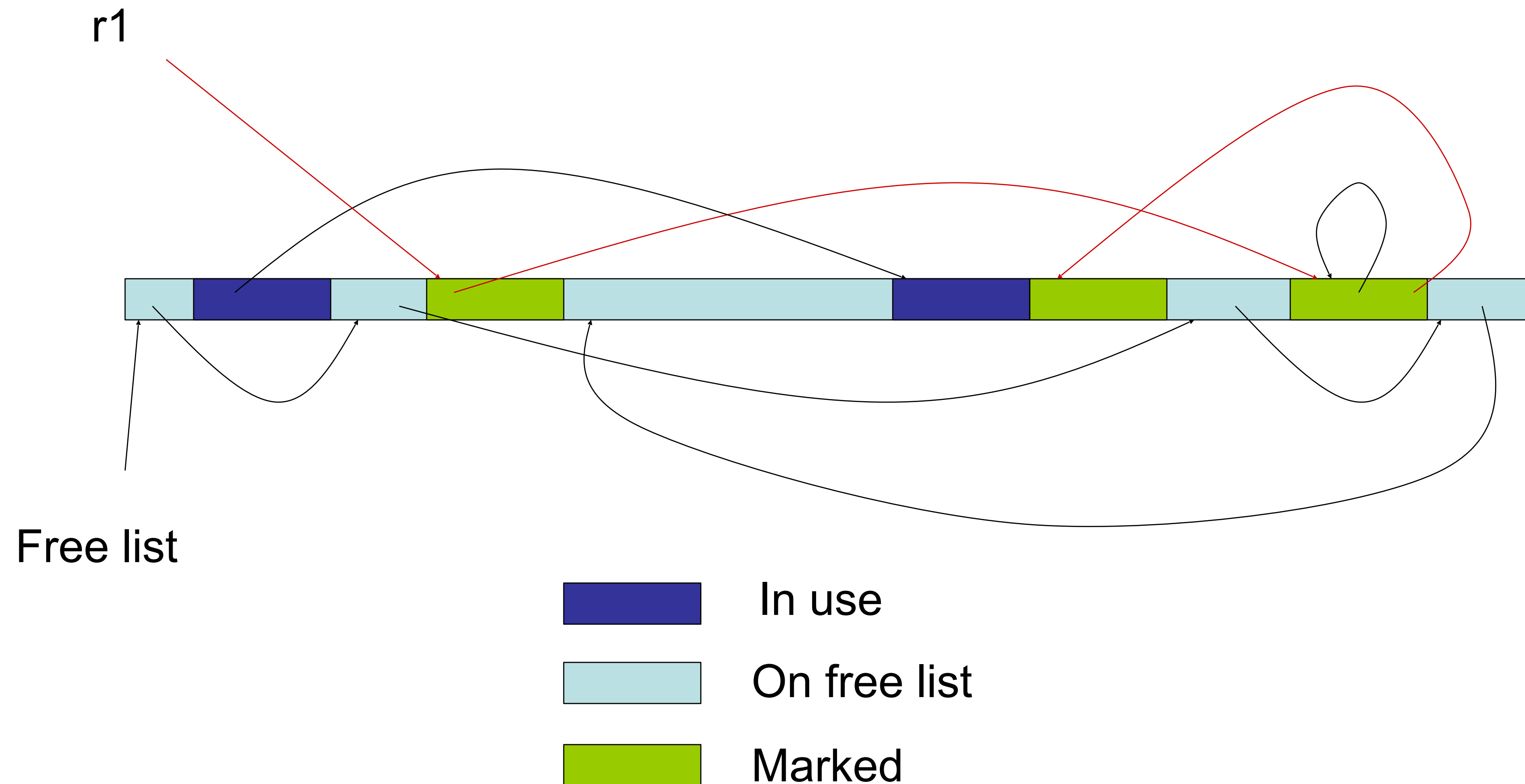
# Example

## Mark Phase: mark nodes reachable from roots



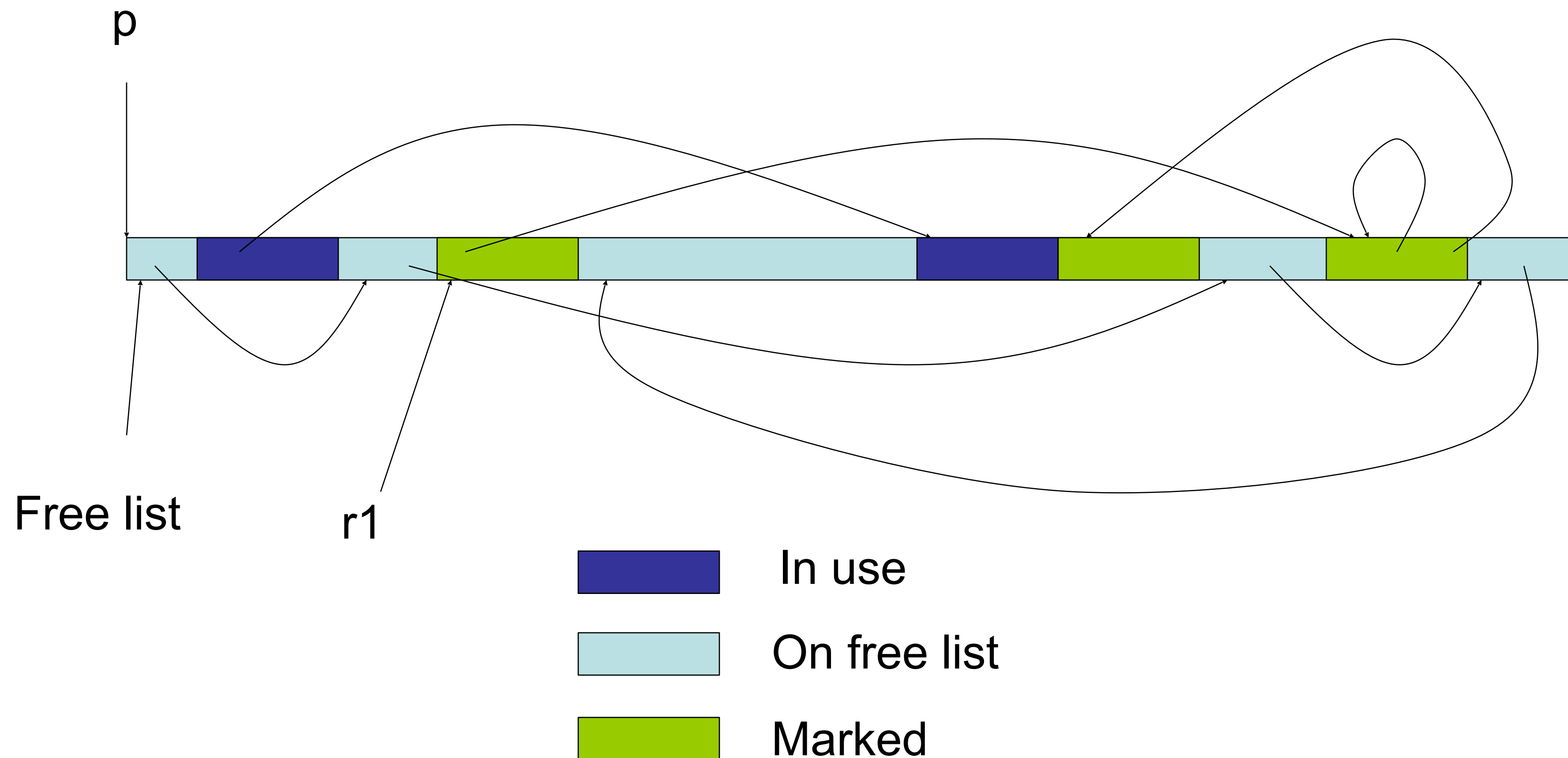
# Example

## Mark Phase: mark nodes reachable from roots



# Example

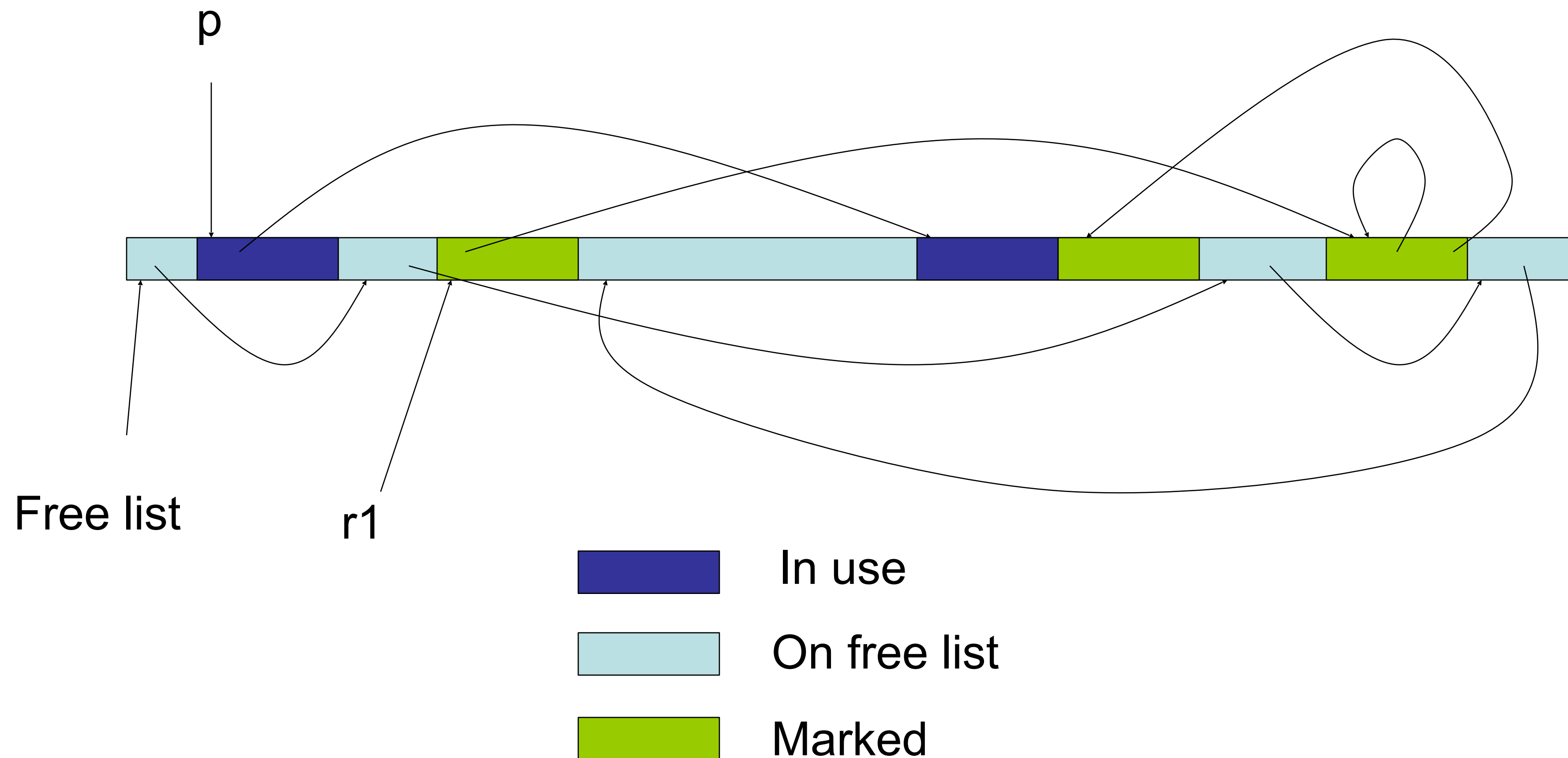
Sweep Phase: set up sweep pointer; begin sweep





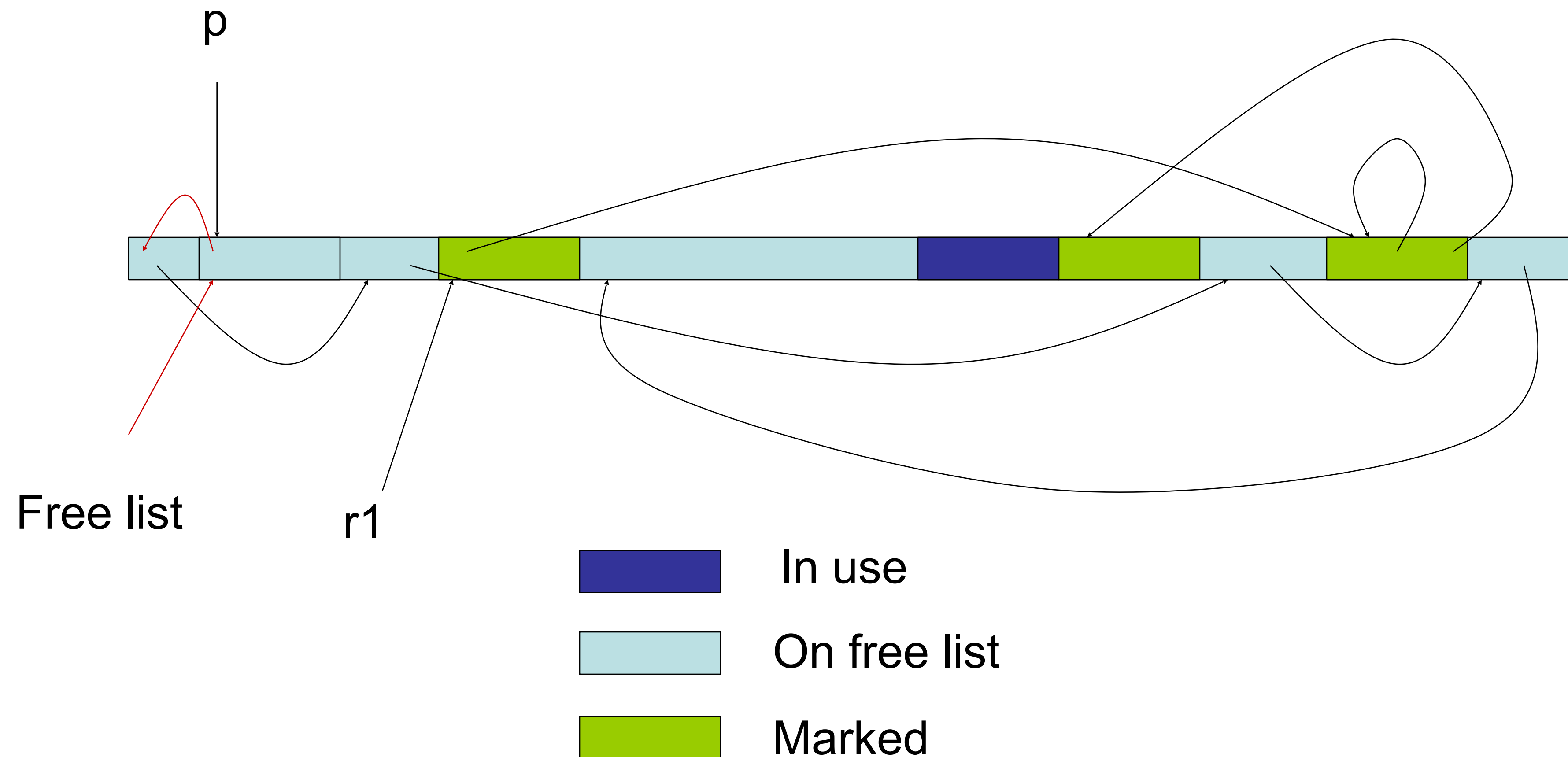
# Example

## Sweep Phase: add unmarked blocks to free list



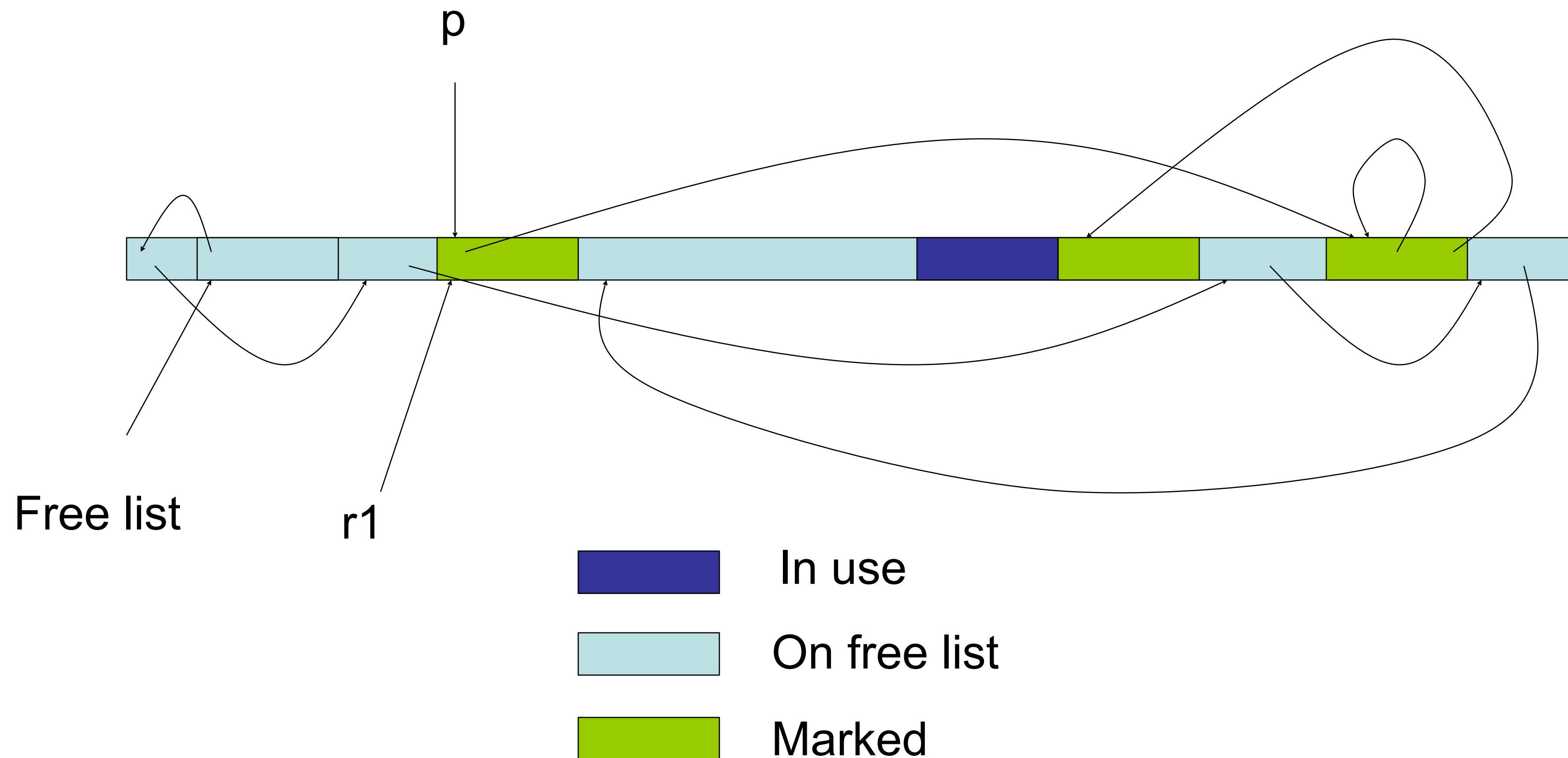
# Example

## Sweep Phase



# Example

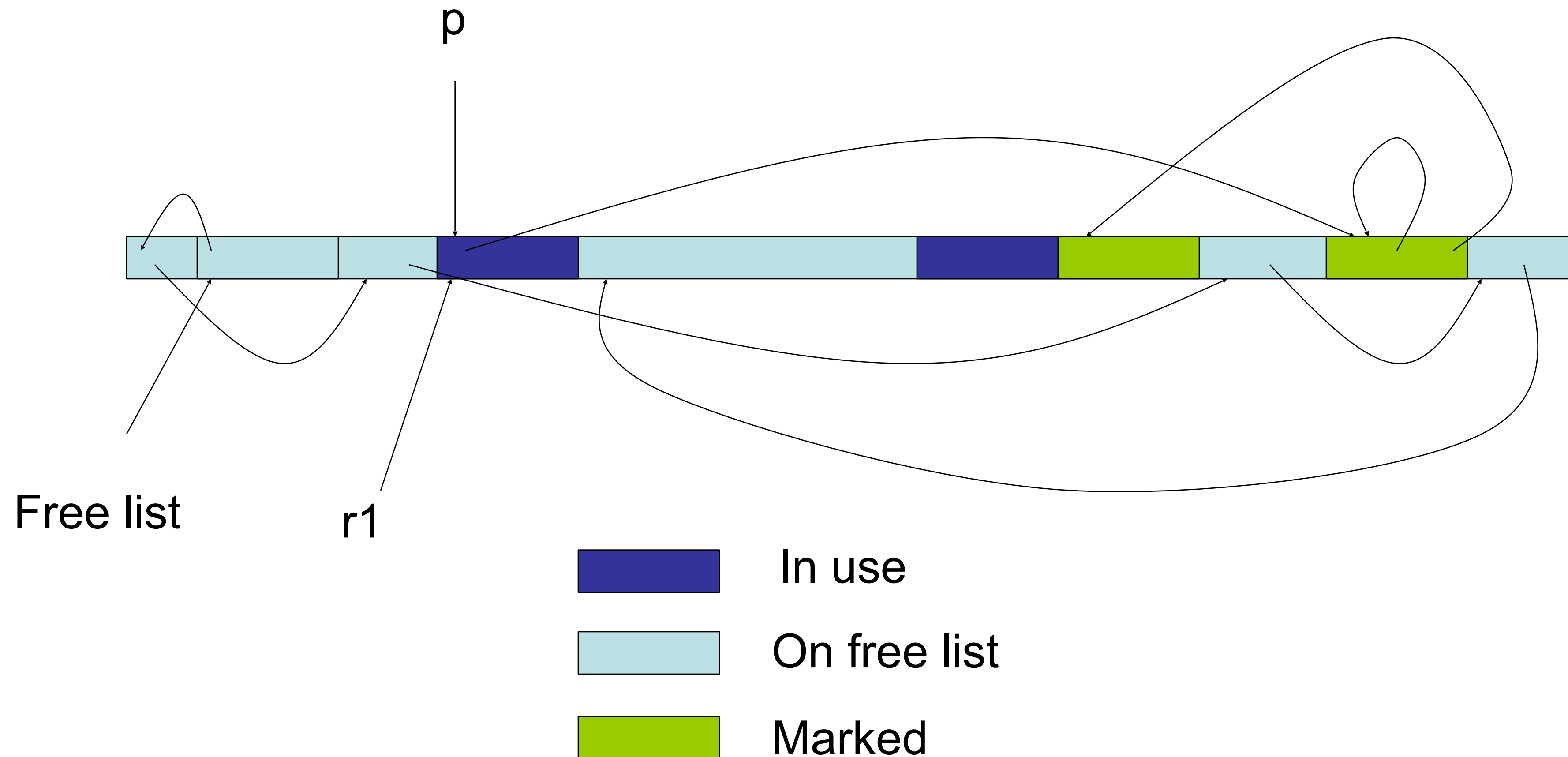
Sweep Phase: retain & unmark marked blocks





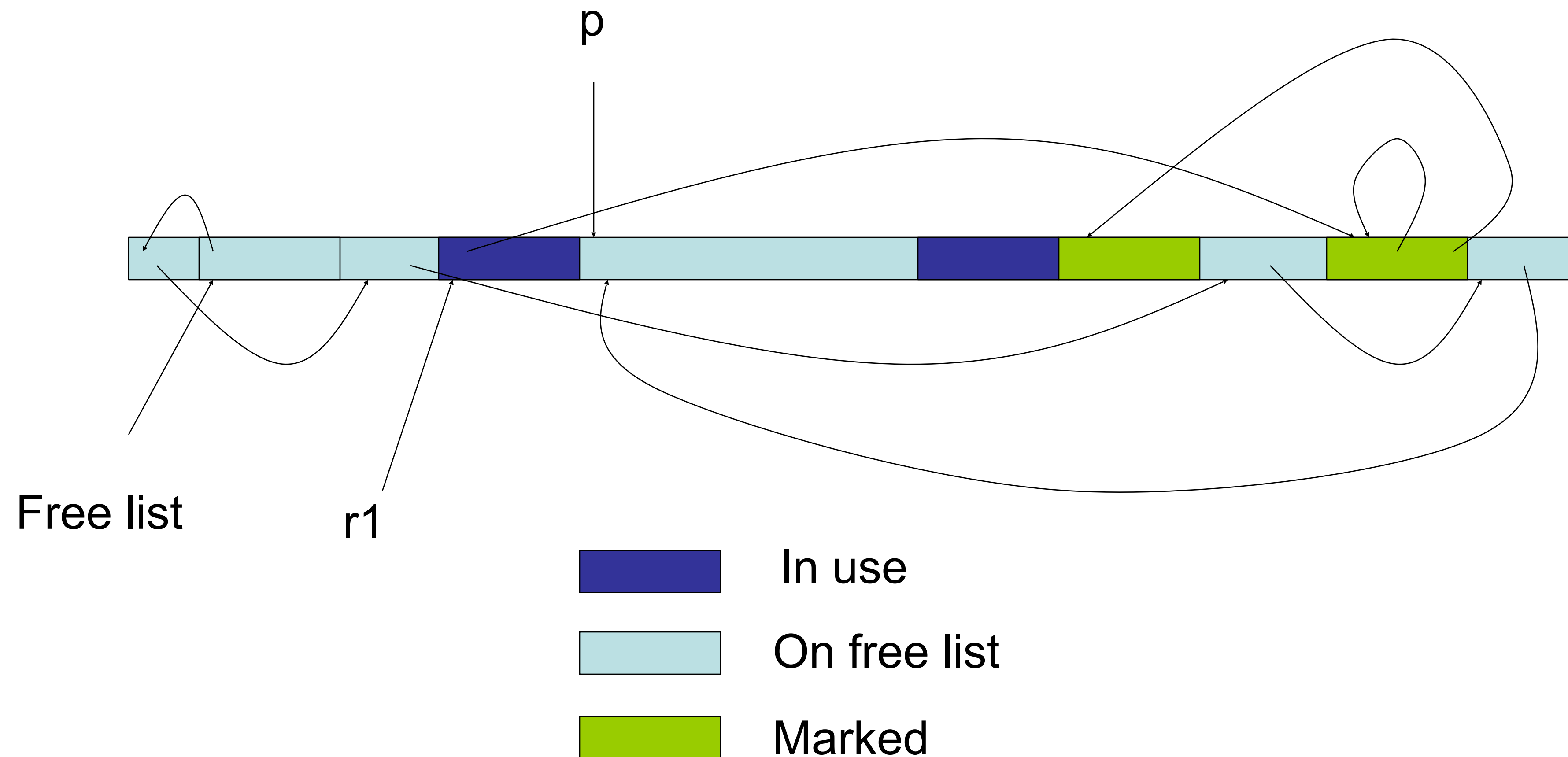
# Example

## Sweep Phase



# Example

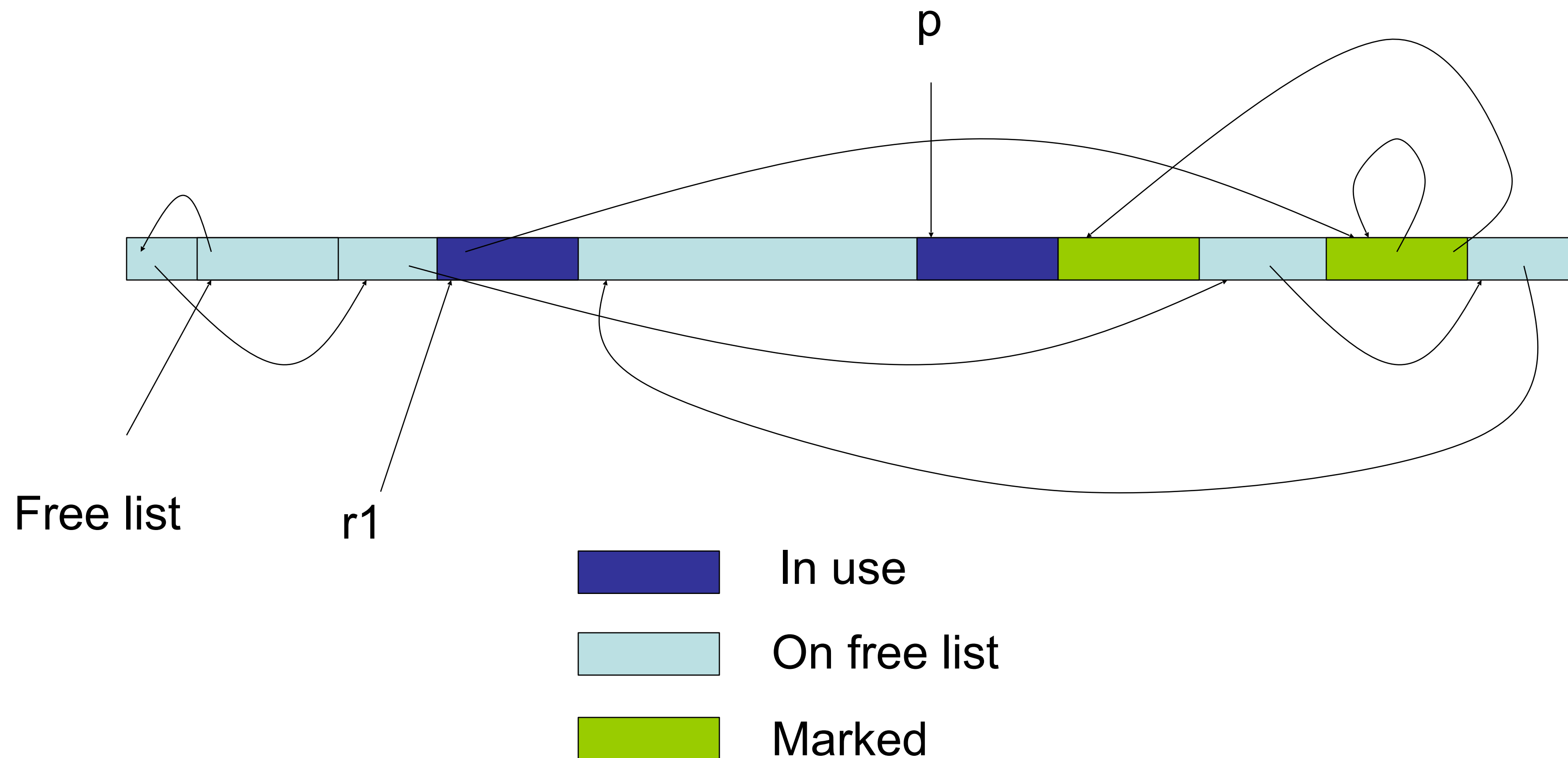
## Sweep Phase





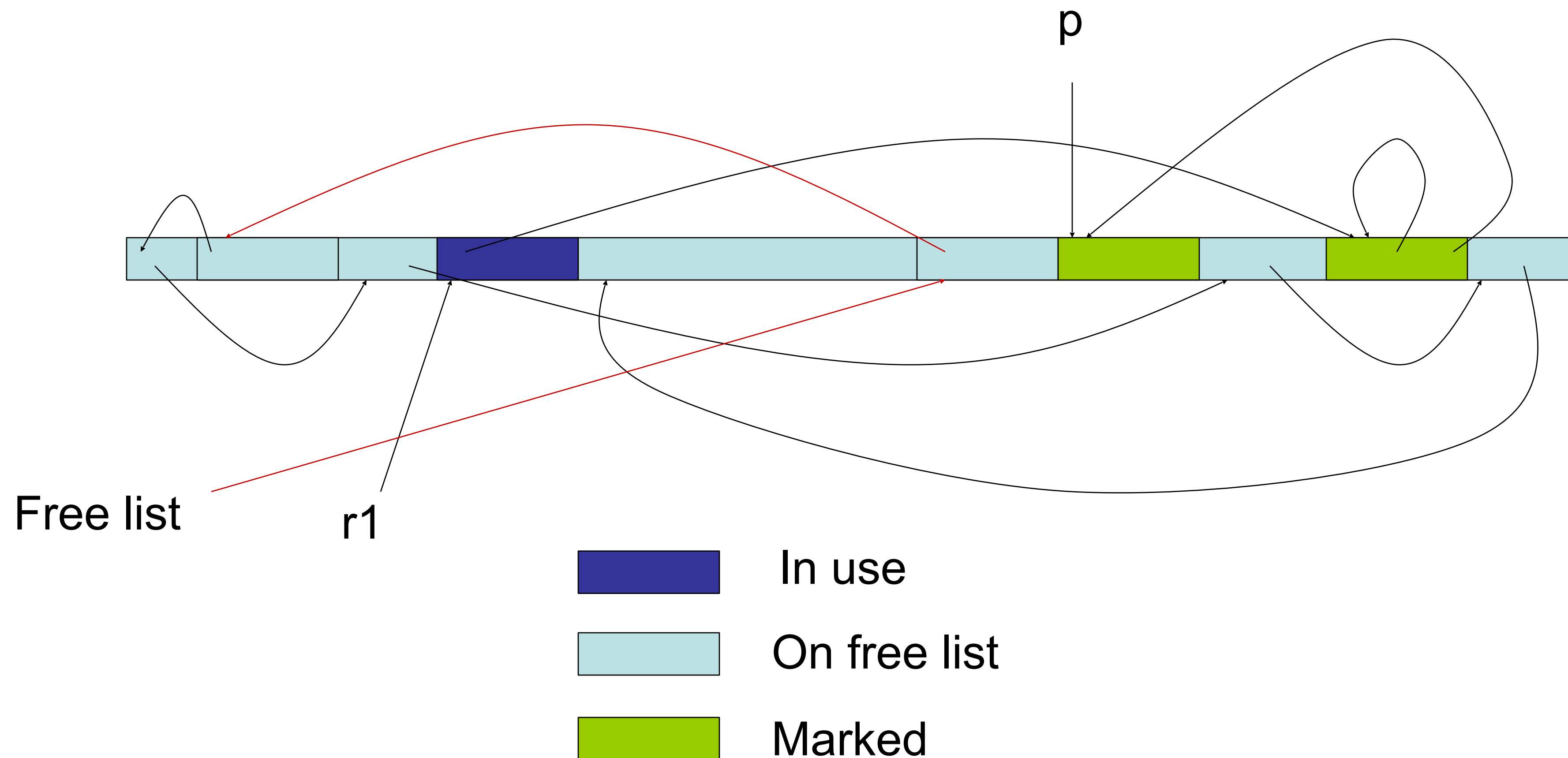
# Example

# Sweep Phase



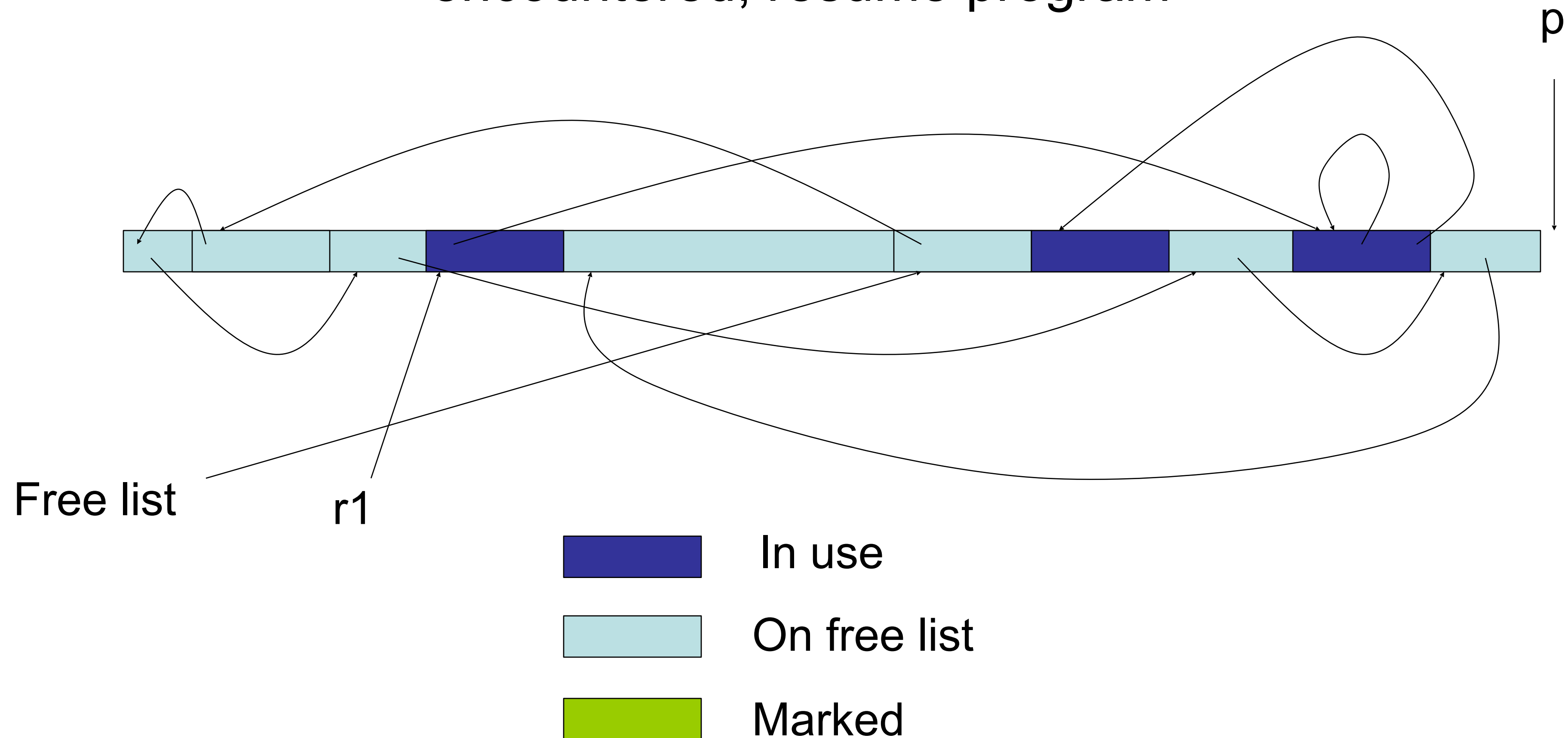
# Example

# Sweep Phase



# Example

Sweep Phase: GC complete when heap boundary encountered; resume program





# Cost of Mark Sweep

- Cost of mark phase:
  - $O(R)$  where  $R$  is the # of reachable words
  - Assume cost is  $c1 * R$  ( $c1$  may be 10 instr's)
- Cost of sweep phase:
  - $O(H)$  where  $H$  is the # of words in entire heap
  - Assume cost is  $c2 * H$  ( $c2$  may be 3 instr's)
- Analysis
  - Each collection returns  $H - R$  words
  - For every allocated word, we have GC cost:
    - $((c1 * R) + (c2 * H)) / (H - R)$
  - $R / H$  must be sufficiently small or GC cost is high
  - Eg: if  $R / H$  is larger than .5, increase heap size
    - Mark-sweep requires extra space like copying collection

# A Hidden Cost

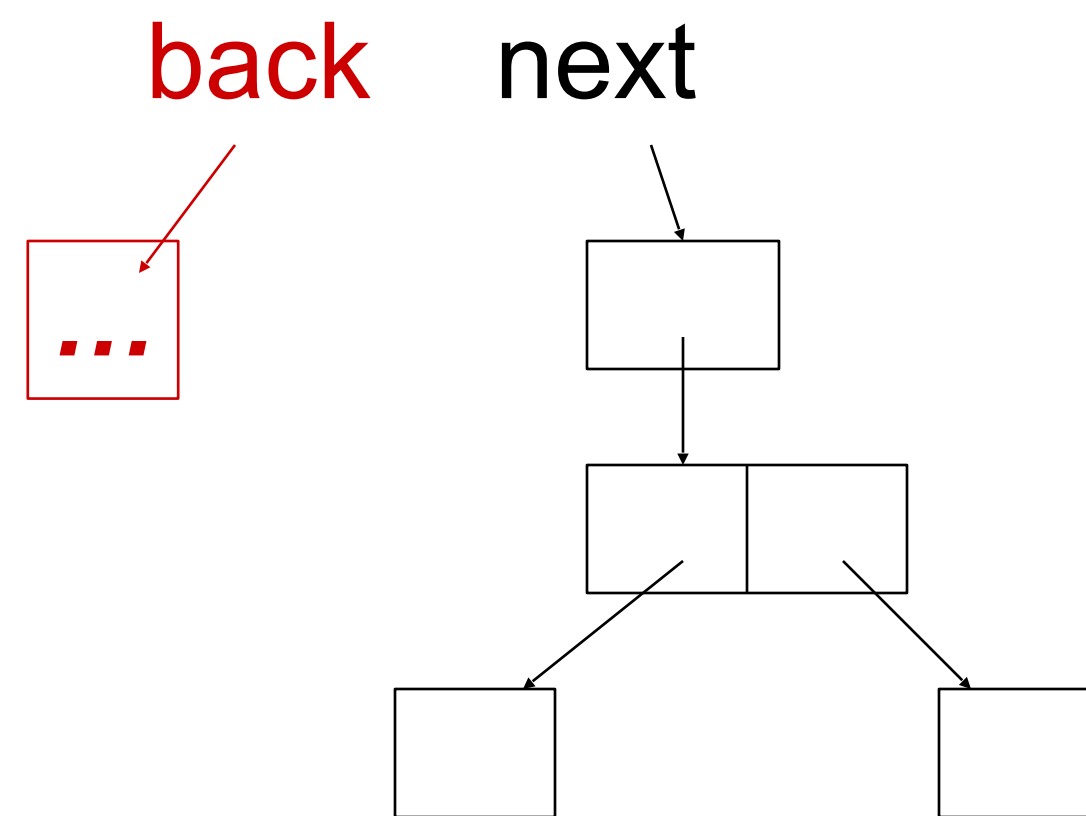
- Depth-first search is usually implemented as a recursive algorithm
  - Uses stack space proportional to the longest path in the graph of reachable objects
    - one activation record/node in the path
    - activation records are big
  - If the heap is one long linked list, the stack space used in the algorithm will be greater than the heap size!!
  - What do we do?

# A nifty trick

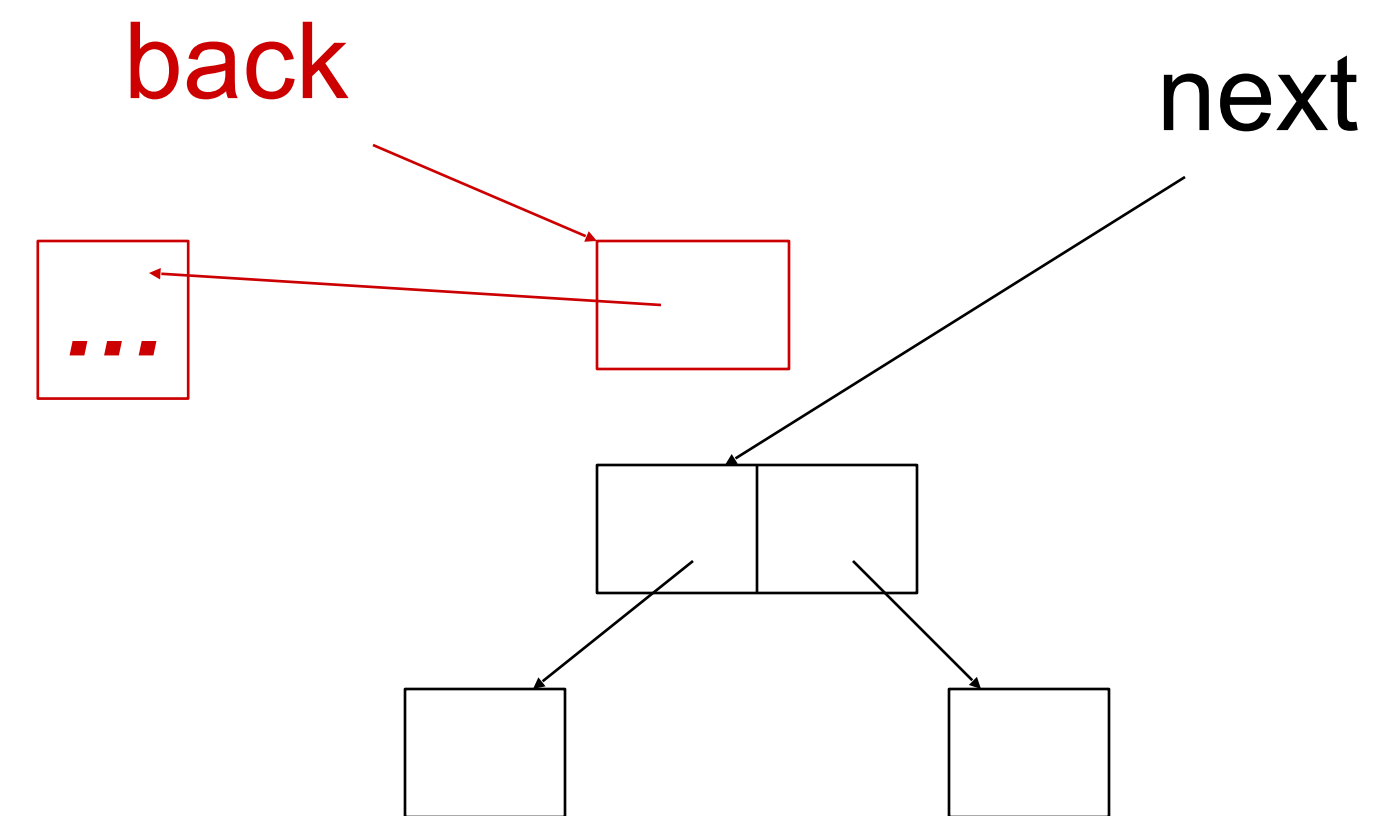
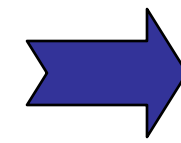
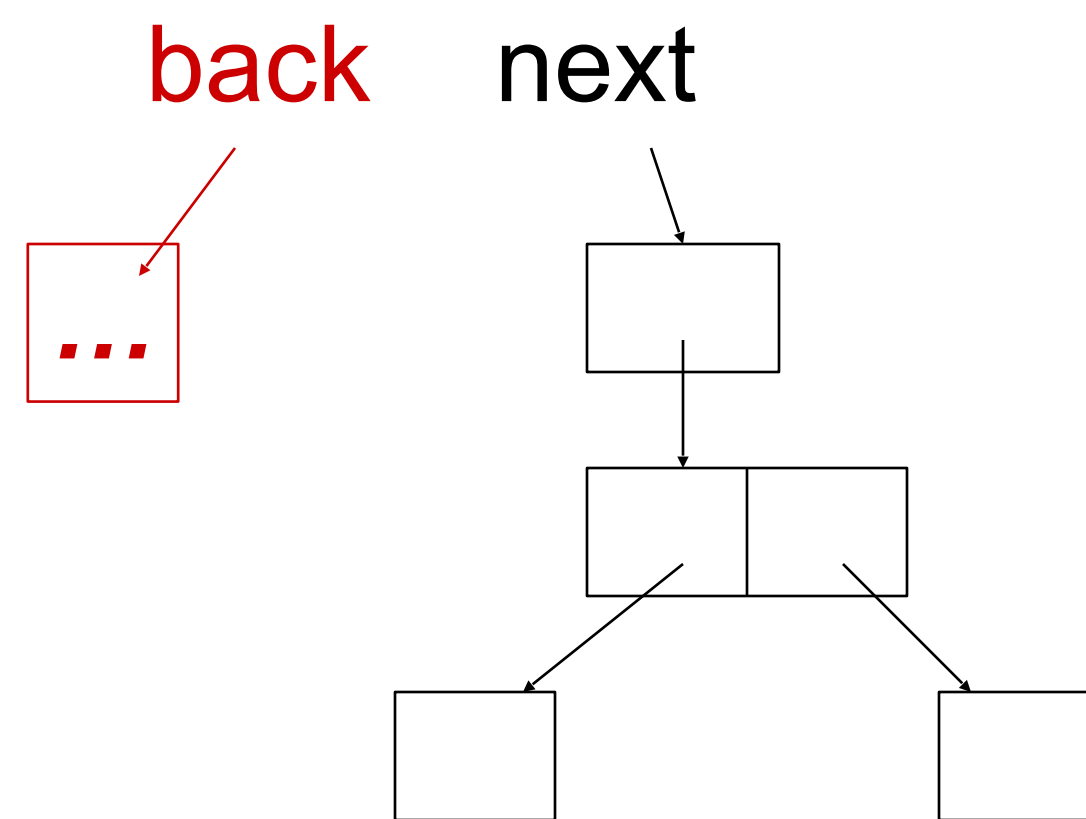
- Deutsch-Schorr-Waite pointer reversal
  - Rather using a recursive algorithm, reuse the components of the graph you are traversing to build an explicit stack
  - This implementation trick only demands a few extra bits/block rather than an entire activation record/block
  - We already needed a few extra bits per block to hold the “mark” anyway



# DSW Algorithm

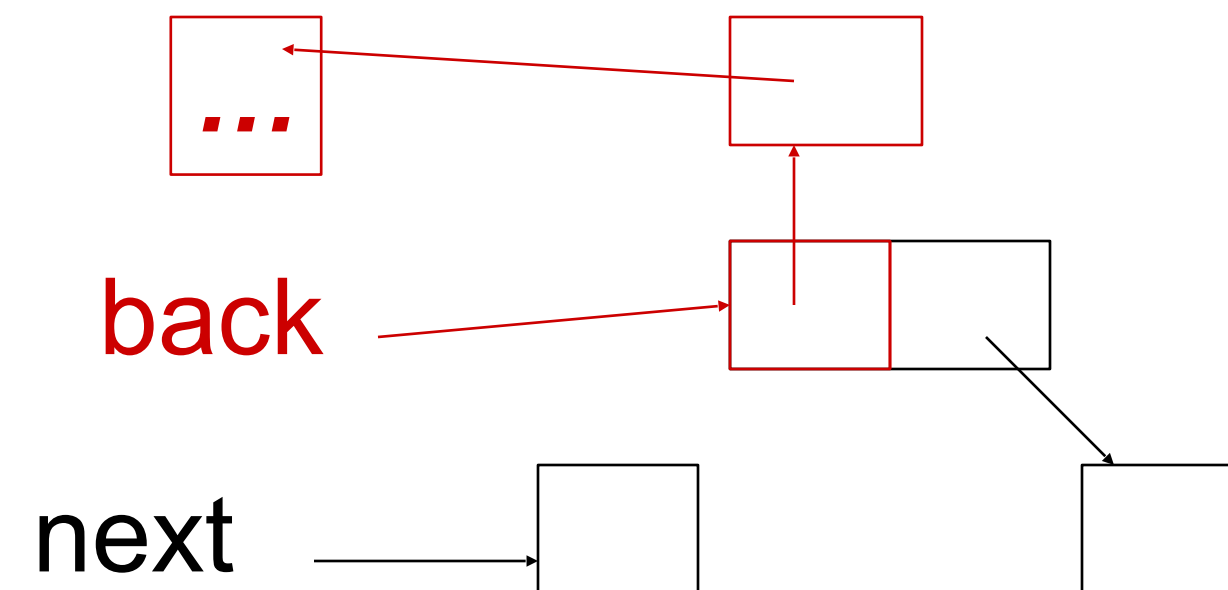
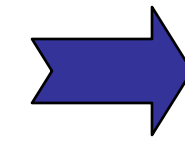
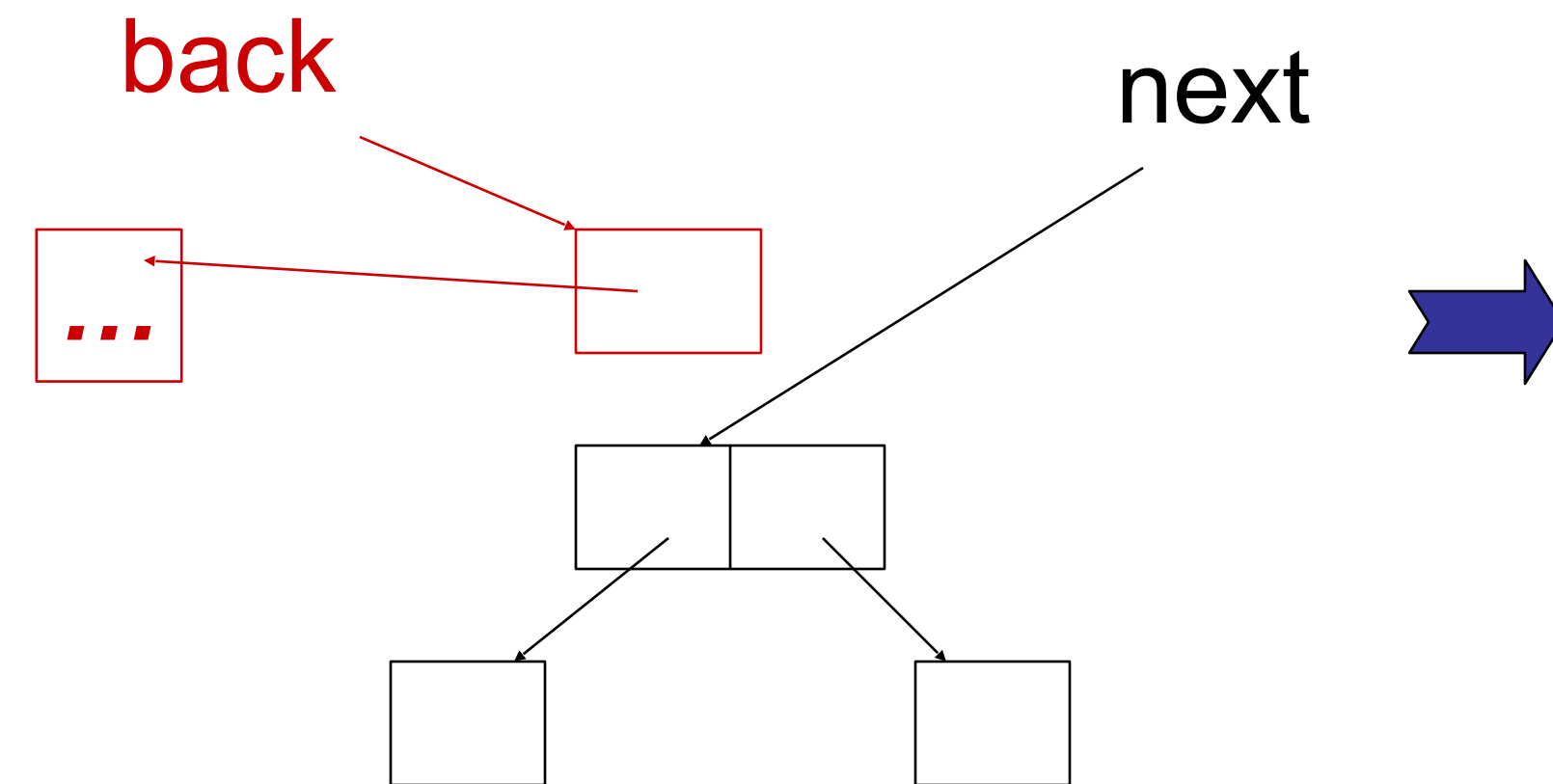
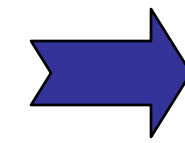
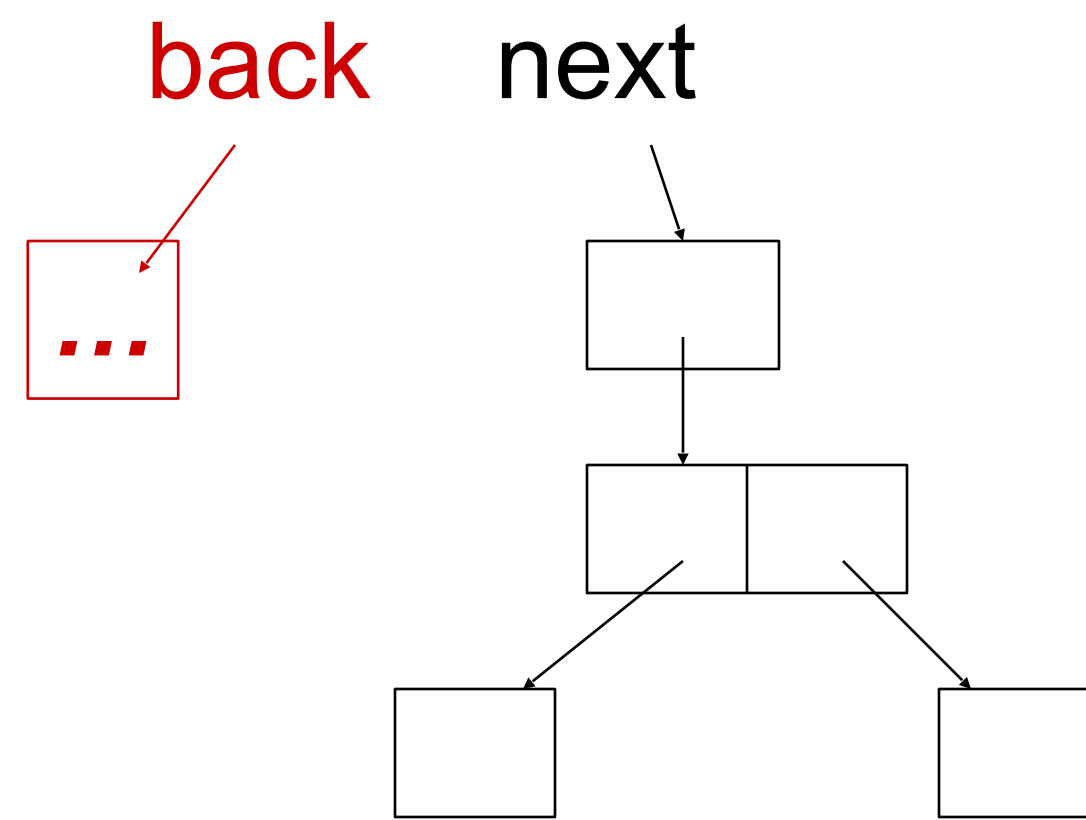


# DSW Algorithm

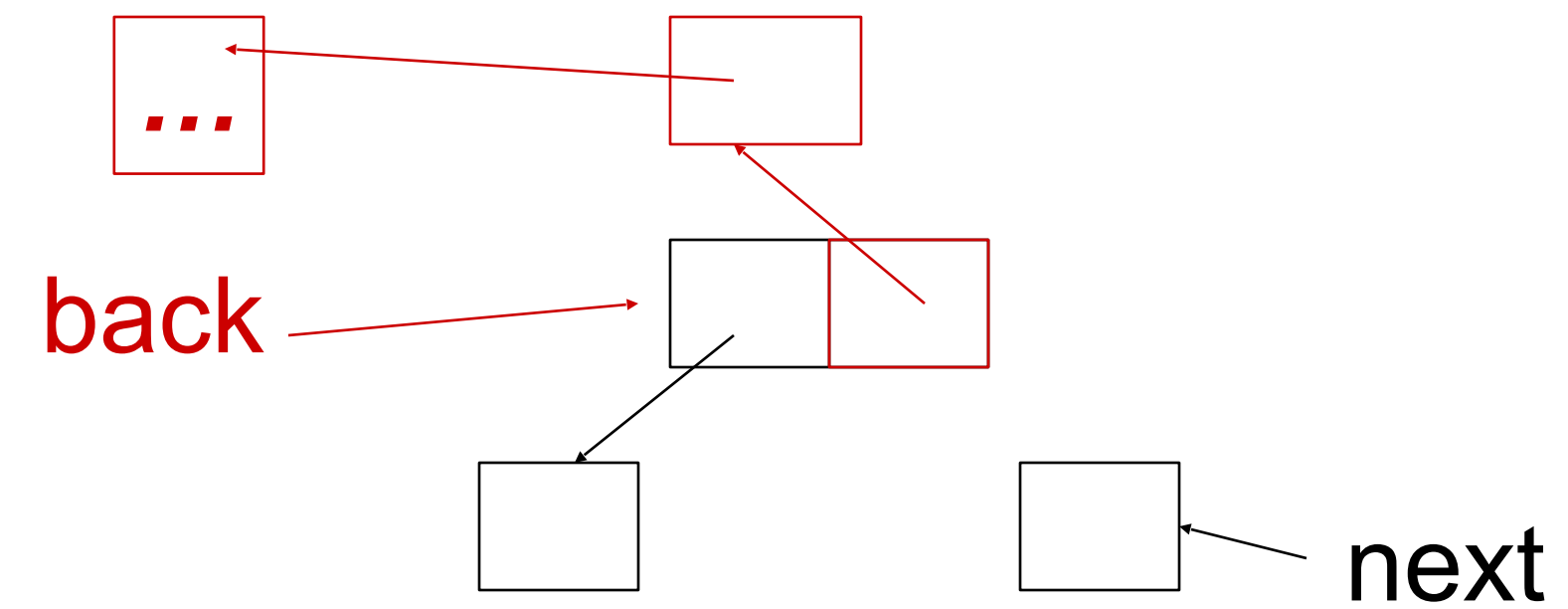
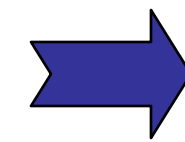
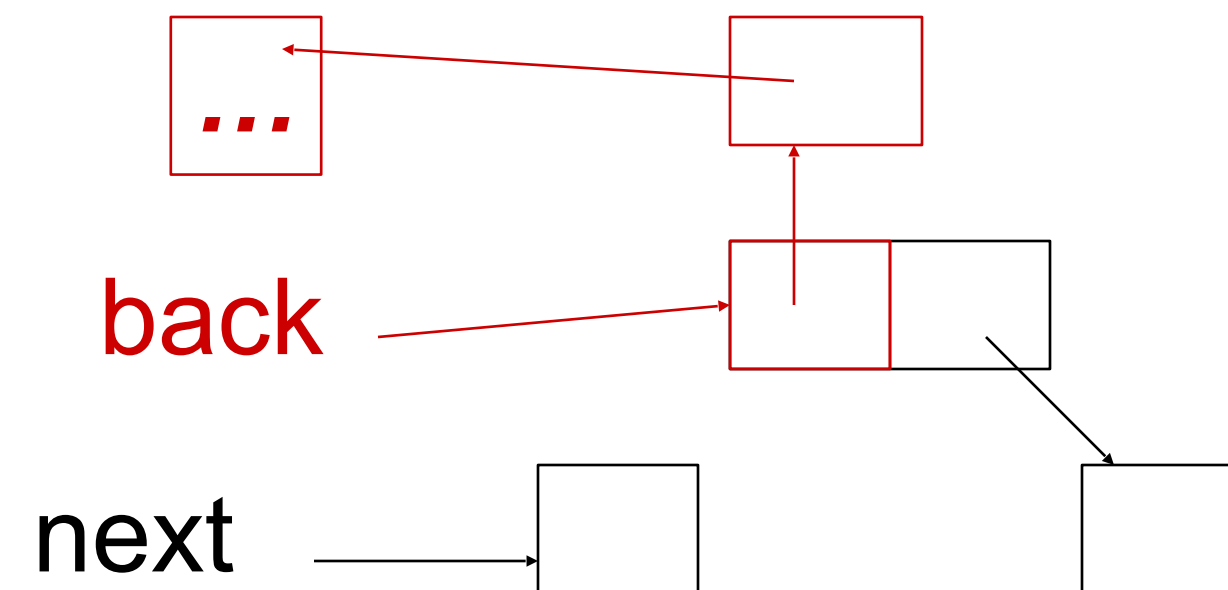
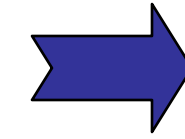
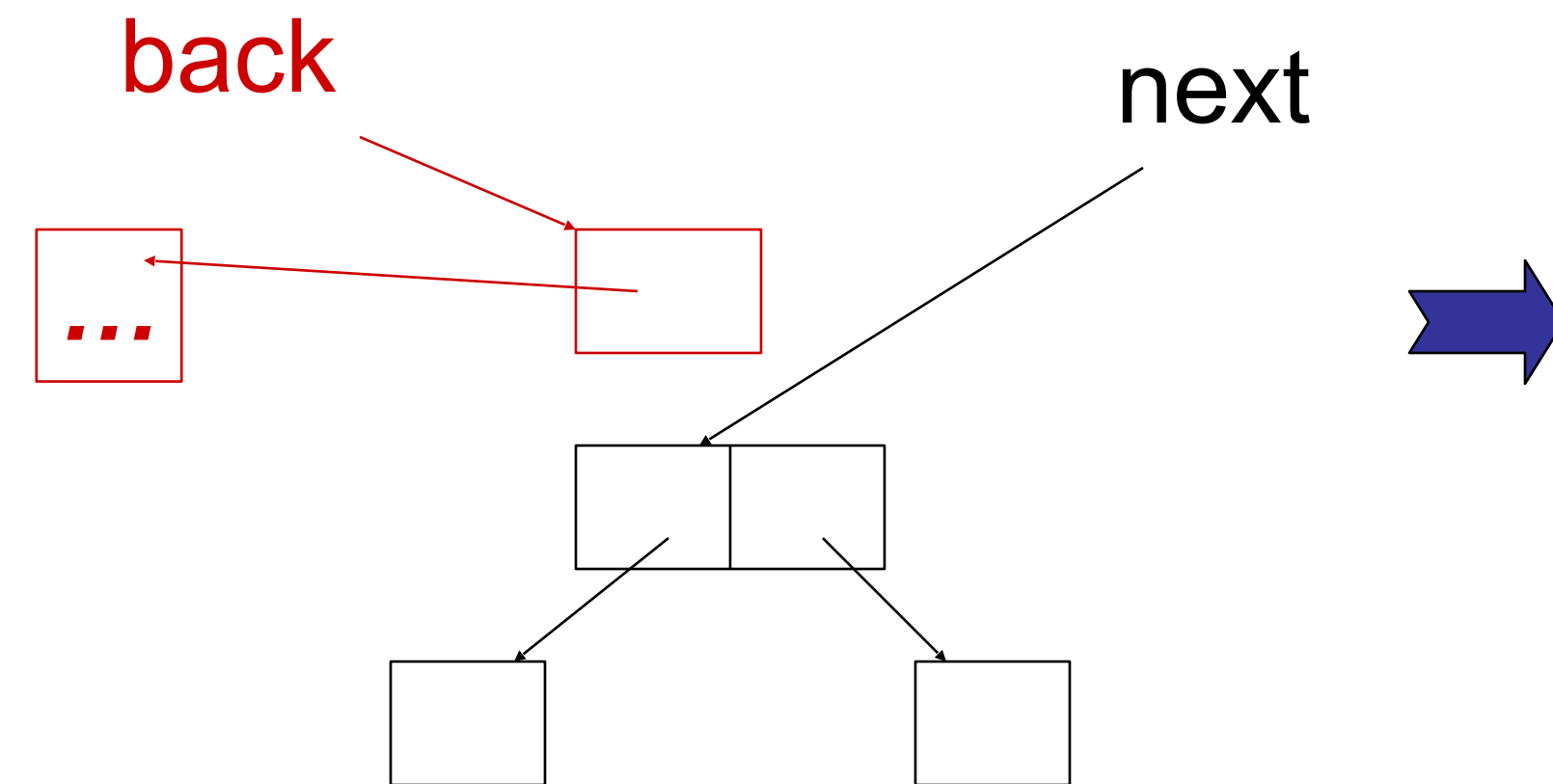
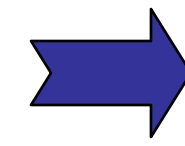
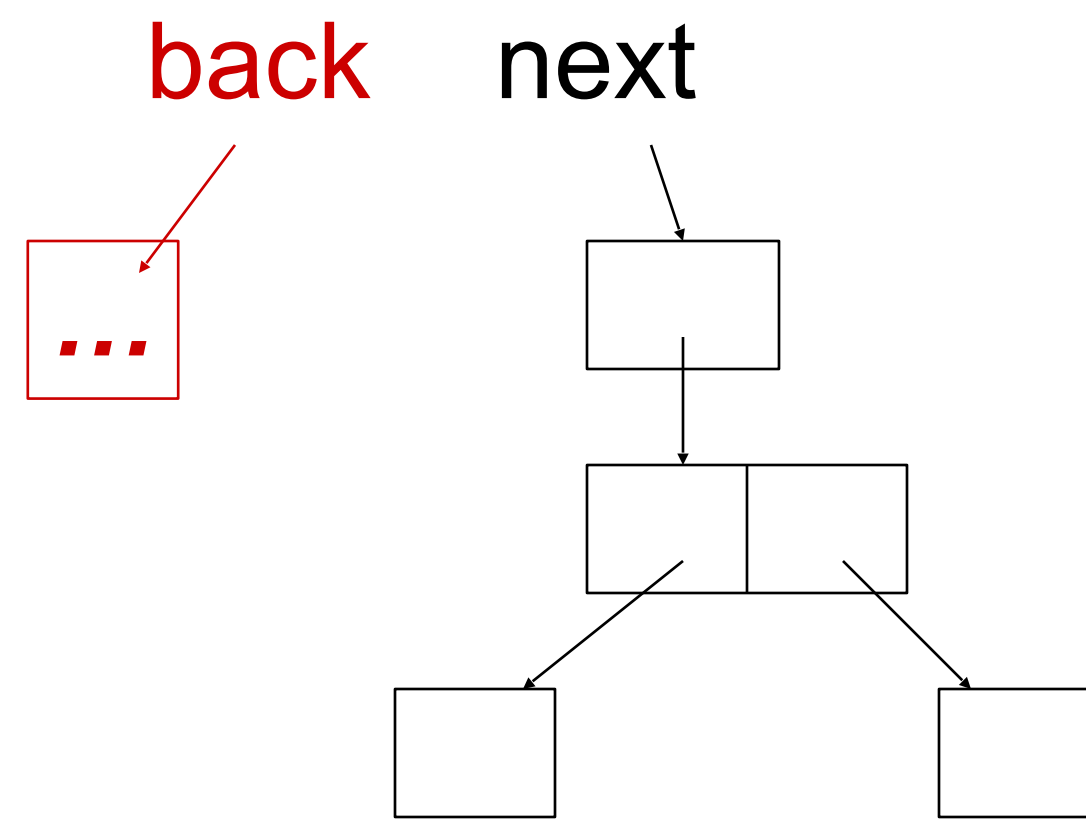




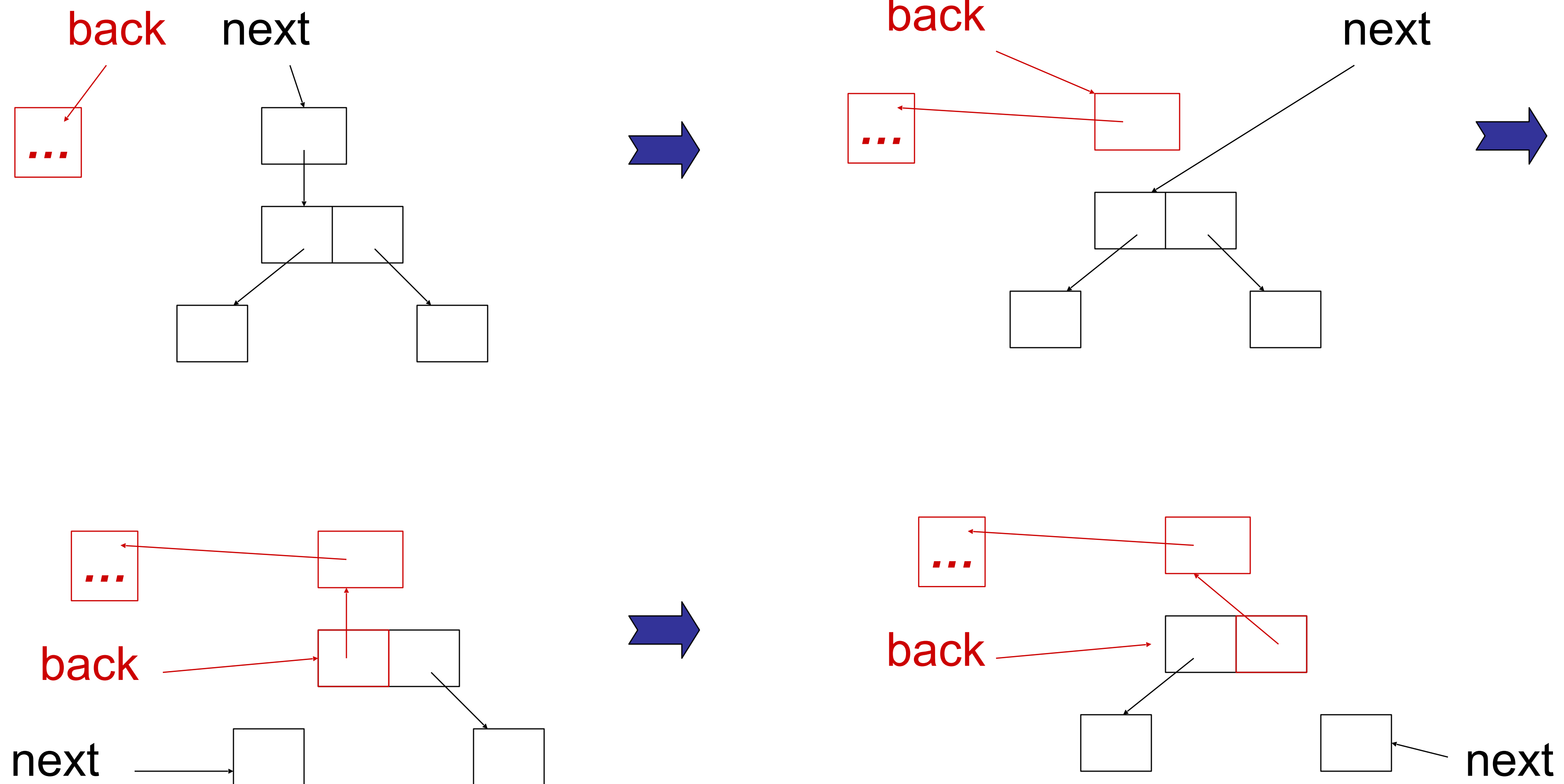
# DSW Algorithm



# DSW Algorithm



# DSW Algorithm



- extra bits needed to keep track of which record fields we have processed so far



# DSW Setup

- Extra space required for sweep:
  - 1 bit/record to keep track of whether the record has been seen (the “mark bit”)
  - $f \log 2$  bits/record where  $f$  is the number of fields in the record to keep track of how many fields have been processed
    - assume a field of each record  $x$ :  $x.done$
- Functions:
  - **mark  $x$**  = sets  $x$ 's mark bit
  - **marked  $x$**  = true if  $x$ 's mark bit is set
  - **pointer  $x$**  = true if  $x$  is a pointer
  - **fields  $x$**  = returns number of fields in the record  $x$

# More Mark-Sweep

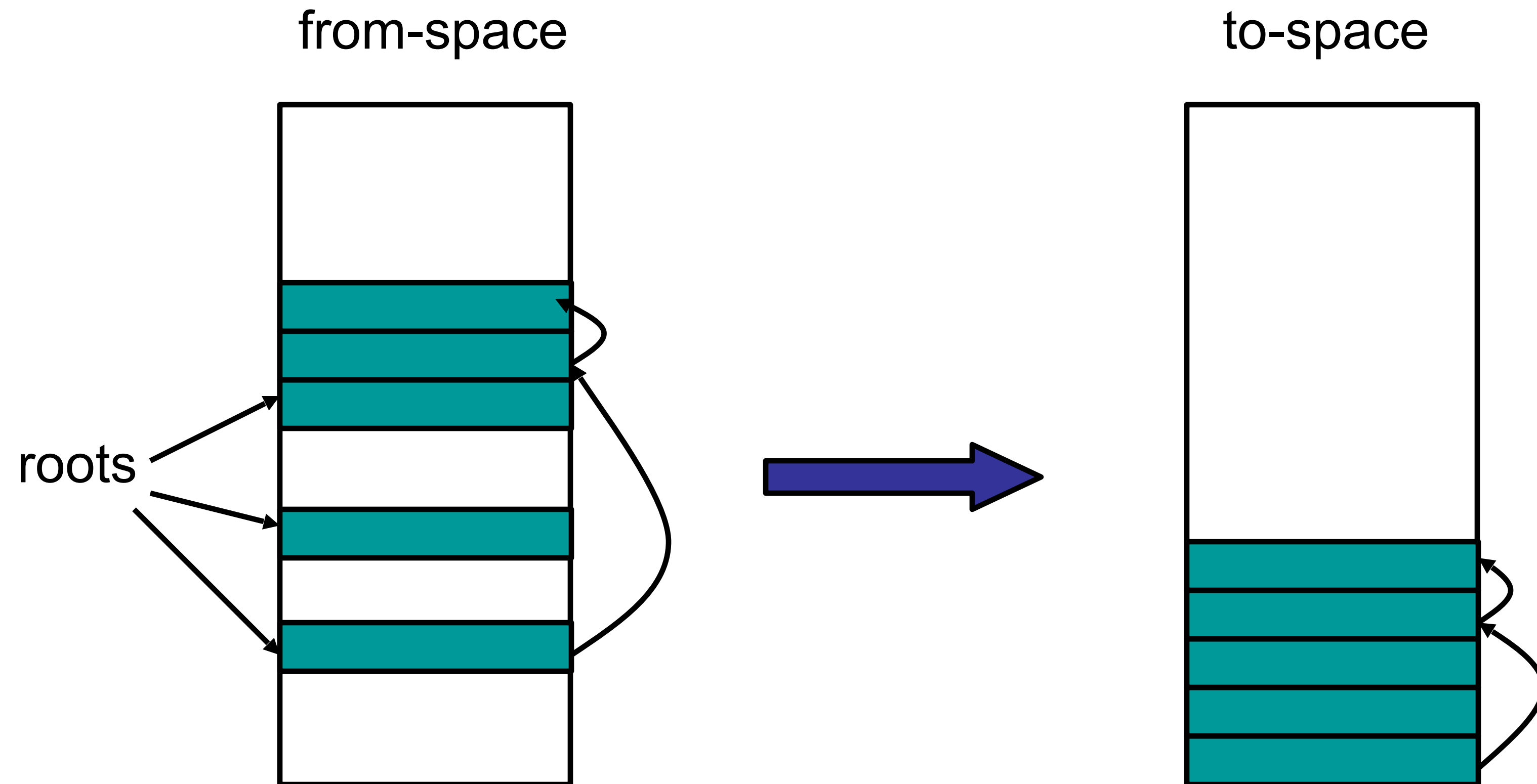
- Mark-sweep collectors can benefit from the tricks used to implement malloc/free efficiently
  - multiple free lists, one size of block/list
- Mark-sweep can suffer from fragmentation
  - blocks not copied and compacted like in copying collection
- Mark-sweep maximum space usage is the total heap size
  - but if the ratio of live data to heap size is too large then performance suffers

# Copying Collection

- Basic idea: use 2 heaps
  - One used by program
  - The other unused until GC time
- GC:
  - Start at the roots & traverse the reachable data
  - Copy reachable data from the active heap (from-space) to the other heap (to-space)
  - Dead objects are left behind in from space
  - Heaps switch roles

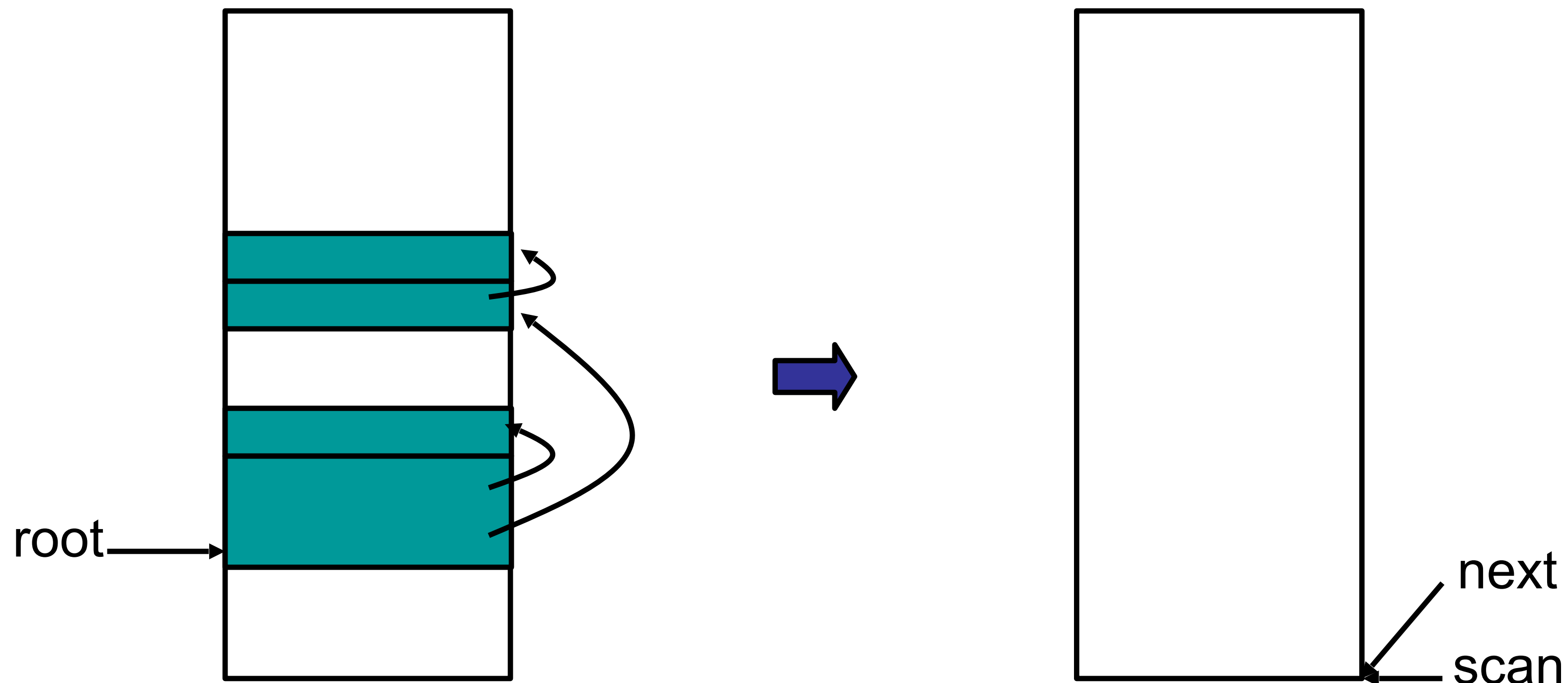


# Copying Collection



# Copying GC

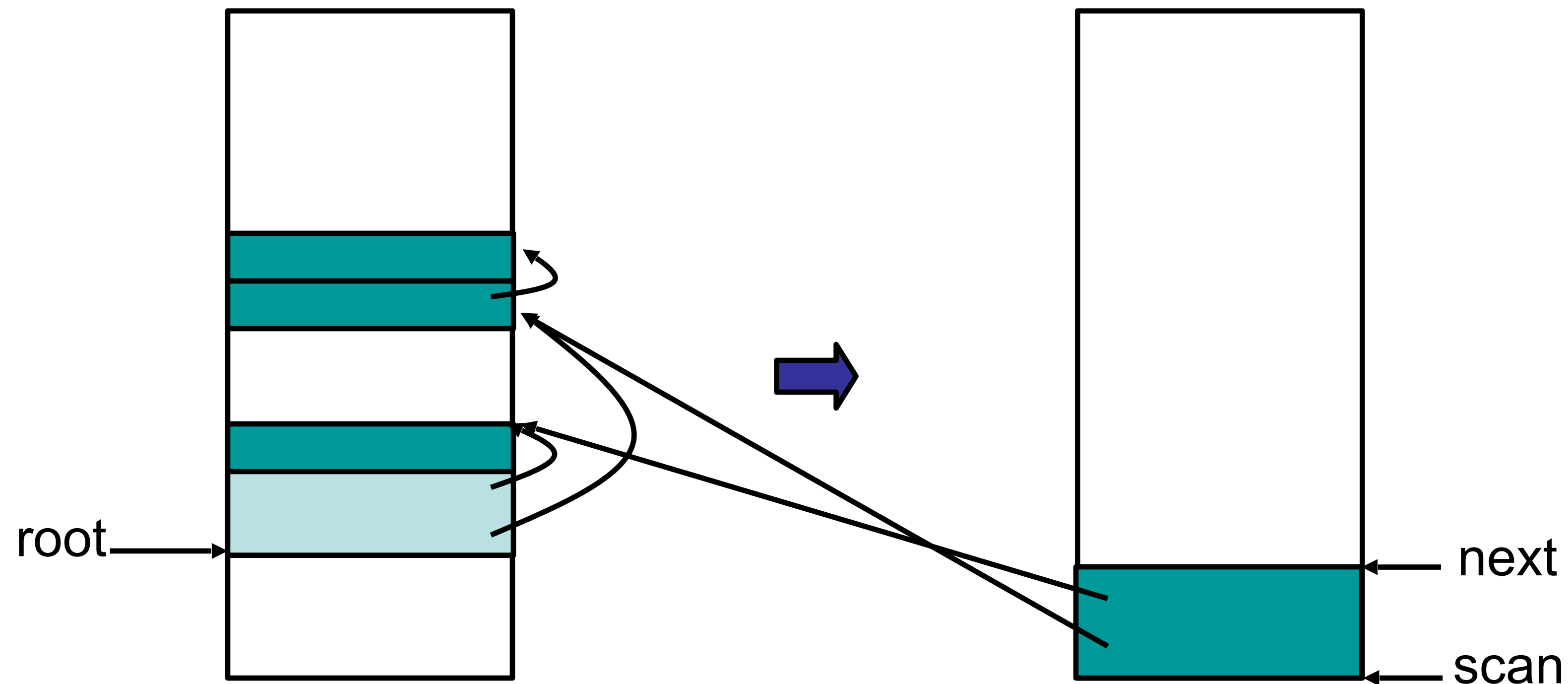
- Cheney's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space





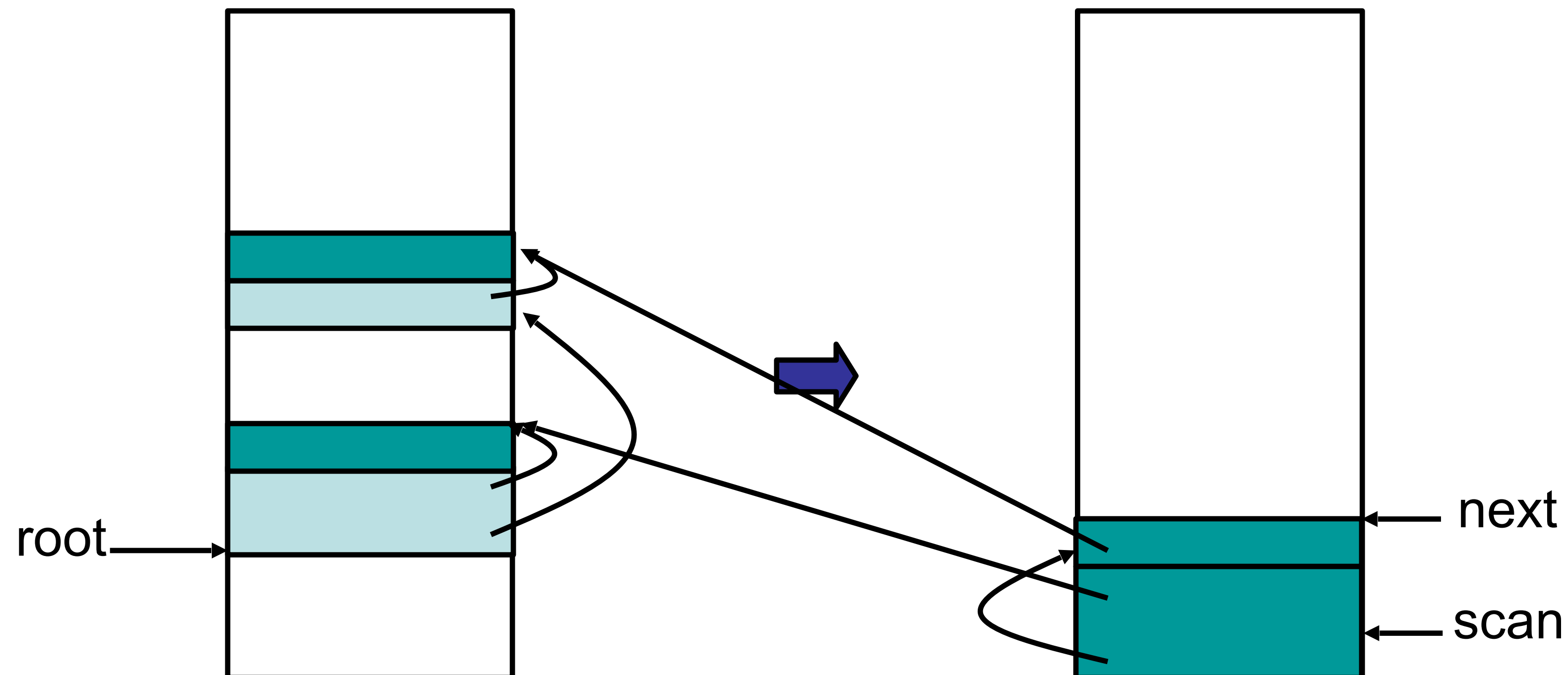
# Copying GC

- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



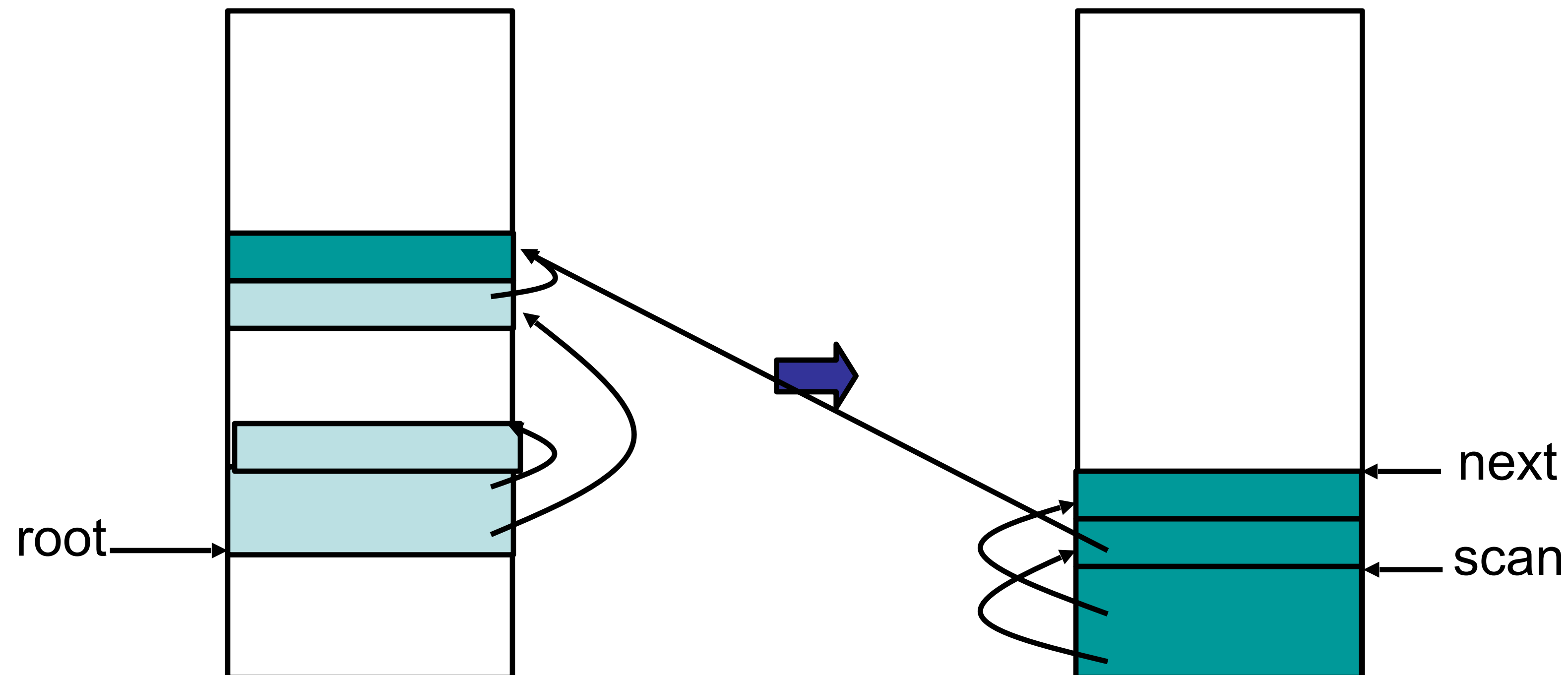
# Copying GC

- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



# Copying GC

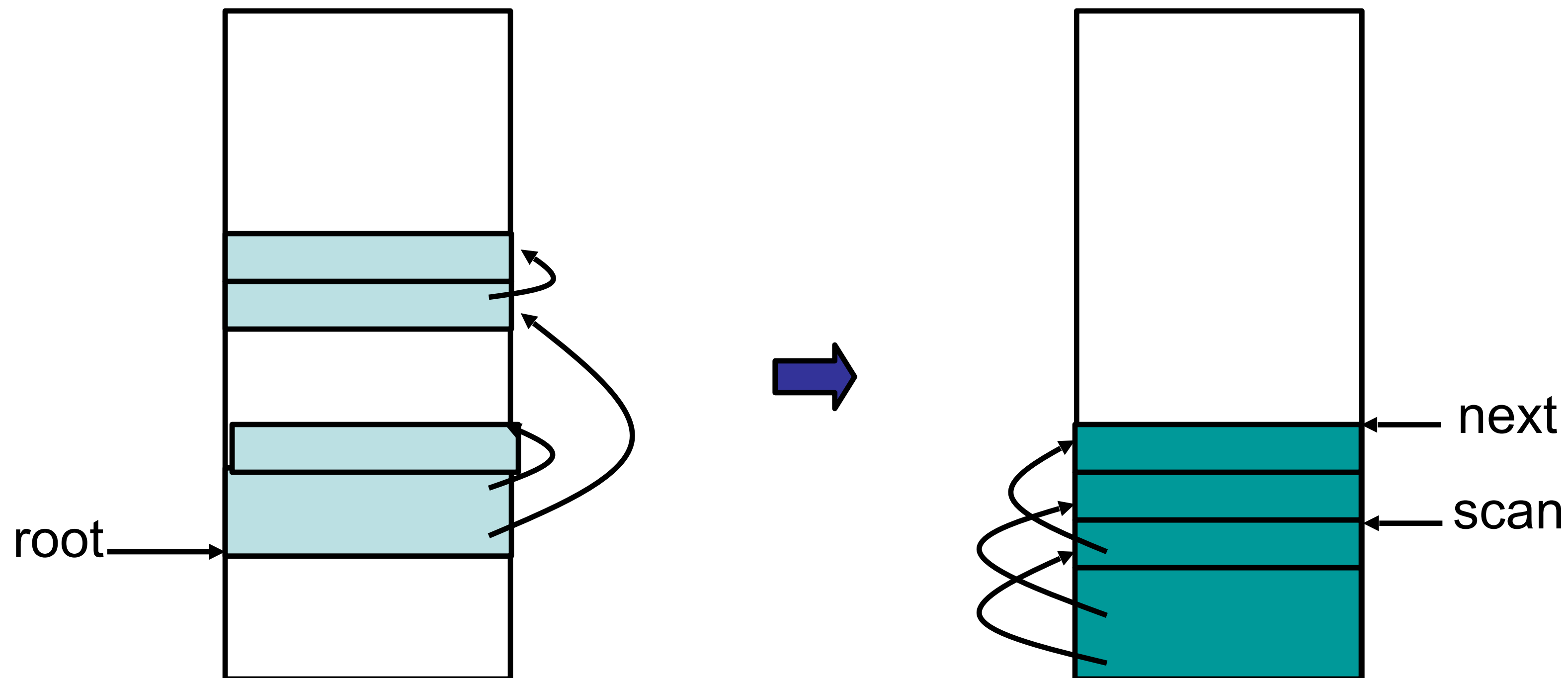
- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space





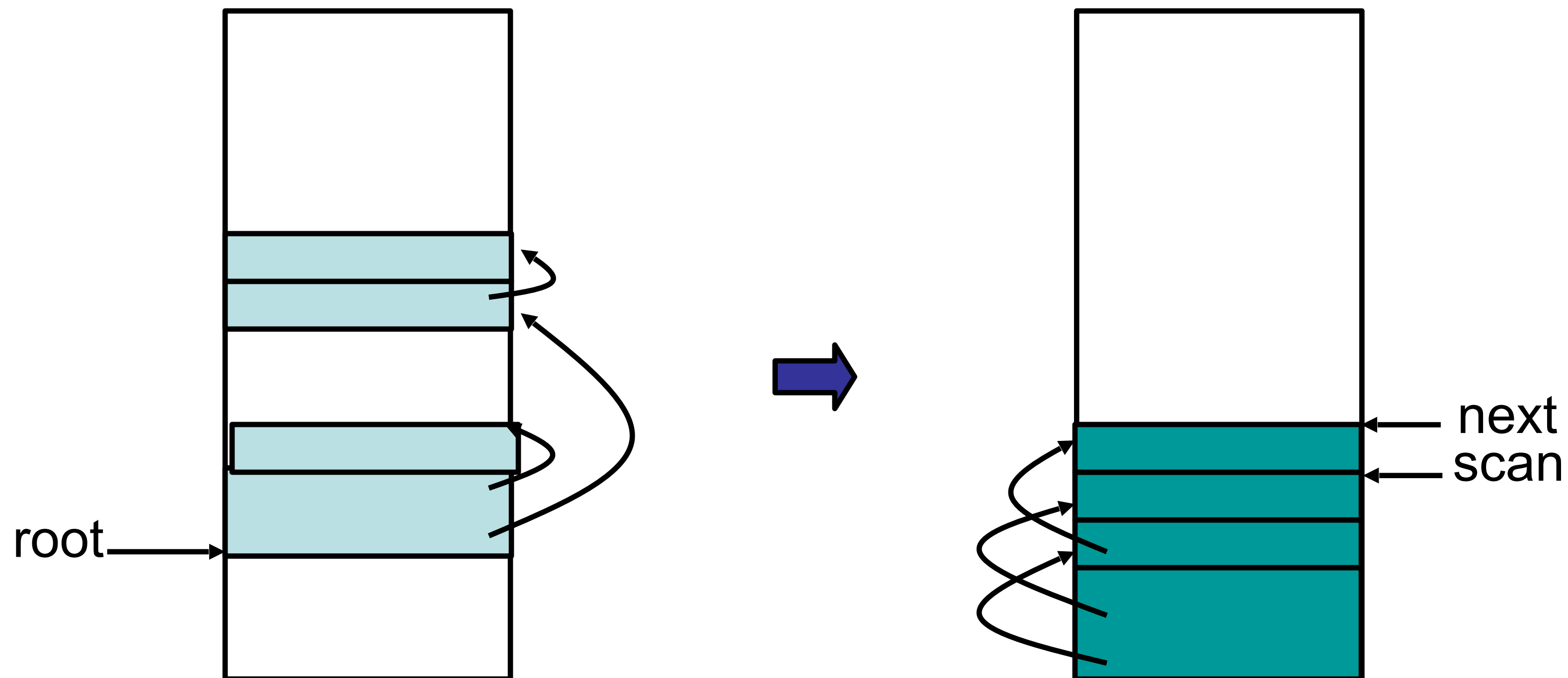
# Copying GC

- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



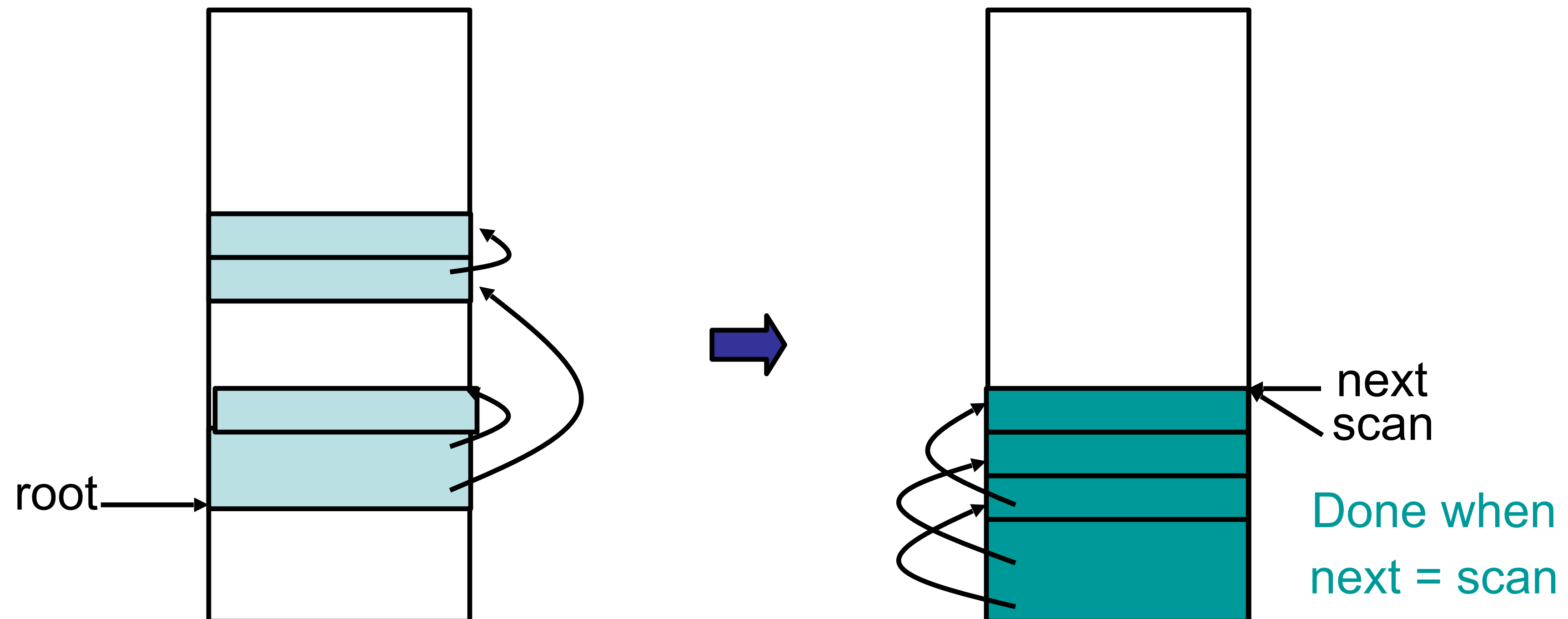
# Copying GC

- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



# Copying GC

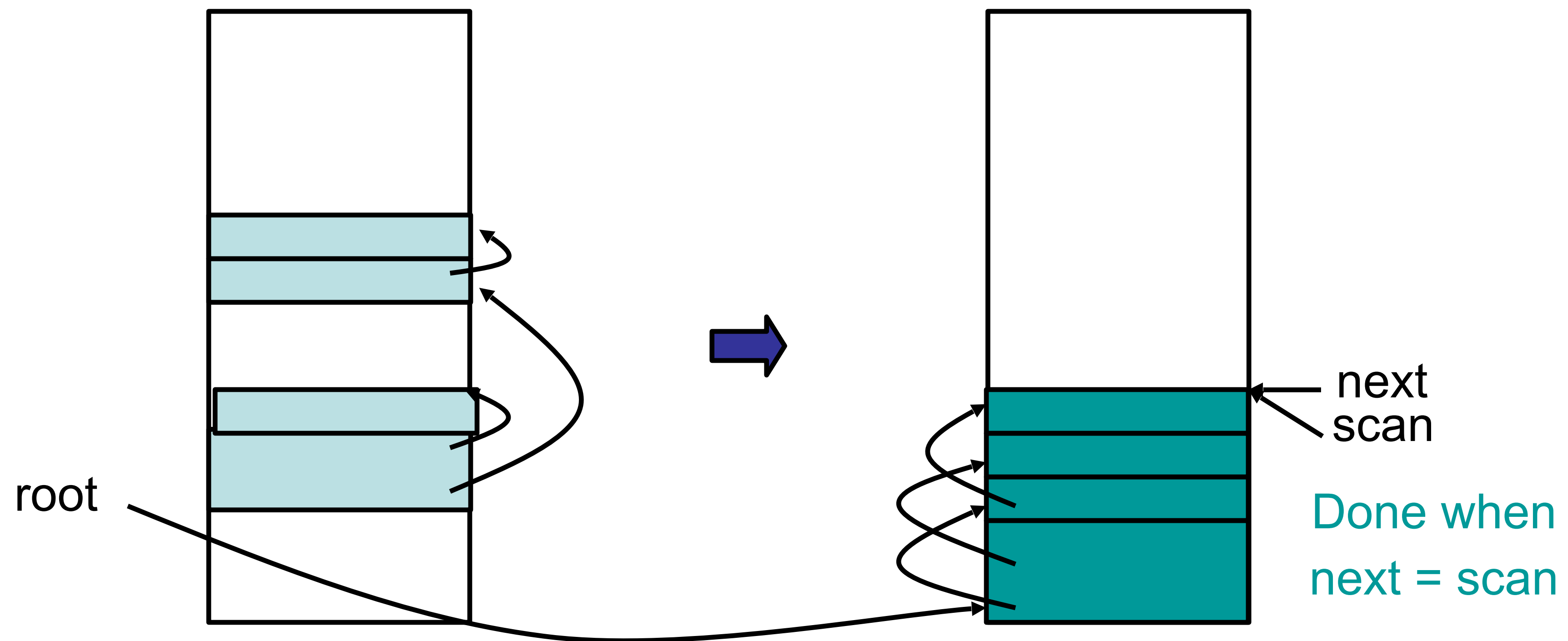
- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space





# Copying GC

- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



# Copying GC

- Pros
  - Simple & collects cycles
  - Run-time proportional to # live objects
  - Automatic compaction eliminates fragmentation
  - Fast allocation: pointer increment by object size
- Cons
  - Precise type information required (pointer or not)
    - Tag bits take extra space; normally use header word
  - Twice as much memory used as program requires
    - Usually, we anticipate live data will only be a small fragment of store
    - Allocate until 70% full
    - From-space = 70% heap; to-space = 30%
  - Long GC pauses = bad for interactive, real-time apps



# Generational GC

- **Empirical observation:** if an object has been reachable for a long time, it is likely to remain so
- **Empirical observation:** in many languages (especially functional languages), most objects died young
- **Conclusion:** we save work by scanning the young objects frequently and the old objects infrequently

# Generational GC

- Assign objects to different generations G0, G1, ...
  - G0 contains young objects, most likely to be garbage
  - G0 scanned more often than G1
  - Common case is two generations (new, tenured)
  - Roots for GC of G0 include all objects in G1 in addition to stack, registers

# Generational GC

- How do we avoid scanning tenured objects?
  - Observation: old objects rarely point to new objects
    - Normally, object is created and when it initialized it will point to older objects, not newer ones
    - Only happens if old object modified well after it is created
    - In functional languages that use mutation infrequently, pointers from old to new are very uncommon
  - Compiler inserts extra code on object field pointer write to catch modifications to old objects
  - Remembered set is used to keep track of objects that point into younger generation. Remembered set included in set of roots for scanning.



# Generational GC

- Other issues
  - When do we **promote** objects from young generation to old generation
    - Usually after an object survives a collection, it will be promoted
  - How big should the generations be?
    - Appel says each should be exponentially larger than the last
  - When do we collect the old generation?
    - After several **minor collections**, we do a **major collection**

# Generational GC

- Other issues
  - Sometimes different GC algorithms are used for the new and older generations.
    - Why? Because they have different characteristics
  - Copying collection for the new
    - Less than 10% of the new data is usually live
    - Copying collection cost is proportional to the live data
  - Mark-sweep for the old

# Conservative Collection

- Even languages like C can benefit from GC
  - Boehm-Weiser-Demers **conservative GC** uses heuristics to determine which objects are pointers and which are integers without any language support
    - last 2 bits are non-zero => can't be a pointer
    - integer is not in allocated heap range => can't be a pointer
    - mark phase traverses all possible pointers
    - conservative because it may retain data that isn't reachable
      - thinks an integer is actually a pointer
      - since it does not copy objects (thereby changing pointer values), mistaking integers for pointers does not hurt
    - all gc is conservative anyway so this is almost never an issue (despite what people say)
    - sound if your program doesn't manufacture pointers from integers by, say, using xor (using normal pointer arithmetic is fine)



# Compiler Interface

- The interface to the garbage collector involves two main parts
  - allocation code
    - languages can allocate up to approx 1 word/7 instructions
    - allocation code must be **blazingly** fast!
    - should be inlined and optimized to avoid call-return overhead
  - gc code
    - to call gc code, the program must identify the roots
    - to traverse data, heap layout must be specified somehow

# Allocation Code

Assume size of record allocated is N:

1. Call alloc code
2. Test  $\text{next} + N < \text{limit}$  (call gc on failure)
3. Move next into function result
4. Clear  $M[\text{next}], \dots, M[\text{next} + N - 1]$
5.  $\text{next} = \text{next} + N$
6. Return from alloc code



# Allocation Code

Assume size of record allocated is N:

1. Call alloc function
2. Test  $\text{next} + N < \text{limit}$  (call gc on failure)
3. Move next into function result
4. Clear  $M[\text{next}], \dots, M[\text{next} + N - 1]$
5.  $\text{next} = \text{next} + N$
6. Return from alloc function
7. Move result into computationally useful place
8. Store useful values into  $M[\text{next}], \dots, M[\text{next} + N - 1]$

useful computation  
not alloc overhead

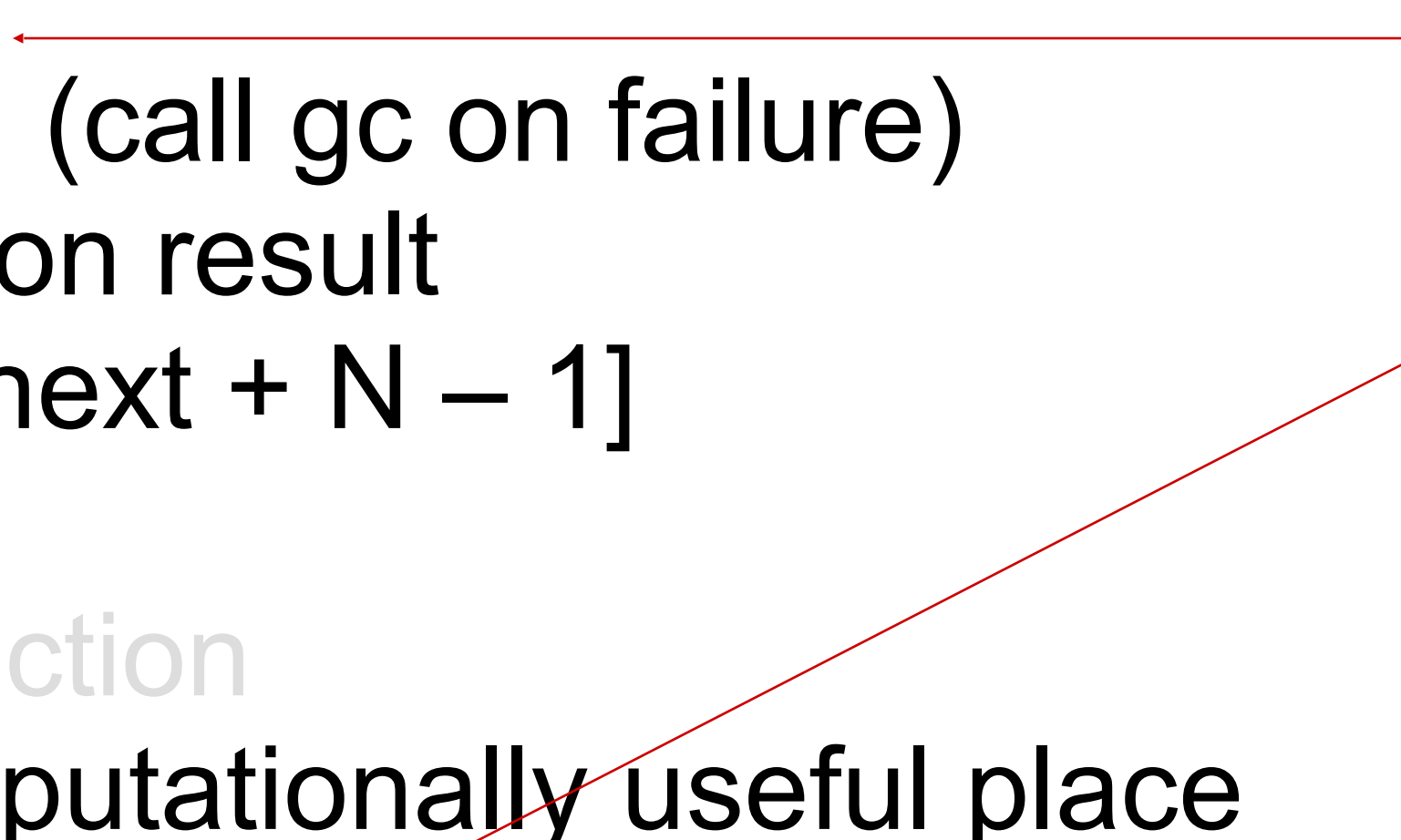


# Allocation Code

Assume size of record allocated is N:

1. Call alloc function
2. Test  $\text{next} + N < \text{limit}$  (call gc on failure)
3. Move next into function result
4. Clear  $M[\text{next}], \dots, M[\text{next} + N - 1]$
5.  $\text{next} = \text{next} + N$
6. Return from alloc function
7. Move result into computationally useful place
8. Store useful values into  $M[\text{next}], \dots, M[\text{next} + N - 1]$

inline  
alloc  
code

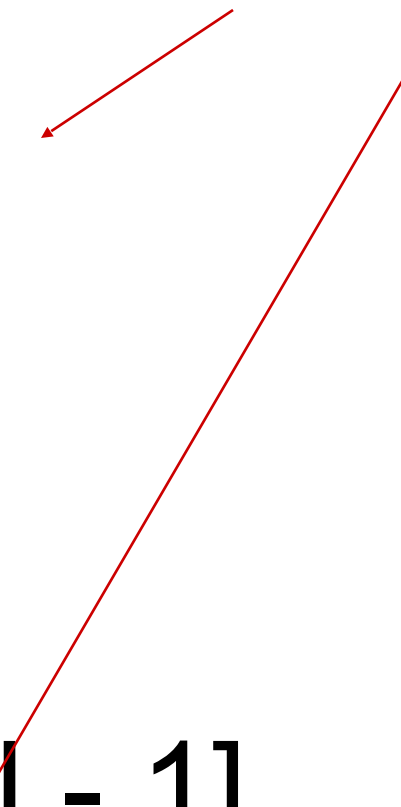


# Allocation Code

Assume size of record allocated is  $N$ :

1. Call alloc function
2. Test  $\text{next} + N < \text{limit}$  (call gc on failure)
3. Move next **into computationally useful place**
4. Clear  $M[\text{next}], \dots, M[\text{next} + N - 1]$
5.  $\text{next} = \text{next} + N$
6. Return from alloc function
7. Move next into computationally useful place
8. Store useful values into  $M[\text{next}], \dots, M[\text{next} + N - 1]$

combine  
moves



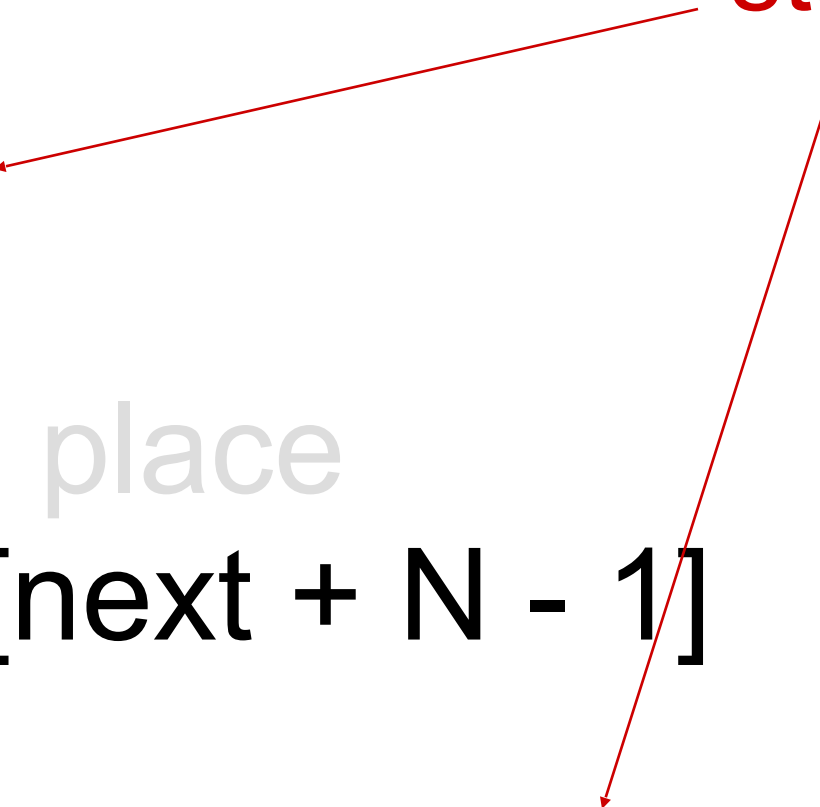


# Allocation Code

Assume size of record allocated is N:

1. Call alloc function
2. Test  $\text{next} + N < \text{limit}$  (call gc on failure)
3. Move next into computationally useful place
4. Clear  $M[\text{next}], \dots, M[\text{next} + N - 1]$
5.  $\text{next} = \text{next} + N$
6. Return from alloc function
7. Move next into computationally useful place
8. Store useful values into  $M[\text{next}], \dots, M[\text{next} + N - 1]$

eliminate  
useless  
store



# Allocation Code

Assume size of record allocated is  $N$ :

1. Call alloc function
2. Test  $\text{next} + N < \text{limit}$  (call gc on failure)
3. Move next into computationally useful place
4. Clear  $M[\text{next}], \dots, M[\text{next} + N - 1]$
5.  $\text{next} = \text{next} + N$
6. Return from alloc function
7. Move next into computationally useful place
8. Store useful values into  $M[\text{next}], \dots, M[\text{next} + N - 1]$

total overhead for allocation on the order of 3 instructions/alloc

# Calling GC code

- To call the GC, program must:
  - identify the **roots**:
    - a **GC-point**, is an control-flow point where the garbage collector may be called
      - allocation point; function call
    - for any GC-point, compiler generates a **pointer map** that says which registers, stack locations in the current frame contain pointers
    - a global table maps GC-points (code addresses) to pointer maps
    - when program calls the GC, to find all roots:
      - GC scans down stack, one activation record at a time, looking up the current pointer map for that record



# Calling GC code

- To call the GC, program must:
  - enable GC to determine **data layout** of all objects in the heap
    - every record has a header with size and pointer info
  - in object oriented languages like Java:
    - each object has an extra field that points to class definition
    - gc uses class definition to determine object layout including size and pointer info

# Summary

- Garbage collectors are a complex and fascinating part of any modern language implementation
- Different collection algs have pros/cons
  - explicit MM, reference counting, copying, generational, mark-sweep
  - all methods, including explicit MM have costs
  - optimizations make allocation fast, GC time, space and latency requirements acceptable
  - additional reading: Appel Chapter 13



# Program Analysis and Optimization

# Liveness, Conflicts and Register Allocation

# Where Were We?

We've discussed so far how to compile many features **correctly** (functional correctness) only worrying about preserving **asymptotic complexity**.

But our generated code has high **constant factors** in its complexity:

- Use stack-allocation for all local variables.
- Many redundant dynamic type checks
- Simple arithmetic is preserved even if we can evaluate it at compile time

...



# Where Were We?

We've discussed so far how to compile many features **correctly** (functional correctness) only worrying about preserving **asymptotic complexity**.

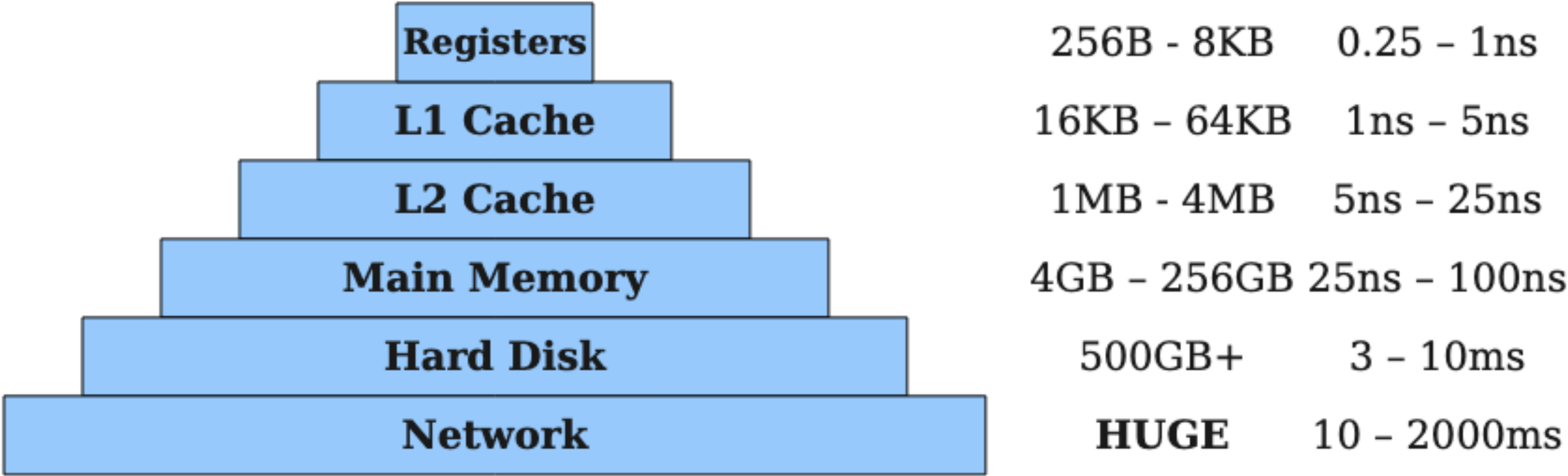
But our generated code has high **constant factors** in its complexity:

- **Use stack-allocation for all local variables.**
- Many redundant dynamic type checks
- Simple arithmetic is preserved even if we can evaluate it at compile time

...

# Memory Hierarchy

Systems  
view of  
memory:



Snake/SSA  
view of  
memory

variables, heap allocated objects

# Memory Allocation for Locals

For code generation, we need to map our SSA local variables to memory locations where they are stored.

Current strategy:

- Allocate all variables onto the stack, based on nesting of the current scope.

- Not completely naive: we do re-use some stack space in nested sub-blocks

Big Performance hit: need to move values in and out of registers frequently.



# Register Allocation

For code generation, we need to map our SSA local variables to memory locations where they are stored.

Goal:

- Store variable's values in registers whenever possible.
- Only use stack space if we run out of registers.

Performance gains:

- 3-10x+ faster variable accesses (by far the **most important** optimization for a compiler)
- Space gain: smaller stack frames

High computational complexity: often the slowest part of the compiler

# Register Allocation Examples



# Register Allocation Examples

$f(a) :$

$x = a * 2$

$y = x + 7$

ret  $y$

# Register Allocation Examples

`f(a):`

`x = a * 2`

`y = x + 7`

`ret y`

Currently:

`a: stack [rsp - 8]`

`x: stack [rsp - 16]`

`y: stack [rsp - 32]`

# Register Allocation Examples

`f(a):`

`x = a * 2`

`y = x + 7`

`ret y`

`With register alloc:`

`a: rax`

`x: rax`

`y: rax`



# Register Allocation Examples

`f(a):`

`x = a * 2`

`y = x + 7`

`ret y`

`With register alloc:`

`a: rax`

`x: rax`

`y: rax`

`... assertInt`

`sar rax, 1`

`imul rax, 4`

`... assertInt`

`add rax, 14`

`ret`

# Register Allocation Examples

`f(a):`

`x = a * 2`

`y = x + 7`

`z = x * y`

`ret z`

With register alloc:

`a: rax`

`x: rax`

`y: rcx`

`z: rax`

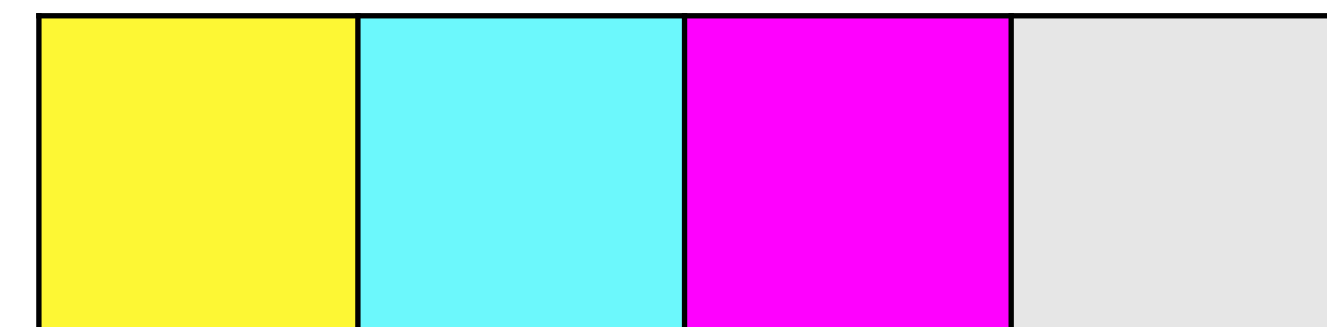
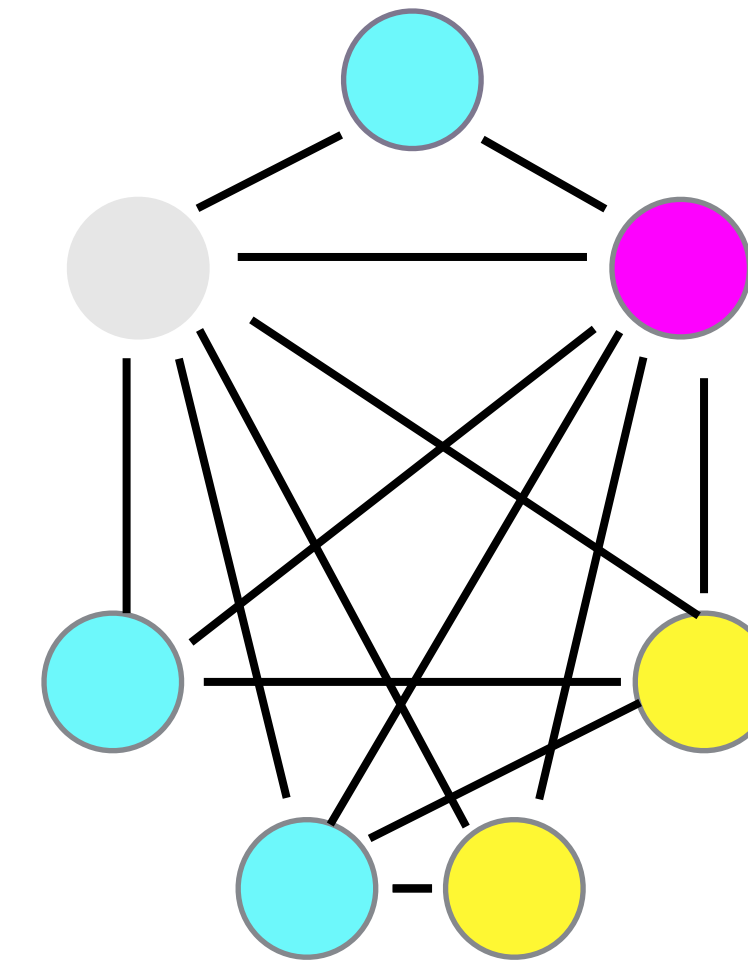
Can't put x and y in the same register  
Say they are **in conflict** or **interfering**

# Register Allocation: Graph Coloring Approach

The best register allocation algorithms (in terms of quality of output, not efficiency) use **graph coloring**

Graph coloring problem:

Given a graph  $(V, E)$  and set of colors  $K$  assign each vertex a color so that adjacent vertices all have **different** colors.





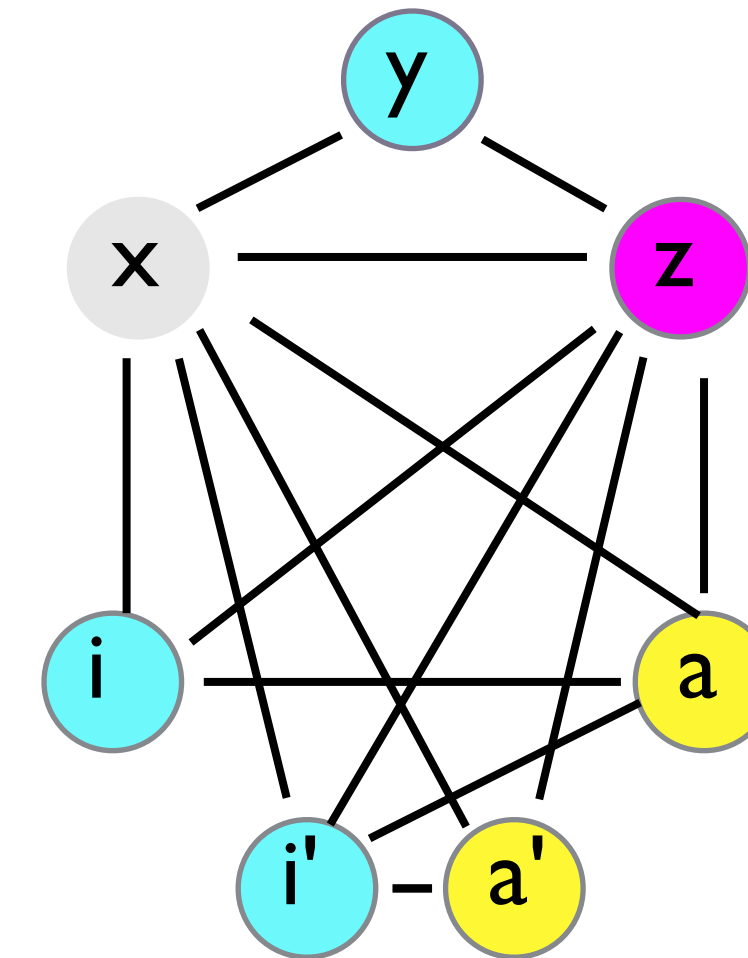
# Register Allocation: Graph Coloring Approach

Graph Coloring register allocation:

Make an **interference graph**:  
vertices are variables, edges are **interference** relationships

Colors are the different registers

A solution is a valid register assignment



# Register Allocation Overview

Break down into three tasks:

1. **Liveness analysis:** determine which values are needed at every program point
2. **Conflict analysis:** use liveness info to construct interference graph
3. **Graph coloring:** attempt to color the interference graph, **spilling** variables onto the stack if no solution can be found

More complicated in practice because registers are not all treated the same (argument/return registers, caller/callee-save, shift instructions)



# Liveness Analysis

One of the most fundamental analyses a compiler performs is **liveness analysis**.

Used to determine at each program point which variables are **live**, i.e., which variables' values need to be available at runtime.

Example uses:

- Lambda lifting: the arguments that **need** to be added in lambda lifting are exactly the **live** ones
- Register allocation: only need to store all of the **live** variables in registers/ the stack. Means we can re-use space when a variable is no longer live.
- Function calls: only need to save the values of caller-save registers if the value is **live**

# Liveness Analysis

Semantic definition:

a variable **x** is live in a block **b** (or expression, operation, etc) if the observable behavior of **b** depends on the value of **x**.

Can be done for ASTs or SSA blocks

- ASTs: determining what values are captured for lambda lifting/closure conversion
- SSA: determining interference for register allocation

# Liveness Examples

---

is  $X$  live?



# Liveness Examples

---

is X live?

x

Yes

# Liveness Example

---

is X live?

$x * y$

Yes

# Liveness Example

---

is X live?

if b: x else: y

if b is ever  
true, yes  
otherwise no



# Liveness Example

---

is X live?

```
let b = true in  
if b: x else: y
```

**yes**

# Liveness Example

---

is X live?

```
let b = false in  
if b: x else: y
```

no



# Liveness Example

---

is X live?

```
let b = read_input() in  
if b: x else: y
```

**yes**

# Liveness Example

---

is X live?

```
let b = complex_fn() in  
if b: x else: y
```

if complex\_fn ever  
returns true: yes,  
otherwise: no

# Limitation: Computability

---

Determining correct liveness information requires determining the possible values produced by arbitrary functions...

## Rice's Theorem:

Any non-trivial semantic property of programs in a Turing-complete language is undecidable

Determining liveness of variables is undecidable!



# Limitation: Computability

---

Determining **correct** liveness information can be arbitrarily complicated...

What if we determined **incorrect** liveness information sometimes?

- false positives: sometimes we say a variable is live when it's not
- false negatives: sometimes we say a variable is not live when it is

False positives are ok: we will just use more registers/space than necessary

# Limitation: Computability

---

## Goal: **Overapproximate**

The output of our liveness analysis should include every variable that is live, but possibly some that are not live.

Approach so far in class: Use scope as our liveness analysis

- This is an overapproximation: a variable can't be live if it's not in scope

We can do much better

- Only consider variables live if they actually get used
- But consider all execution paths (i.e. branches) to be possible

# Liveness Analysis: Specification

---

Define a function  $\text{LIVE} : \text{Expression} \rightarrow \text{Set}(\text{Variable})$

- $\text{LIVE}(x) = \{ x \}$
- $\text{LIVE}(n) = \{ \}$
- $\text{LIVE}(\text{Prim}(\text{op}, [\text{imm1}, \dots])) = \text{LIVE}(\text{imm1}) \cup \dots$
- $\text{LIVE}(\text{call}(f; [\text{imm1}, \dots])) = \text{LIVE}(\text{imm1}) \cup \dots$
- $\text{LIVE}(x = \text{op in } b) = (\text{LIVE}(b) - x) \cup \text{LIVE}(\text{op})$
- $\text{LIVE}(\text{br } f(\text{imm}, \dots)) = (\text{LIVE}(f.\text{body}) - f.\text{args}) \cup \text{LIVE}(\text{imm1}) \cup \dots$
- $\text{LIVE}(\text{cbr } \text{imm}: f \text{ else: } g) = \text{LIVE}(\text{imm}) \cup \text{LIVE}(f.\text{body}) \cup \text{LIVE}(g.\text{body})$
- $\text{LIVE}(\text{ret } \text{imm}) = \text{LIVE}(\text{imm})$



# Liveness Analysis: Specification

---

Our definition of liveness is **recursive** because our blocks are recursive:

- if a block  $f$  includes a branch to itself, the live variables in  $f$  will depend on the live variables in  $f$ ...
- if multiple blocks mutually recursively branch to each other, we have the same issue.

This means our specification is not fully precise.

Solution: we want the **minimal** solution to our recursive equations.

To implement this, we **initialize** all blocks to have 0 live variables, and **iteratively** improve this information, using the previous round's information each time.

An example of a general process called **dataflow analysis**, more on this later

# Liveness Analysis: Example

---

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      B a' = a + x  
        br loop(i', a')  
      b = i == 0  
      cbr b thn() els()  
  br loop(y, 0)
```

In the sub-expression **B**, which variables are

In scope:

Syntactically occurring:

Live:



# Liveness Analysis: Example

---

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      B a' = a + x  
        br loop(i', a')  
      b = i == 0  
      cbr b thn() els()  
  br loop(y, 0)
```

In the sub-expression **B**, which variables are

In scope:  $x, y, z, i, a, i'$

Syntactically occurring:  $x, a, a', i'$

Live:  $x, z, a, i'$

# Liveness Analysis: Example

---

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      B a' = a + x  
        br loop(i', a')  
      b = i == 0  
      cbr b thn() els()  
  br loop(y, 0)
```

In the sub-expression **B**, which variables are

In scope:  $x, y, z, i, a, i'$

Syntactically occurring:  $x, a, a', i'$

Live:  $x, z, a, i'$

# Liveness Analysis: Example

---

```
f(x,y,z):  
  1 loop(i,a):  
    5 thn():  
      6 r = a * z  
      7 ret r  
    8 els():  
      9 i' = i - 1  
     10 a' = a + x  
     11 br loop(i', a')  
  3 b = i == 0  
  4 cbr b thn() els()  
  2 br loop(y, 0)
```

Round 0

|      |     |
|------|-----|
| 1:   | { } |
| 2:   | { } |
| 3:   | { } |
| 4:   | { } |
| 5/6: | { } |
| 7:   | { } |
| 8/9: | { } |
| 10:  | { } |
| 11:  | { } |

Round 1

|      |   |
|------|---|
| 1:   | ? |
| 2:   | ? |
| 3:   | ? |
| 4:   | ? |
| 5/6: | ? |
| 7:   | ? |
| 8/9: | ? |
| 10:  | ? |
| 11:  | ? |



# Liveness Analysis: Example

---

```
f(x,y,z):  
  1 loop(i,a):  
    5 thn():  
      6 r = a * z  
      7 ret r  
    8 els():  
      9 i' = i - 1  
     10 a' = a + x  
     11 br loop(i', a')  
  3 b = i == 0  
  4 cbr b thn() els()  
  2 br loop(y, 0)
```

## Round 0

```
1:  { }  
2:  { }  
3:  { }  
4:  { }  
5/6: { }  
7:  { }  
8/9: { }  
10: { }  
11: { }
```

## Round 1

```
1:  {x, z}  
2:  {y}  
3:  {a, i, x, z}  
4:  {a, b, i, x, z}  
5/6: {a, z}  
7:  {r}  
8/9: {a, i, x}  
10: {a, i', x}  
11: {a', i'}
```

# Liveness Analysis: Example

---

```
f(x,y,z):  
  1 loop(i,a):  
    5 thn():  
      6 r = a * z  
      7 ret r  
    8 els():  
      9 i' = i - 1  
     10 a' = a + x  
     11 br loop(i', a')  
  3 b = i == 0  
  4 cbr b thn() els()  
  2 br loop(y, 0)
```

Round 1

1: {x, z}  
2: {y}  
3: {a, i, x, z}  
4: {a, b, i, x, z}  
5/6: {a, z}  
7: {r}  
8/9: {a, i, x}  
10: {a, i', x}  
11: {a', i'}

Round 2

1: ?  
2: ?  
3: ?  
4: ?  
5/6: ?  
7: ?  
8/9: ?  
10: ?  
11: ?

# Liveness Analysis: Example

---

## Round 1

## Round 2

```
f(x,y,z):  
  1 loop(i,a):  
    5 thn():  
      6 r = a * z  
      7 ret r  
    8 els():  
      9 i' = i - 1  
     10 a' = a + x  
     11 br loop(i', a')  
  3 b = i == 0  
  4 cbr b thn() els()  
  2 br loop(y, 0)
```

1: {x, z}

2: {y}

3: {a, i, x, z}

4: {a, b, i, x, z}

5/6: {a, z}

7: {r}

8/9: {a, i, x}

10: {a, i', x}

11: {a', i'}

1: {x, z}

2: {x, y, z}

3: {a, i, x, z}

4: {a, b, i, x, z}

5/6: {a, z}

7: {r}

8/9: {a, i, x, z}

10: {a, i', x, z}

11: {a', i', x, z}



# Liveness Analysis: Example

---

```
f(x,y,z):  
  1 loop(i,a):  
    5 thn():  
      6 r = a * z  
      7 ret r  
    8 els():  
      9 i' = i - 1  
     10 a' = a + x  
     11 br loop(i', a')  
  3 b = i == 0  
  4 cbr b thn() els()  
  2 br loop(y, 0)
```

## Round 2

```
1:  {x, z}  
2:  {x, y, z}  
3:  {a, i, x, z}  
4:  {a, b, i, x, z}  
5/6: {a, z}  
7:  {r}  
8/9: {a, i, x, z}  
10: {a, i', x, z}  
11: {a', i', x, z}
```

## Round 3

```
1:  ?  
2:  ?  
3:  ?  
4:  ?  
5/6: ?  
7:  ?  
8/9: ?  
10: ?  
11: ?
```

# Liveness Analysis: Example

---

## Round 2

## Round 3

```
f(x,y,z):  
  1 loop(i,a):  
    5 thn():  
      6 r = a * z  
      7 ret r  
    8 els():  
      9 i' = i - 1  
     10 a' = a + x  
     11 br loop(i', a')  
  3 b = i == 0  
  4 cbr b thn() els()  
  2 br loop(y, 0)
```

1: {x, z}  
2: {x, y, z}  
3: {a, i, x, z}  
4: {a, b, i, x, z}  
5/6: {a, z}  
7: {r}  
8/9: {a, i, x, z}  
10: {a, i', x, z}  
11: {a', i', x, z}

1: {x, z}  
2: {x, y, z}  
3: {a, i, x, z}  
4: {a, b, i, x, z}  
5/6: {a, z}  
7: {r}  
8/9: {a, i, x, z}  
10: {a, i', x, z}  
11: {a', i', x, z}



# Liveness Analysis: Example

---

```
f(x,y,z):  
  1 loop(i,a):  
    5 thn():  
      6 r = a * z  
      7 ret r  
    8 els():  
      9 i' = i - 1  
      B 10 a' = a + x  
        11 br loop(i', a')  
    3 b = i == 0  
    4 cbr b thn() els()  
  2 br loop(y, 0)
```

|      |                 |
|------|-----------------|
| 1:   | {x, z}          |
| 2:   | {x, y, z}       |
| 3:   | {a, i, x, z}    |
| 4:   | {a, b, i, x, z} |
| 5/6: | {a, z}          |
| 7:   | {r}             |
| 8/9: | {a, i, x, z}    |
| 10:  | {a, i', x, z}   |
| 11:  | {a', i', x, z}  |

In the sub-expression **B**, which variables are

In scope: x, y, z, i, a, i'

Syntactically occurring: x, a, a', i'

Live: x, z, a, i'

# Implementation Concerns

---

How to store live sets?

- Add annotation metadata to the SSA AST
  - `init_liveness(e: BB<T>) -> BB<HashSet<String>>`
  - `update_liveness(e: BB<HashSet<String>>) -> BB<HashSet<String>>`
- iterate until you reach a fixed point
  - `update_liveness(b) == b`

# Conflict Analysis

---

Once we know when we need the value of each variable, we determine which variables cannot be assigned the same register

2 variables truly conflict when

- They are live at the same time
- with different values

Err on the side of *\*too many\** conflicts.



# Conflict Analysis

---

Simple approach:

- Initialize the graph with all variables in the program
- Whenever a variable is assigned to, add conflicts with all the live variables
- Also, parameters of a block all must mutually conflict, as they are assigned to simultaneously.

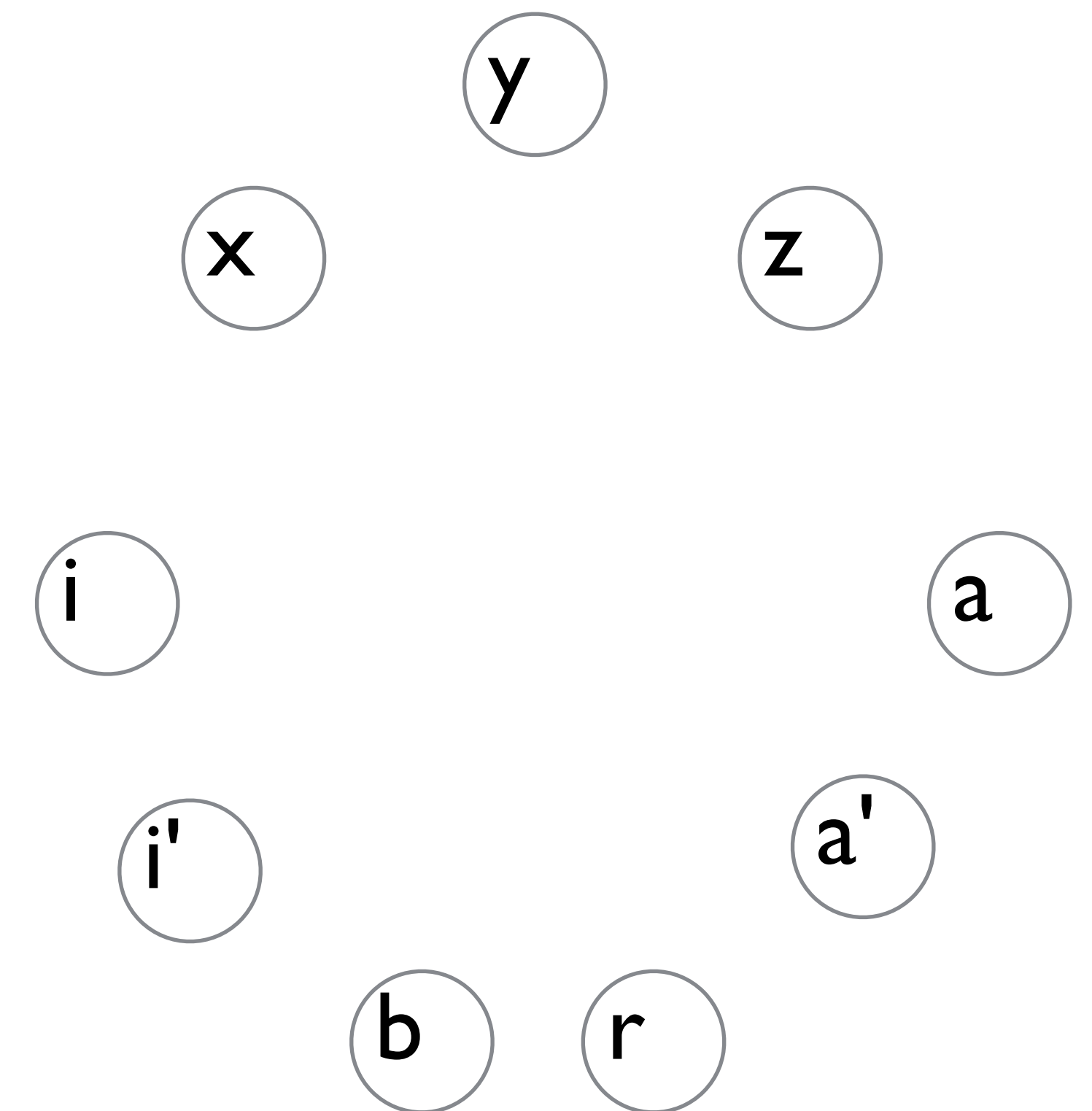
This is an overapproximation to true conflicts

# Conflict Analysis

---

```
f(x,y,z):  
  1 loop(i,a):  
    5 thn():  
      6 r = a * z  
      7 ret r  
    8 els():  
      9 i' = i - 1  
     10 a' = a + x  
     11 br loop(i', a')  
  3 b = i == 0  
  4 cbr b thn() els()  
  2 br loop(y, 0)
```

```
1:    {x, z}  
2:    {x, y, z}  
3:    {a, i, x, z}  
4:    {a, b, i, x, z}  
5/6: {a, z}  
7:    {r}  
8/9: {a, i, x, z}  
10:   {a, i', x, z}  
11:   {a', i', x, z}
```



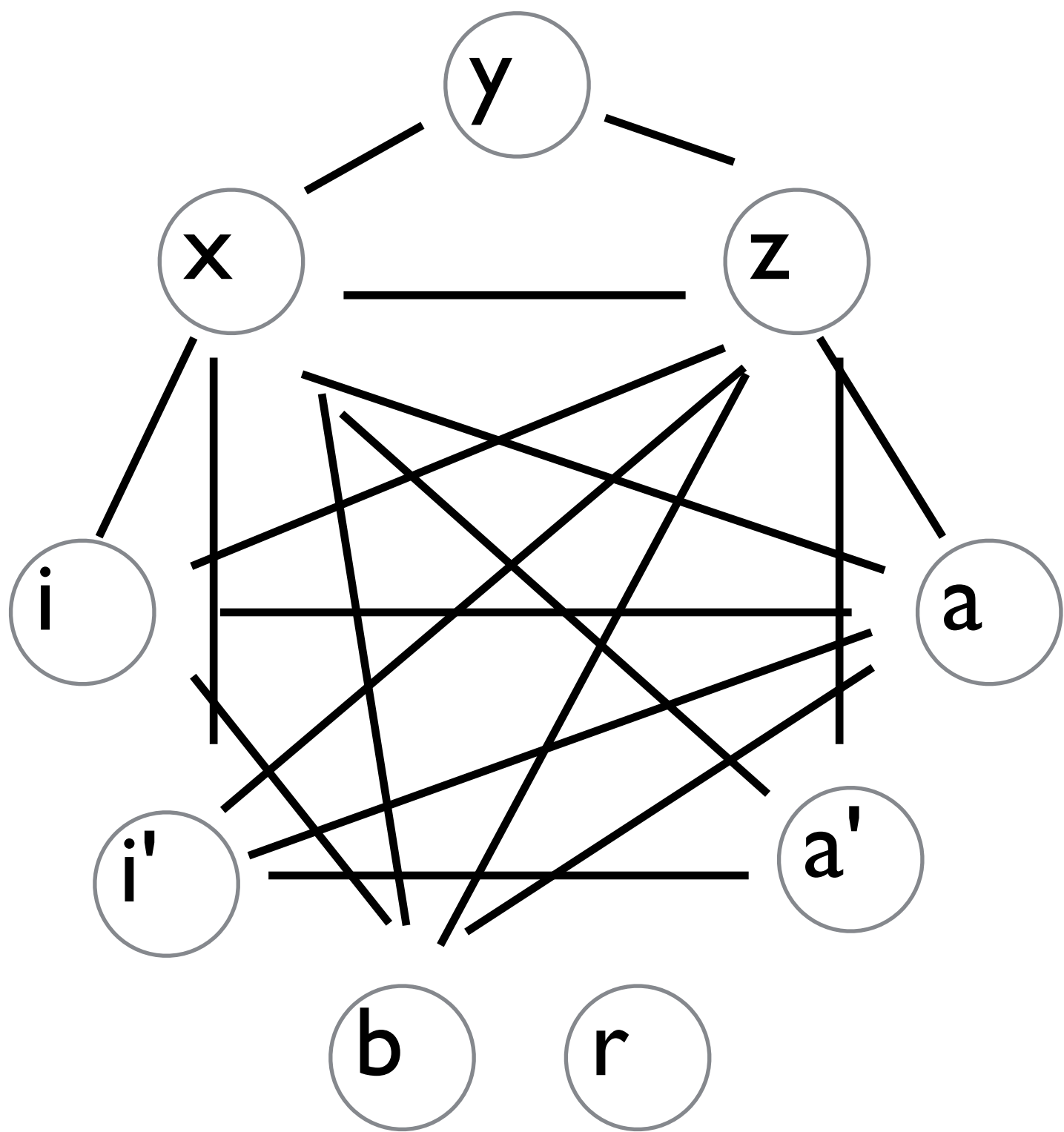
# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

# Liveness Info

- 1: {x, z}
- 2: {x, y, z}
- 3: {a, i, x, z}
- 4: {a, b, i, x, z}
- 5/6: {a, z}
- 7: {r}
- 8/9: {a, i, x, z}
- 10: {a, i', x, z}
- 11: {a', i', x, z}

# Interference Graph

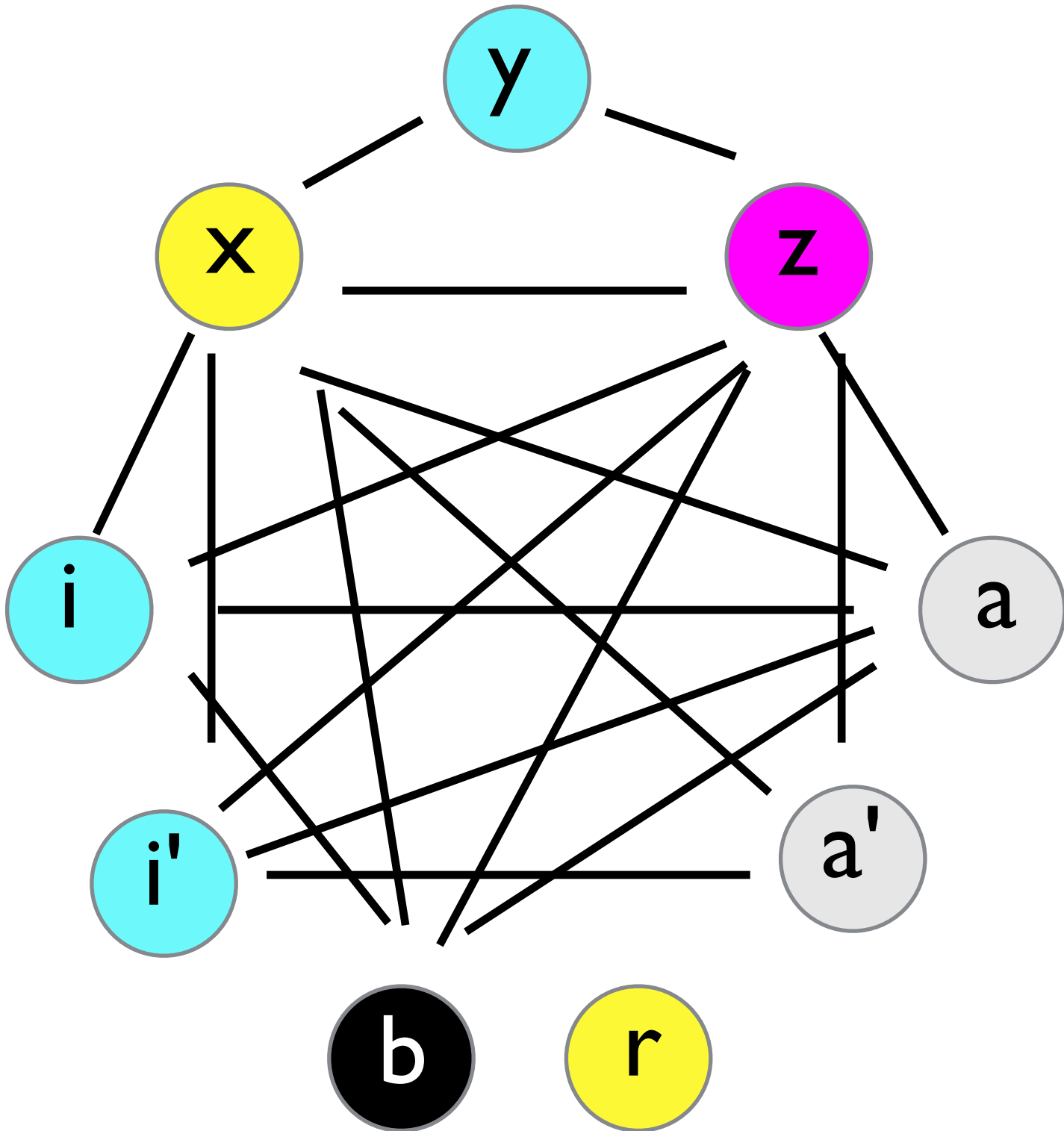




# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

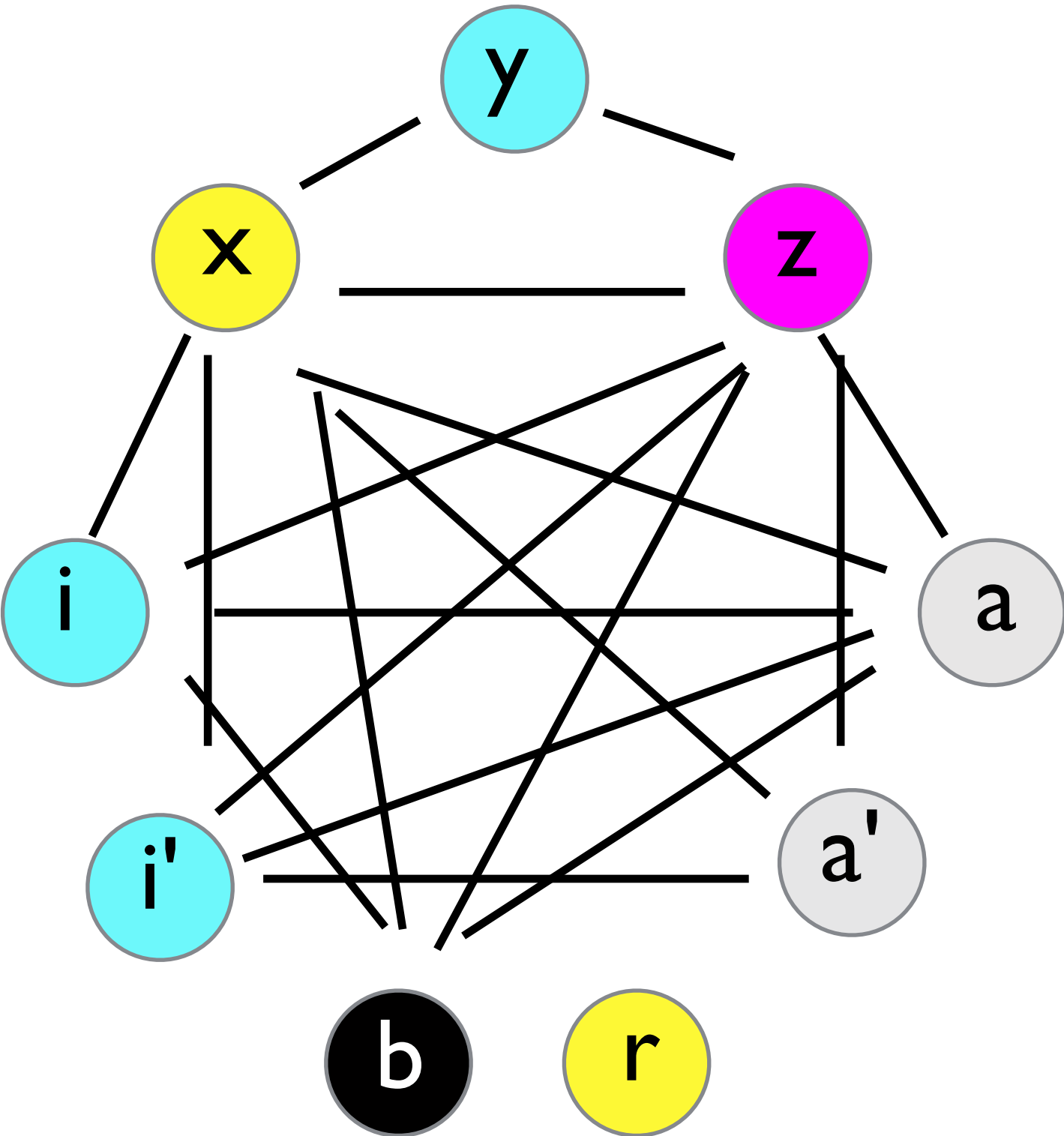
# Interference Graph



# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

# Interference Graph



|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| rax | rsi | rdi | rcx | rdx |
|-----|-----|-----|-----|-----|

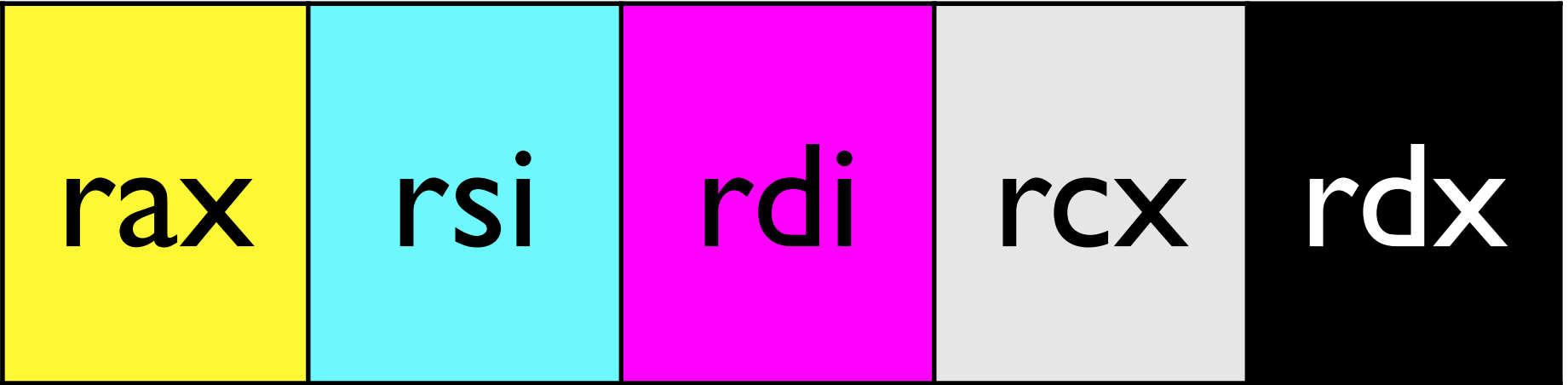
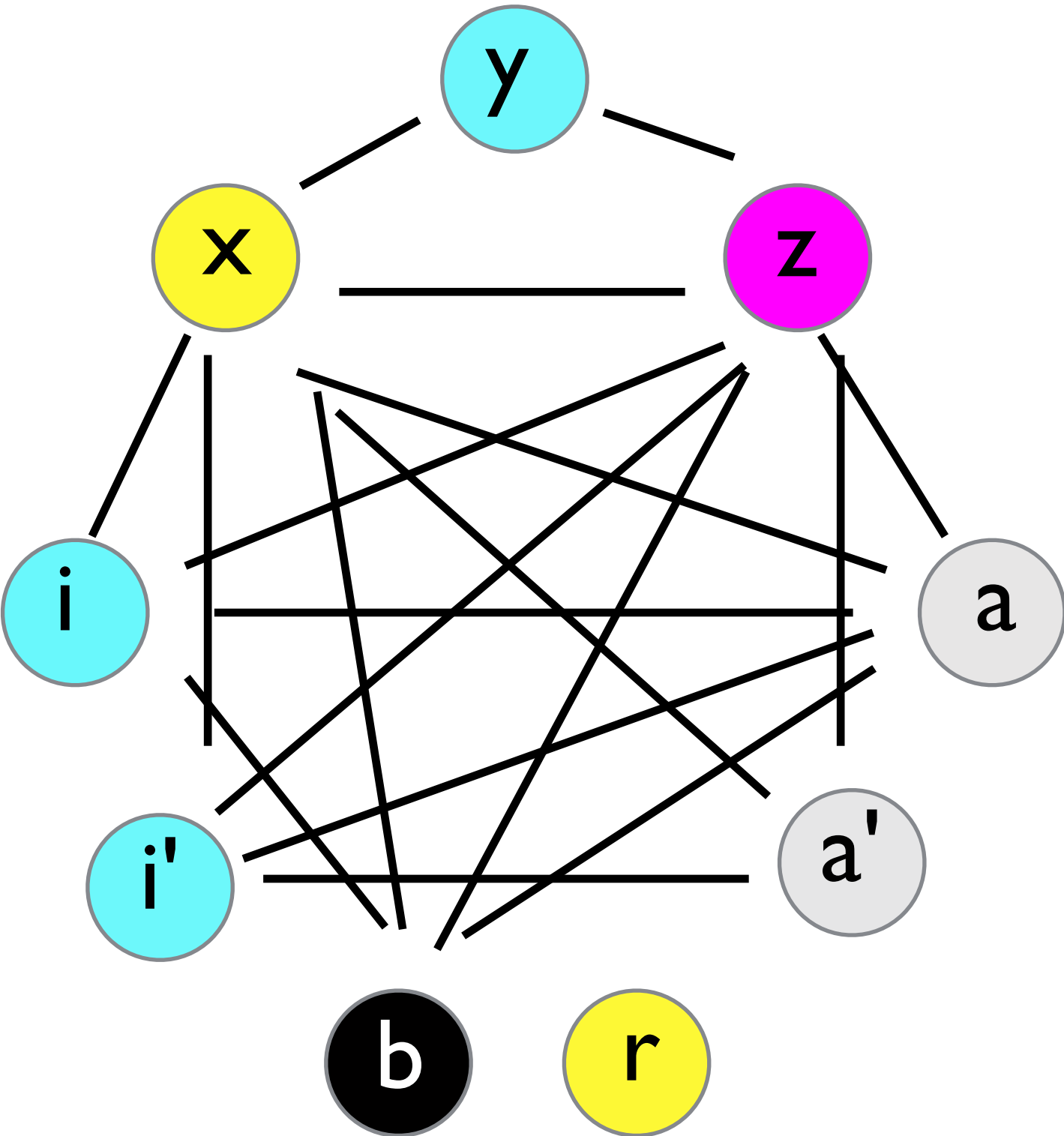


# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

```
f:  
  mov rcx, 0  
  jmp loop  
loop:  
  cmp rsi, 0  
  mov rdx, 0  
  sete rdx  
  cmp rdx 0  
  jne thn  
  jmp els  
thn:  
  mov rax, rcx  
  imul rax, rdi  
  ret  
els:  
  sub rsi, 1  
  add rcx, rax  
  jmp loop
```

Interference  
Graph



# Graph Coloring Register Allocation

---

Given our register interference graph, want to assign a register to each variable so that no interfering variables are assigned the same register.

Equivalent to graph coloring of the interference graph

- think of each register as a “color” and we want to paint each node so that no adjacent nodes are the same color.

Efficient algorithm for graph coloring -> efficient algorithm for register allocation!

# Graph Coloring is Hard

---

Determining whether a graph is  $k$ -colorable is NP-complete for  $k > 2$ .

- So no polytime algorithm is known

Does that mean register allocation is NP-hard?



# Is Register Allocation Hard?

---

Chaitin et al, "Register allocation via coloring", *Computer Languages* 1981

- Showed that the register allocation problem for a language with assignments and arbitrary control flow (goto) is **equivalent** to graph coloring
- every graph arises as the interference graph of some program

So register allocation of an imperative language with goto is NP complete.

- But our programs are more restrictive: SSA form...

# Chaitin's Algorithm

# Chaitin's Algorithm

---

Chaitin's algorithm is efficient ( $O(|V| + |E|)$ ), simple to implement



# Chaitin's Algorithm

---

Simple recursive algorithm for constructing a valid coloring.

Fix an ordering on the nodes of the graph.

- Base case: if the graph is empty, it is trivially colorable
- Recursive case:
  - Remove a node from the graph, and all of its edges
  - Recursively color the remaining graph
  - Add the node and its edges back to the graph, pick a color that is different from all of its neighbors. If the neighbors have exhausted all of the colors, "spill"

# Chaitin's Algorithm

---

Chaitin's algorithm is efficient ( $O(|V| + |E|)$ ), simple to implement

- How good the coloring is depends on the order we color the nodes to the graph
  - called the **elimination ordering**



# Chaitin's Algorithm

---

Chaitin's algorithm is efficient ( $O(|V| + |E|)$ ), simple to implement

- How good the coloring is depends on the order we color the nodes to the graph
  - called the **elimination ordering**
- Theorem: For every graph, there is an elimination ordering such that Chaitin's algorithm produces an optimal coloring
  - therefore finding this optimal elimination ordering for a general graph is NP-complete

# Graph Coloring SSA Programs

---

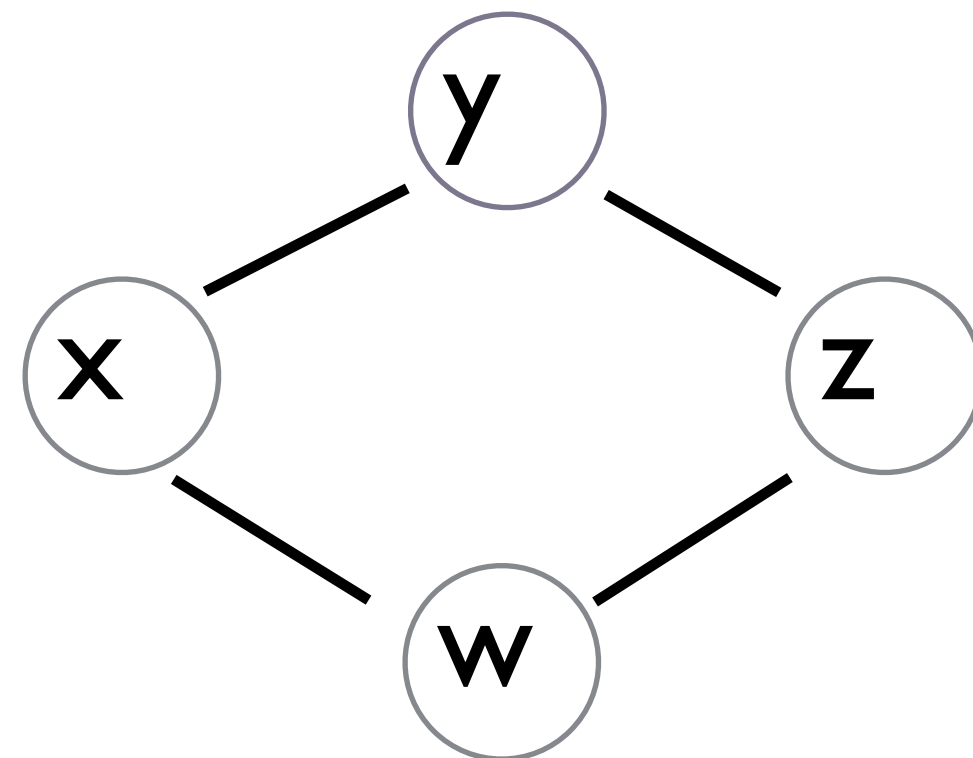
Hack et al, "Register Allocation for Programs in SSA-Form",  
*Compiler Construction* 2006

# Graph Coloring SSA Programs

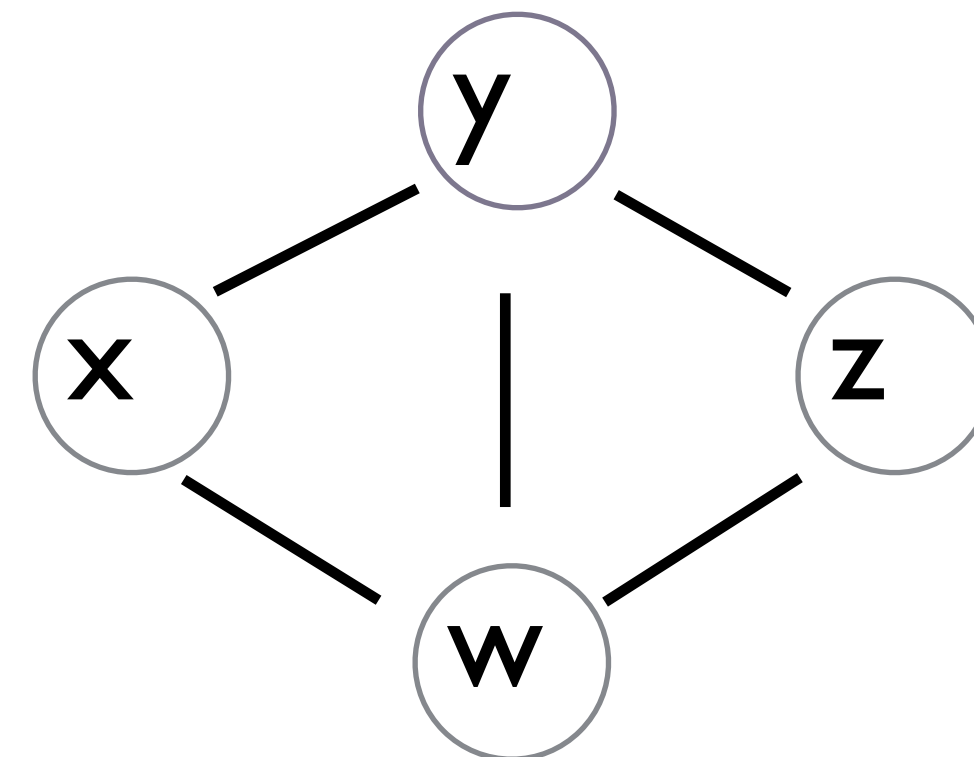
---

Hack et al, "Register Allocation for Programs in SSA-Form",  
*Compiler Construction 2006*

- The interference graphs of an SSA program are all **chordal**
  - Every cycle  $\geq 4$  nodes has a **chord**



Not chordal



chordal

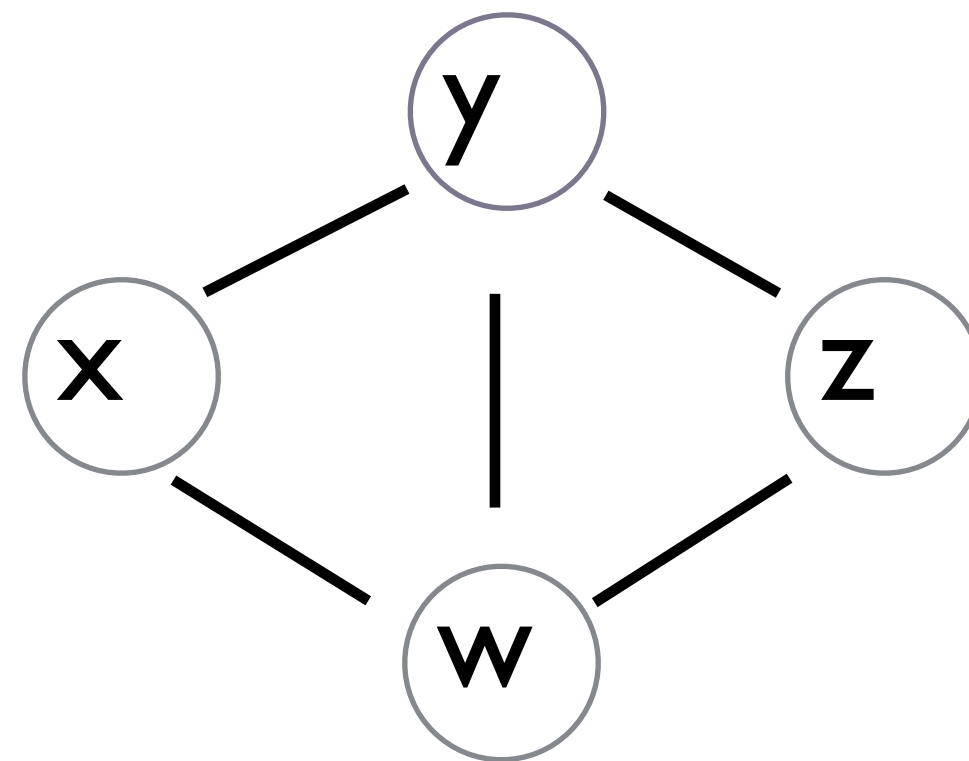


# Coloring Chordal Graphs

---

Theorem: A graph is chordal iff it has a **perfect elimination ordering** (PEO)

- a total ordering of nodes  $v_1, v_2, v_3, \dots$  such that for each  $v_i$ ,  $v_i$  forms a clique with all its neighbors earlier in the order
- Chaitin's algo produces an optimal coloring if we use a PEO



$x, y, z, w$   
not perfect:  $N(w)$  non-clique

$w, x, y, z$   
perfect

# Coloring Chordal Graphs

---

Theorem: A graph is chordal iff it has a **perfect elimination ordering (PEO)**

Theorem: A graph is chordal iff it is the intersection graph of a group of subtrees of a tree

- In an SSA program, each variable's liveness is a subtree of the AST
- The interference graph is exactly the intersection graph of those subtrees
- Therefore, the interference graph of an SSA program is chordal!

# SSA Interference Graphs are Chordal!

---

Every SSA Interference Graph is chordal

Chaitin's algorithm computes optimal coloring given a PEO

So we can color SSA interference graphs if we can find a PEO

SSA programs have a perfect elimination ordering that is easy to compute:

"in-scope" or "dominance" relation

a variable  $x$  dominates  $y$  if  $x$  in scope when  $y$  is defined  
(includes simultaneous binding)

- $x$ 's definition is "closer to the root" of the AST than  $y$
- easy to compute: pre-order traversal of the nodes

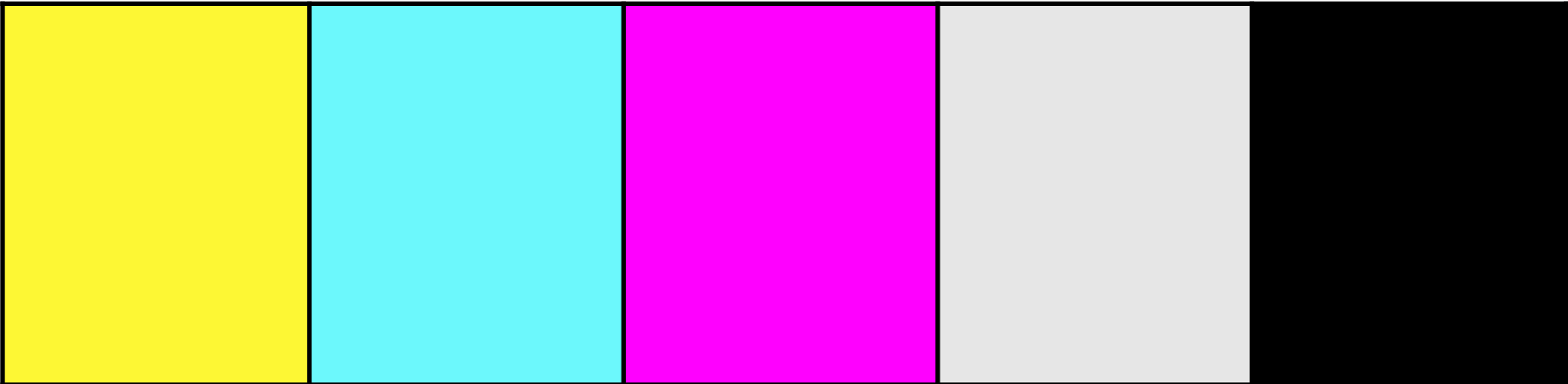
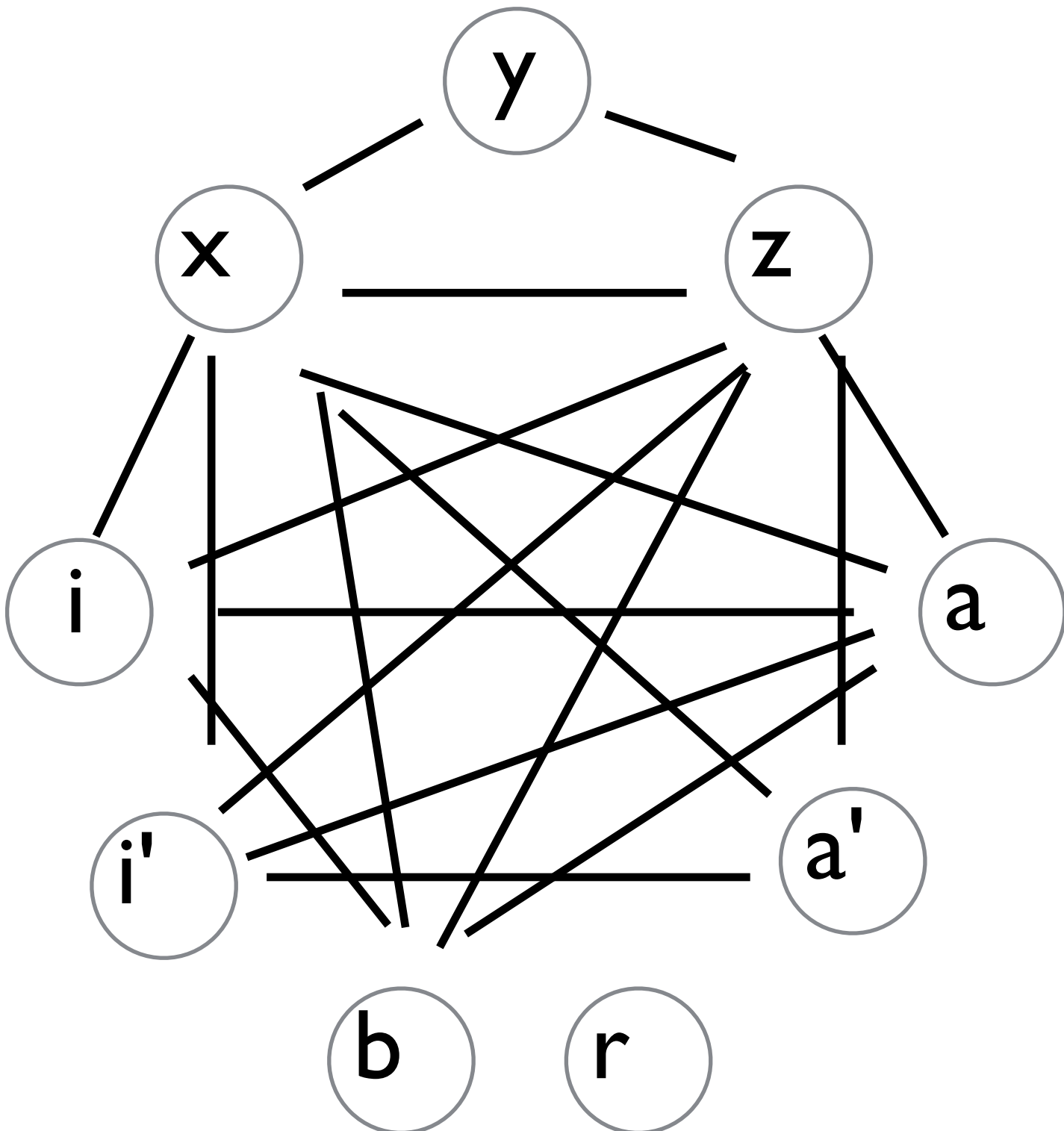


# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

Pick a PEO:  
  
variables should be colored  
after anything that was  
already in scope

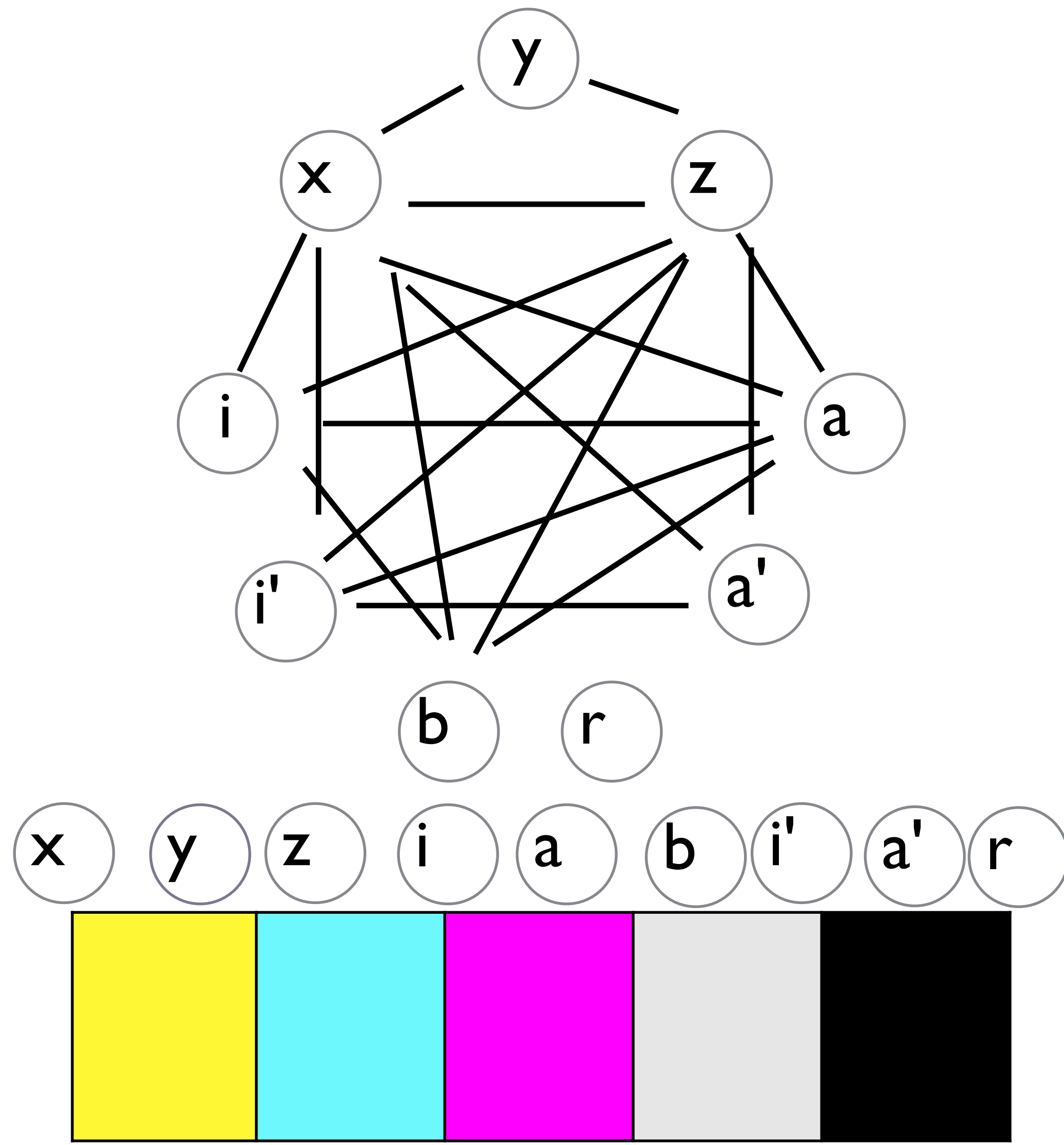
# Interference Graph



# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

# Interference Graph

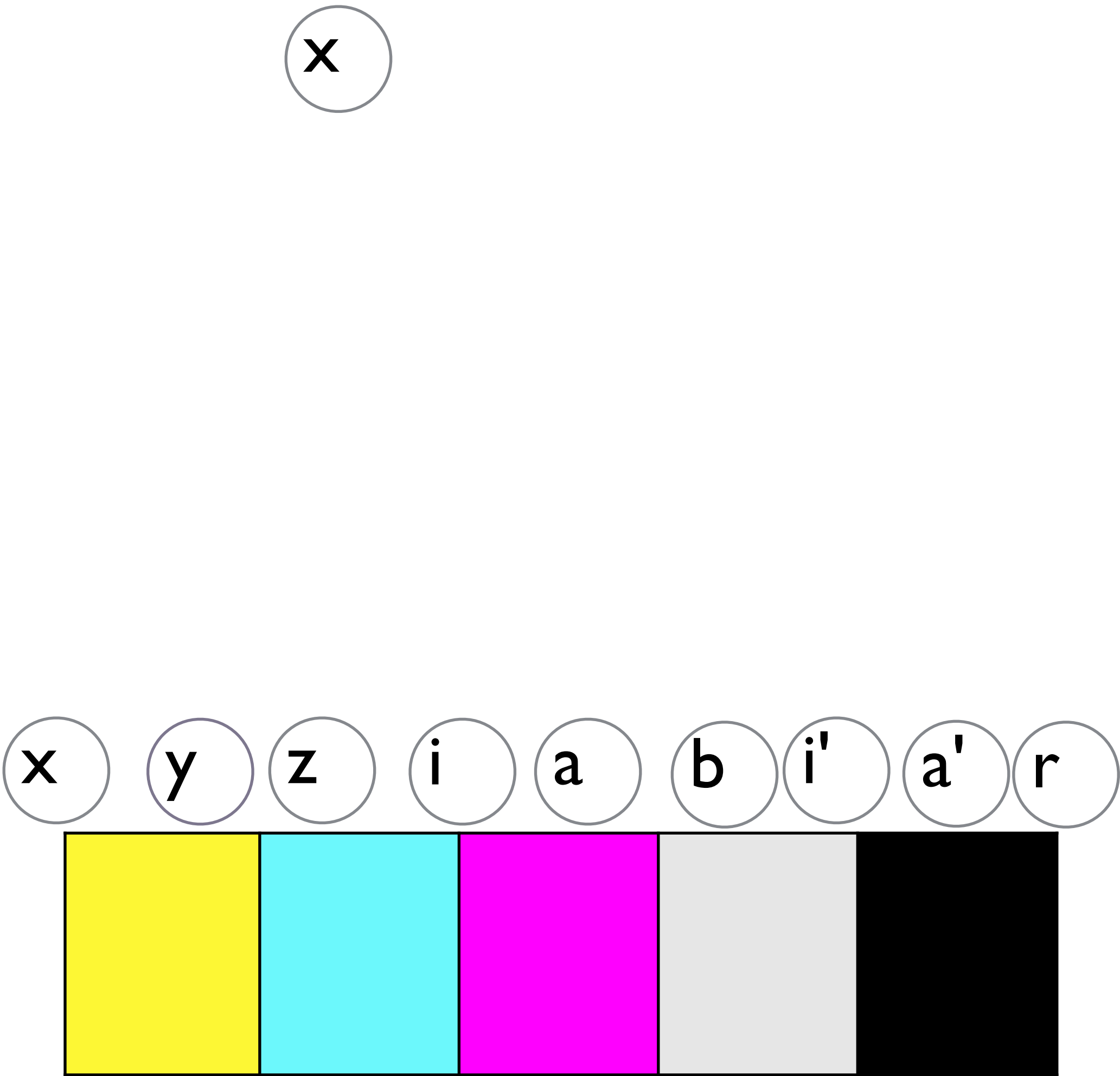




# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

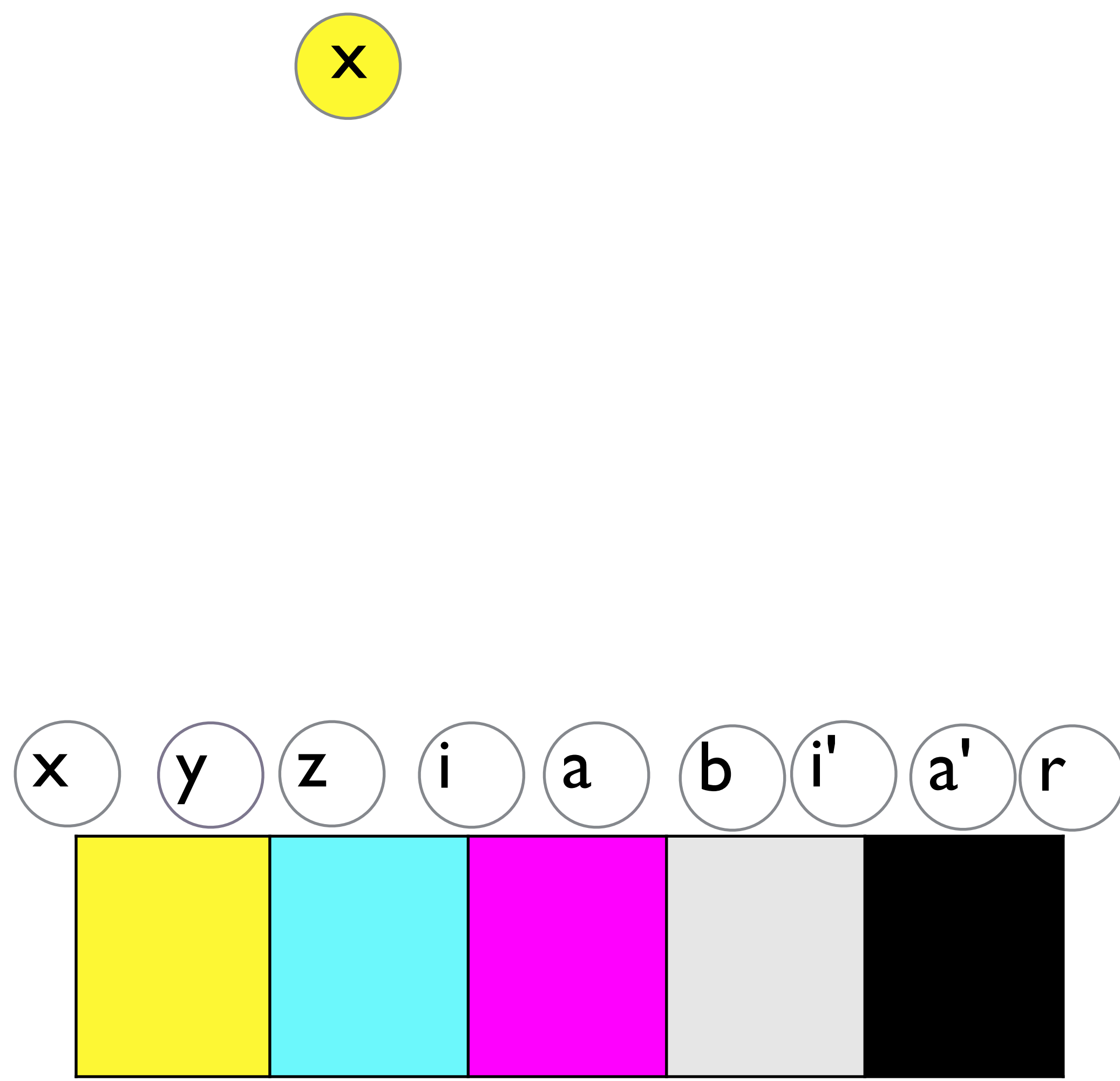
# Interference Graph



# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

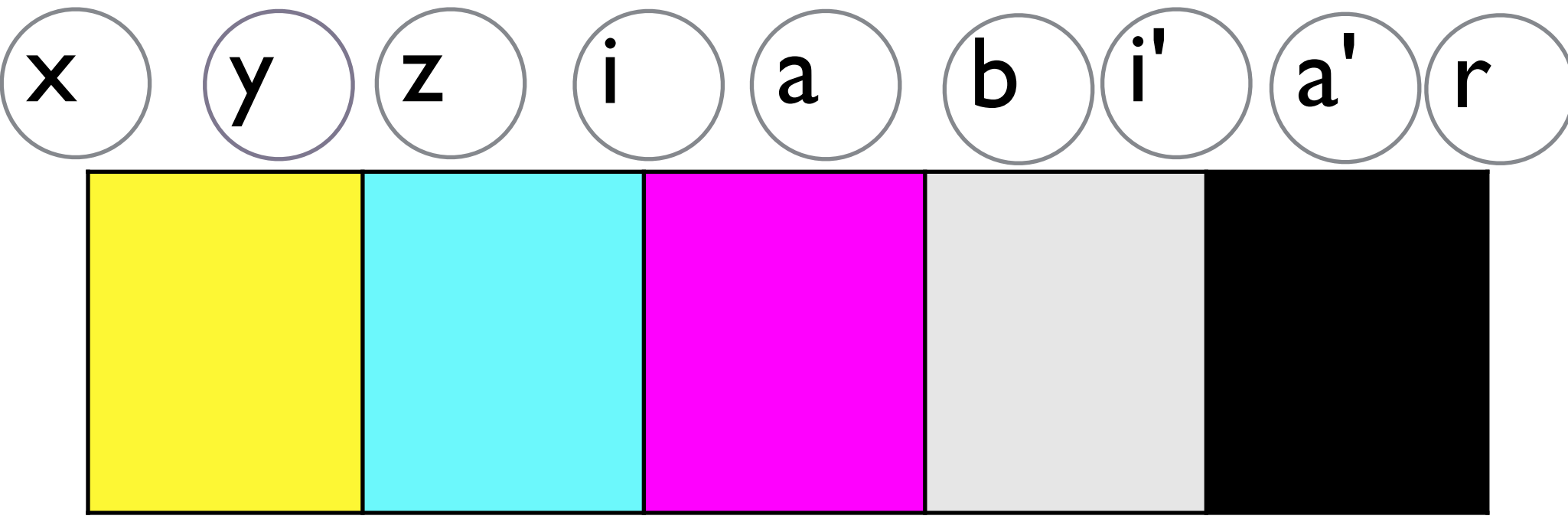
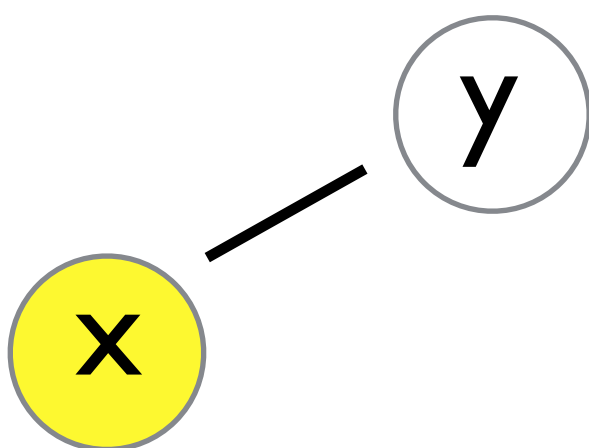
# Interference Graph



# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

# Interference Graph

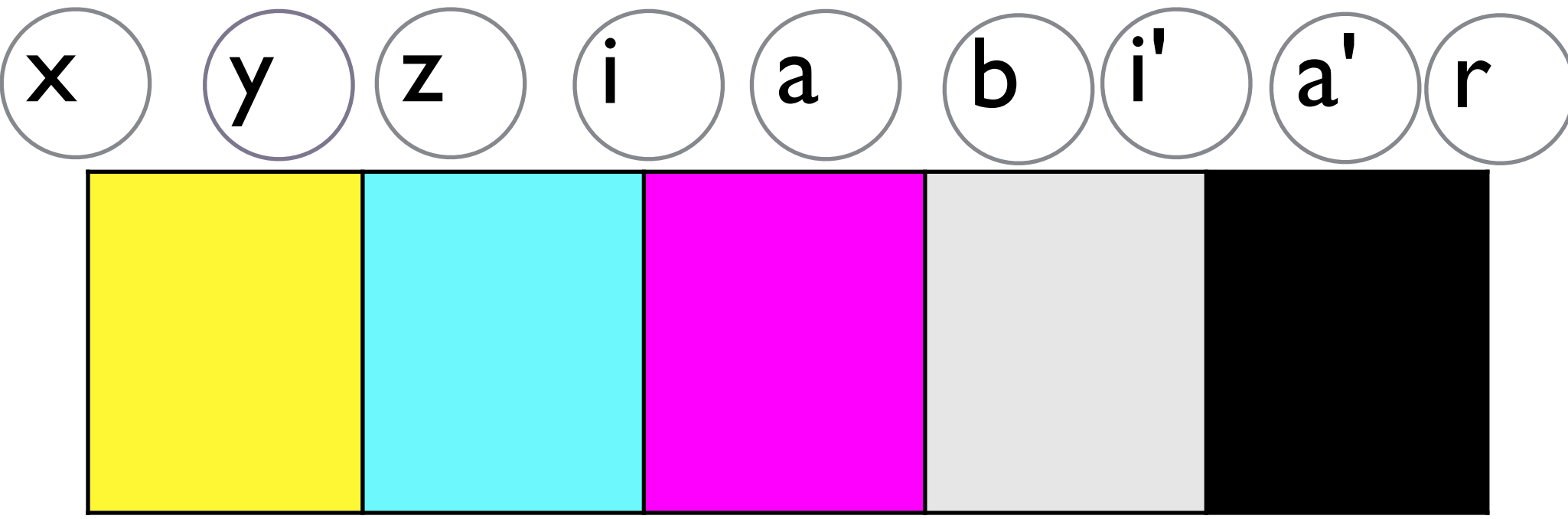
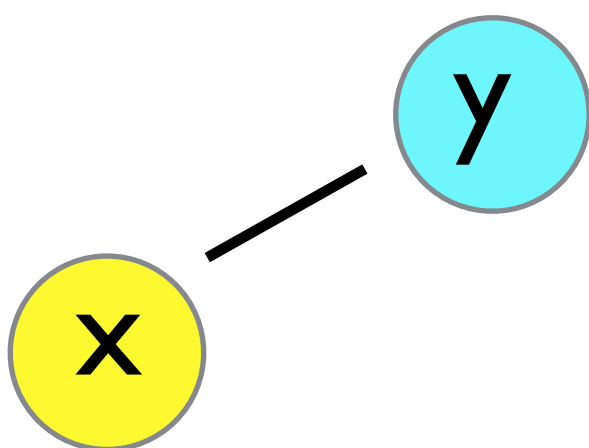




# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

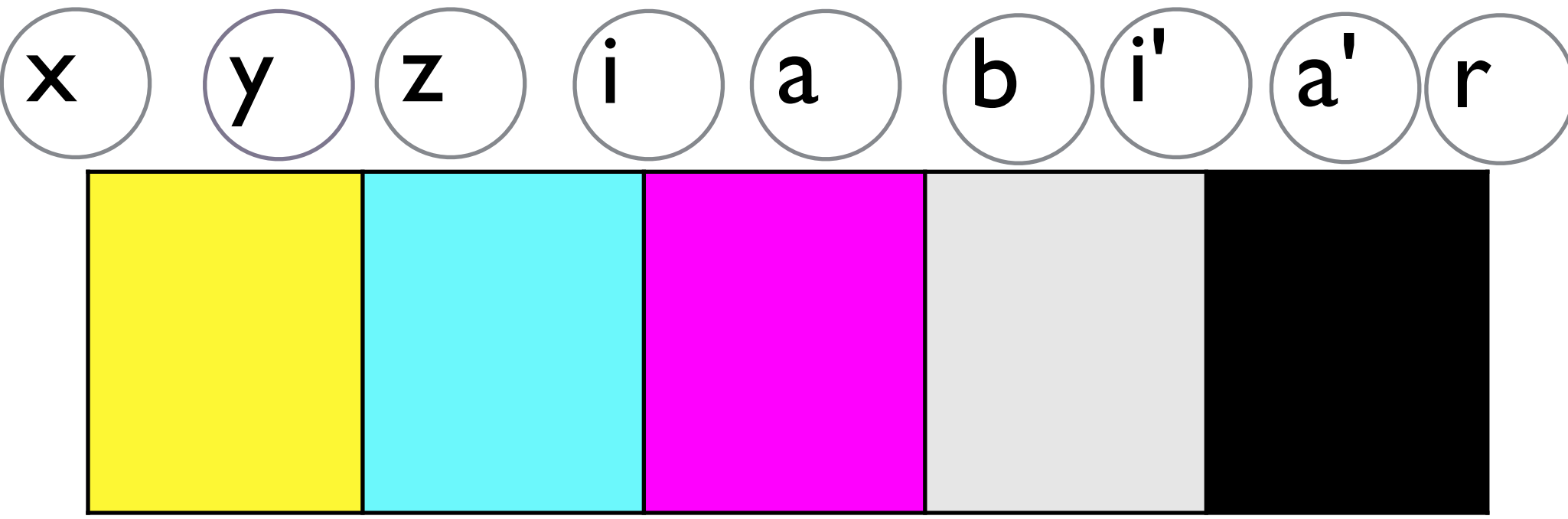
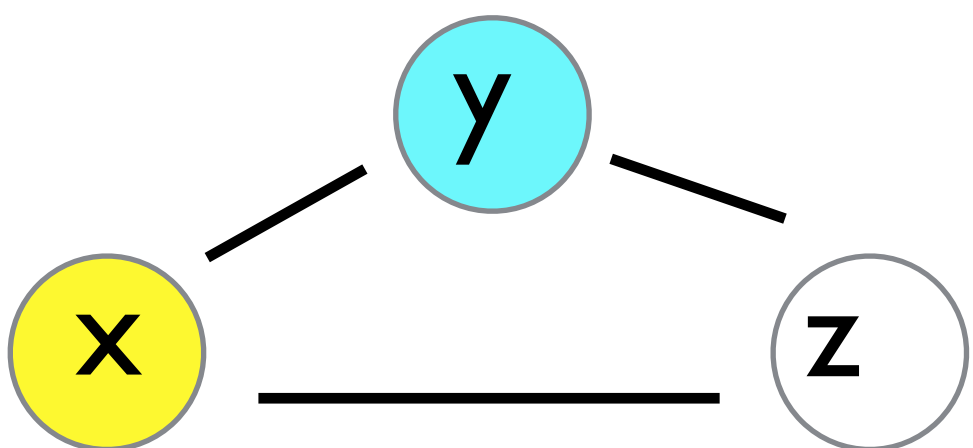
# Interference Graph



# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

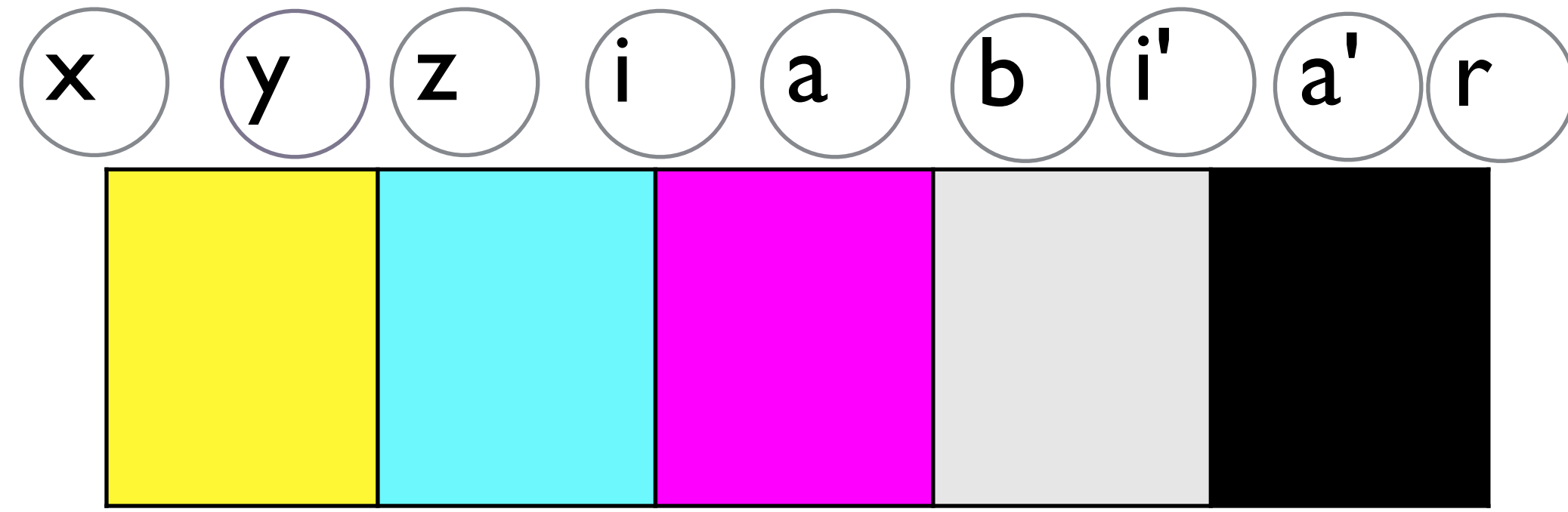
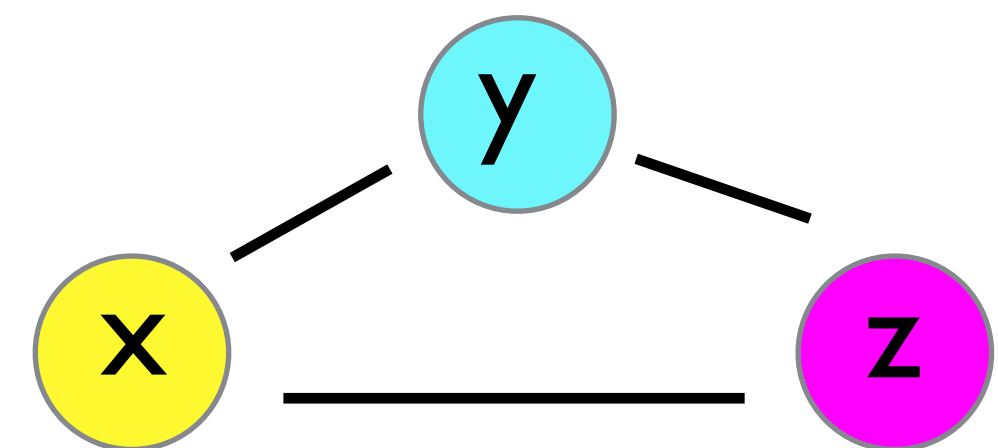
# Interference Graph



# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

# Interference Graph

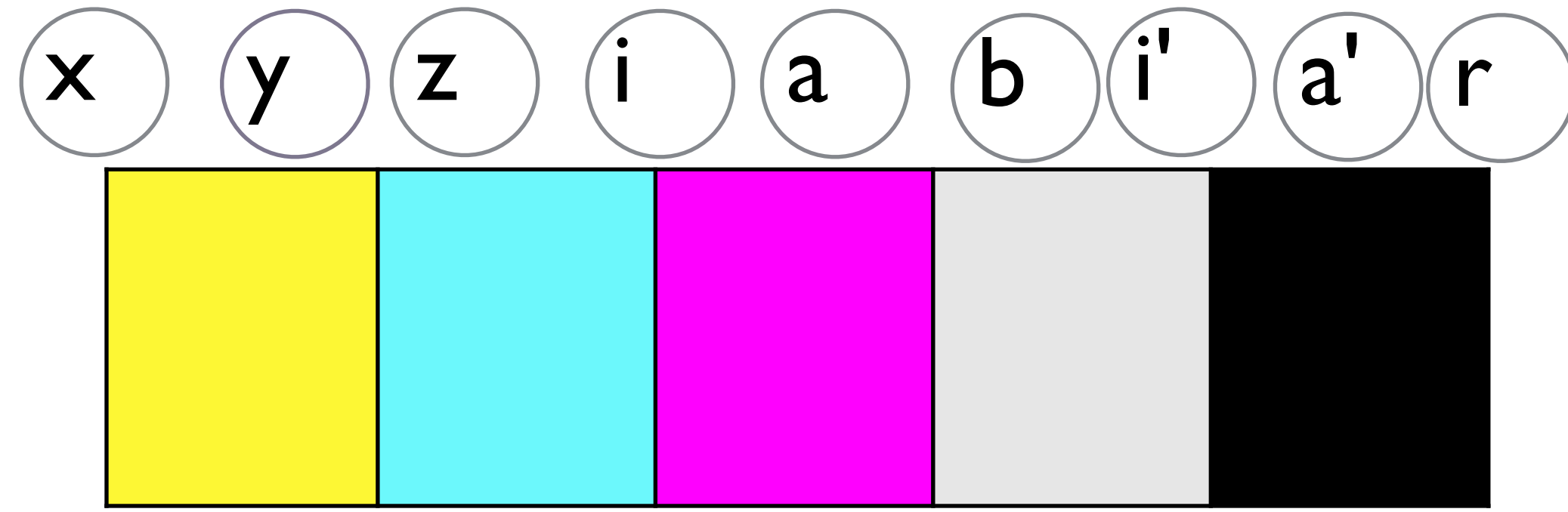
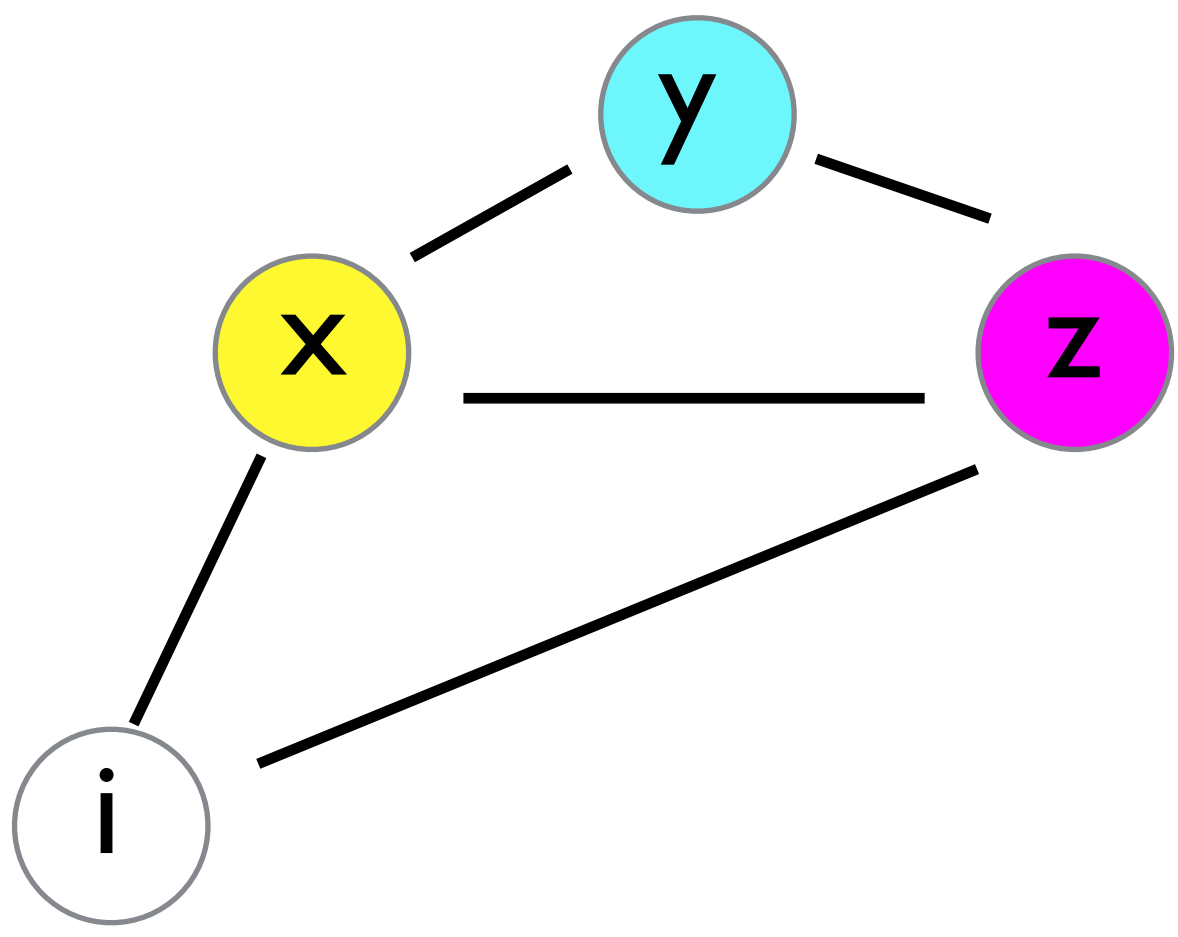




# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

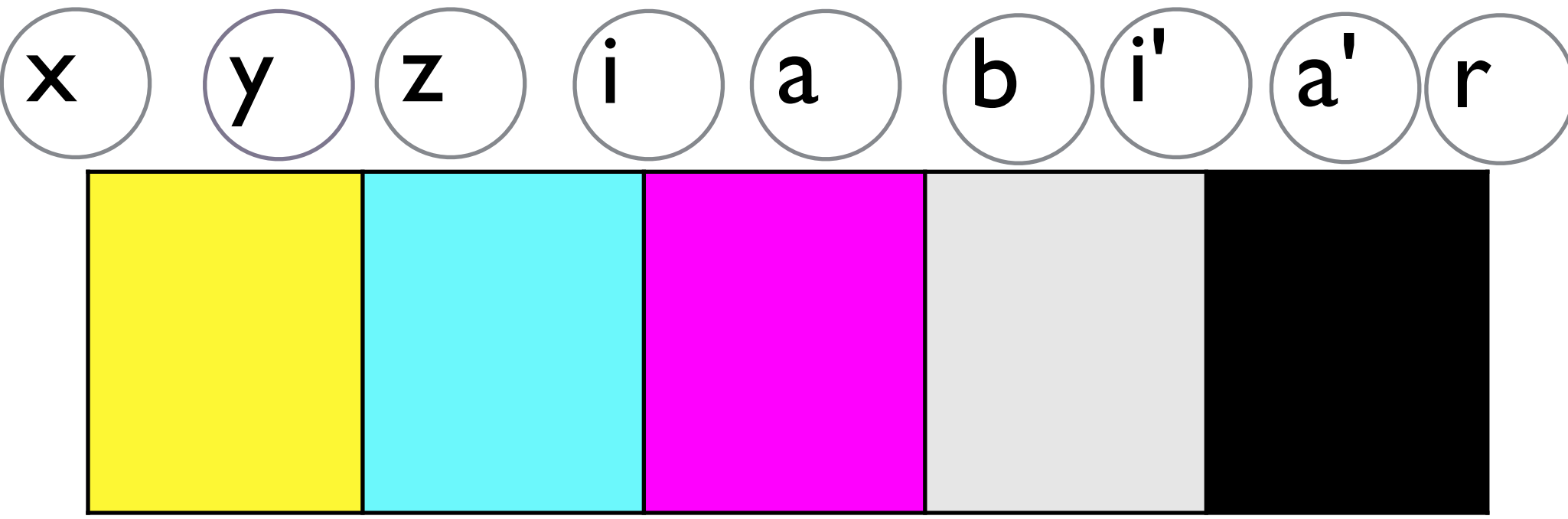
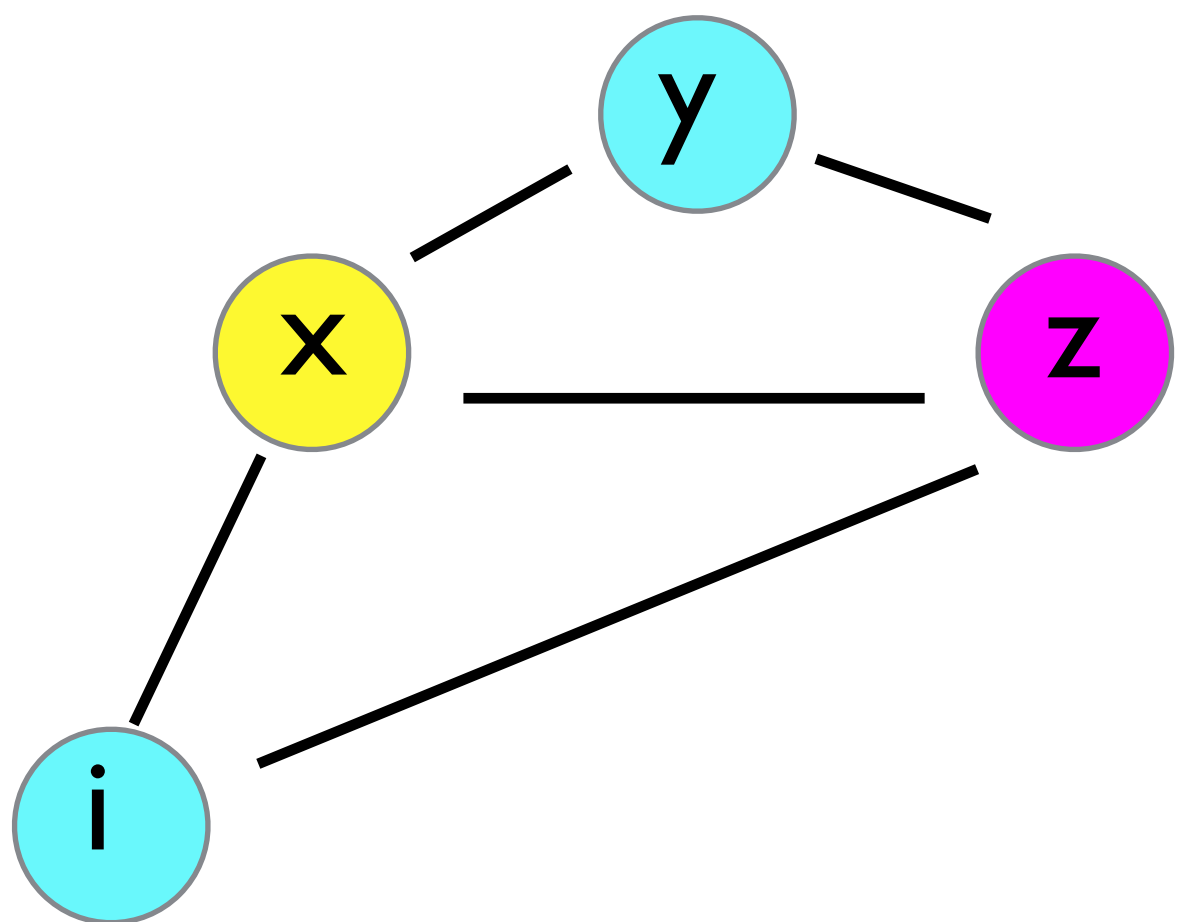
# Interference Graph



# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

# Interference Graph

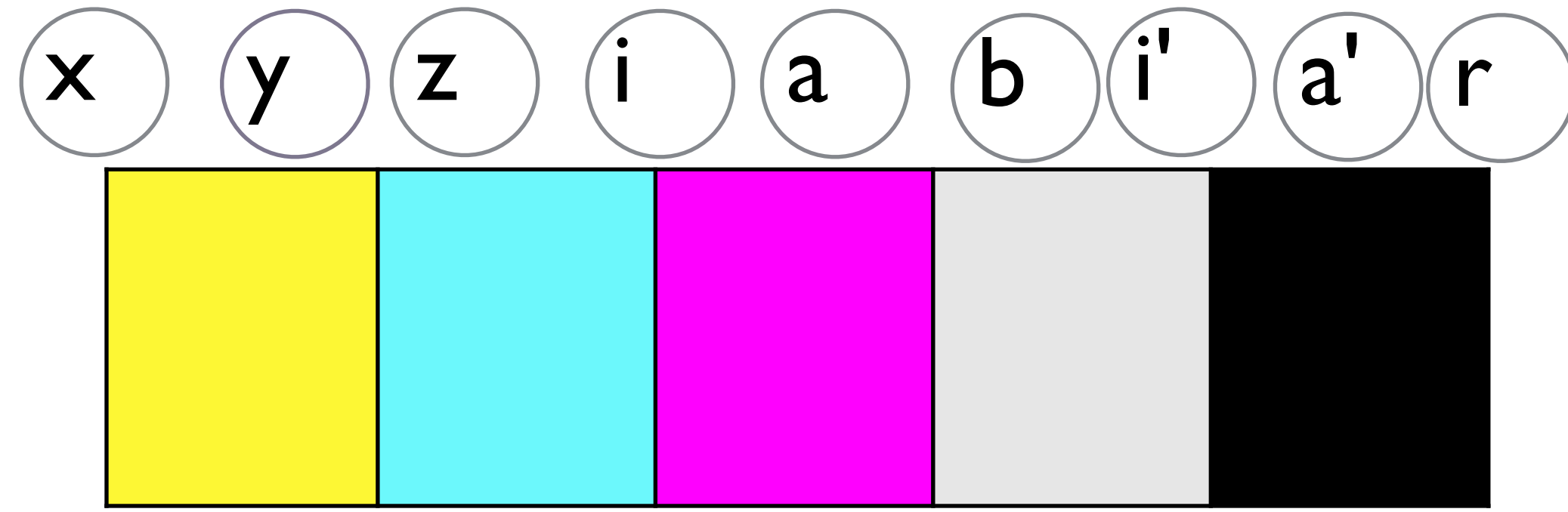
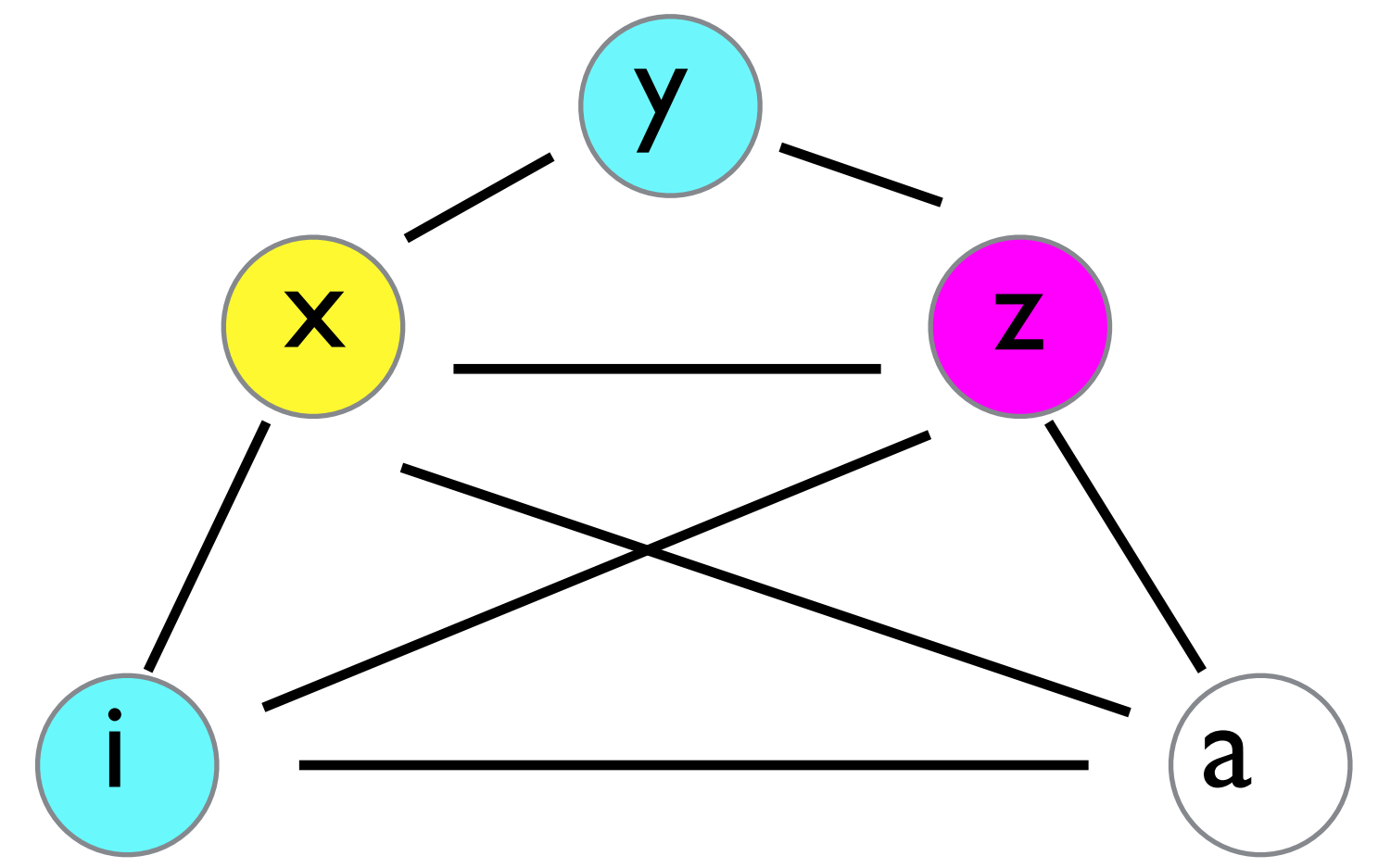




# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

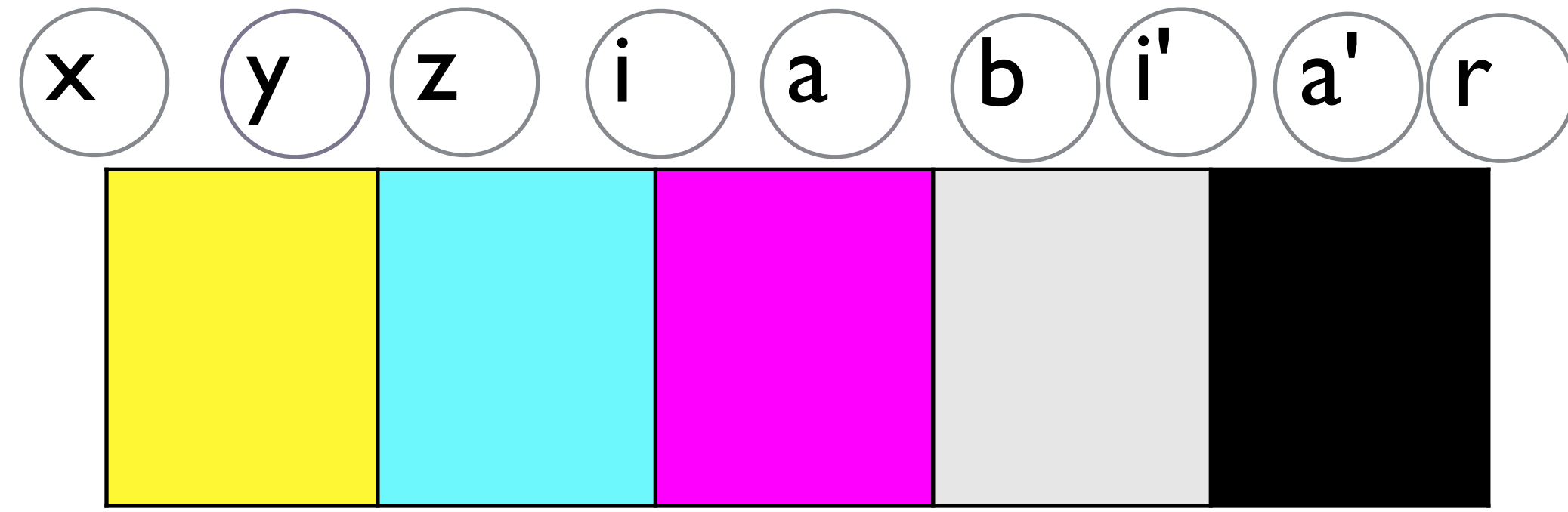
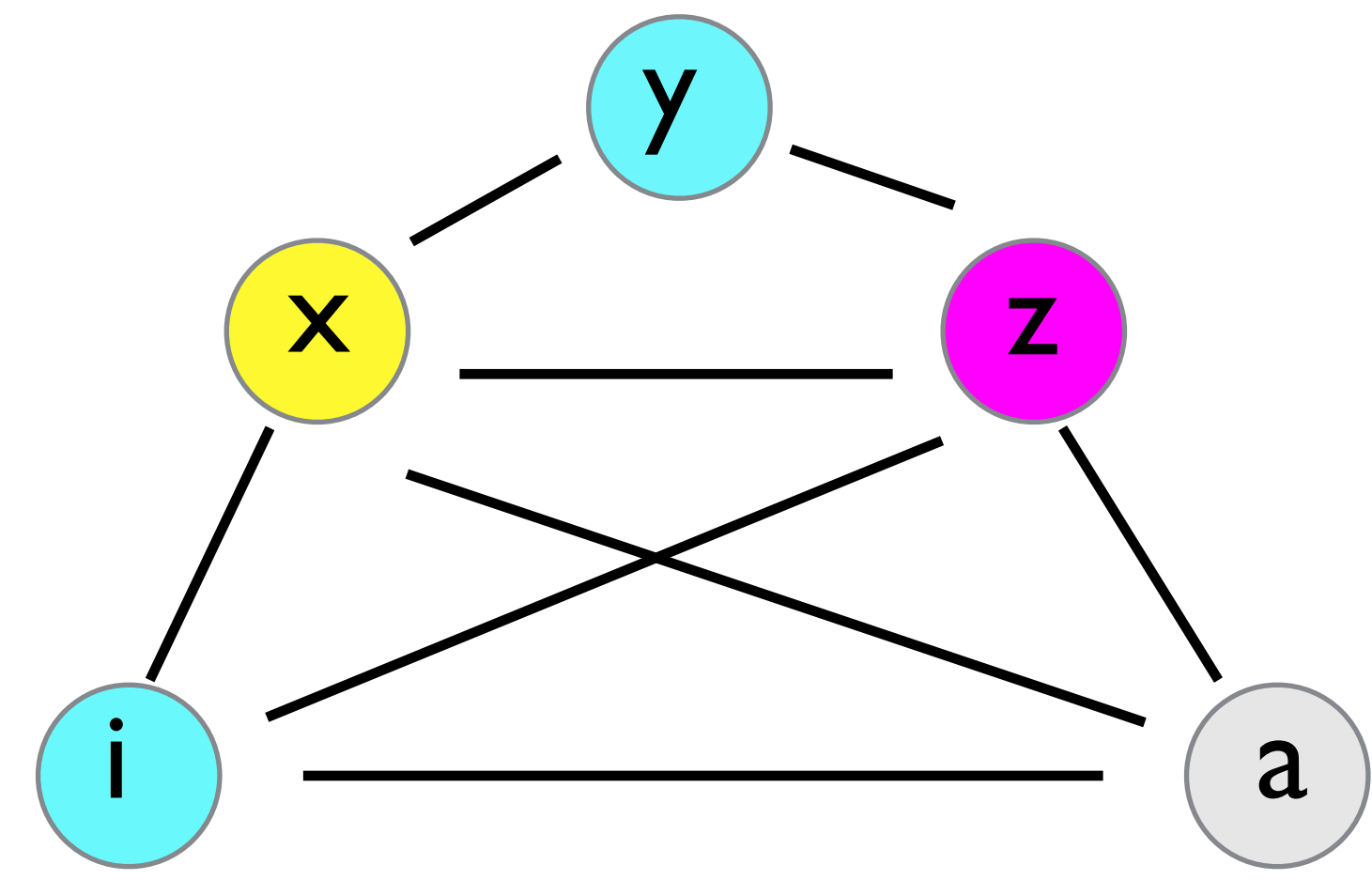
# Interference Graph



# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

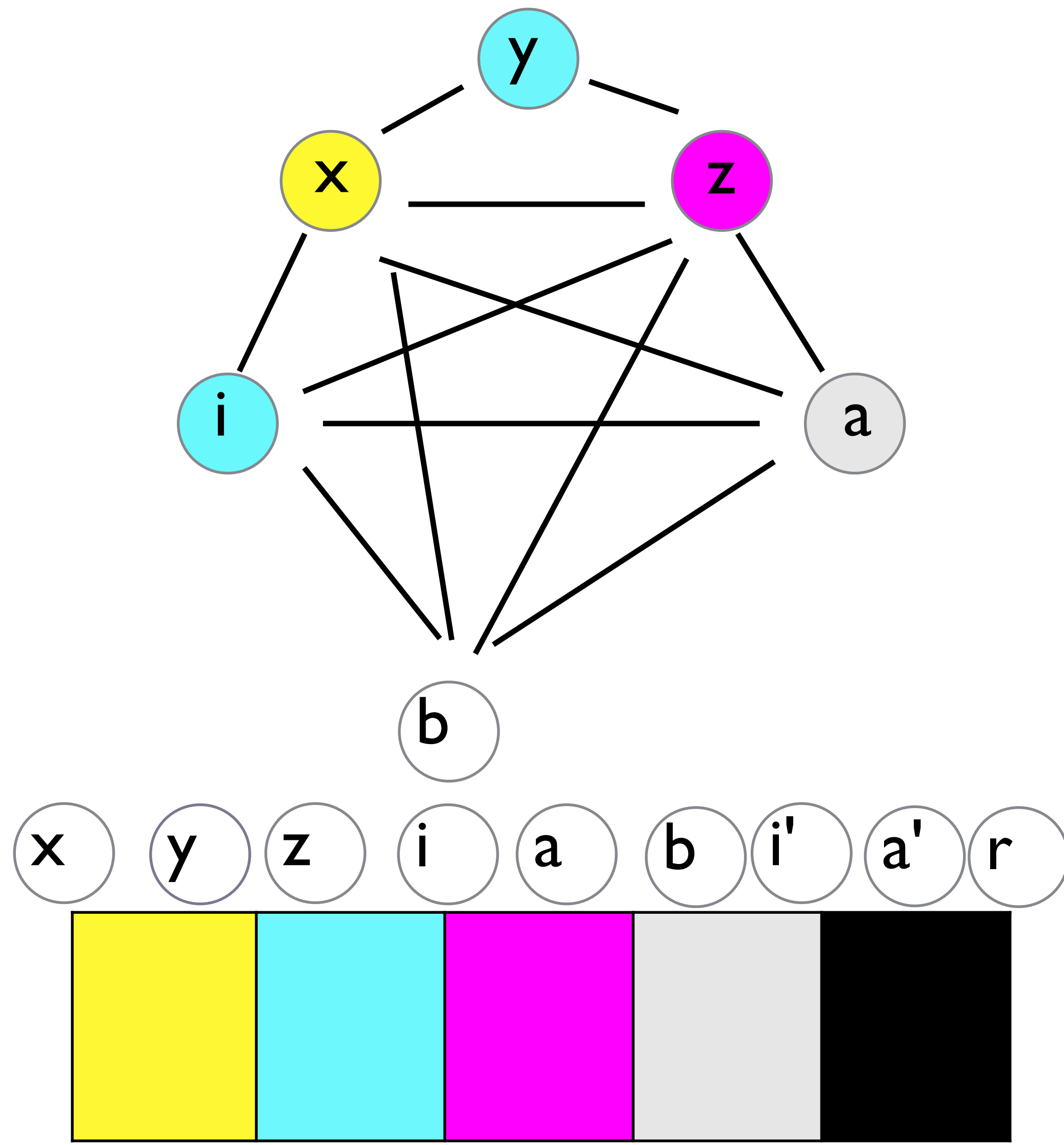
# Interference Graph



# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

# Interference Graph

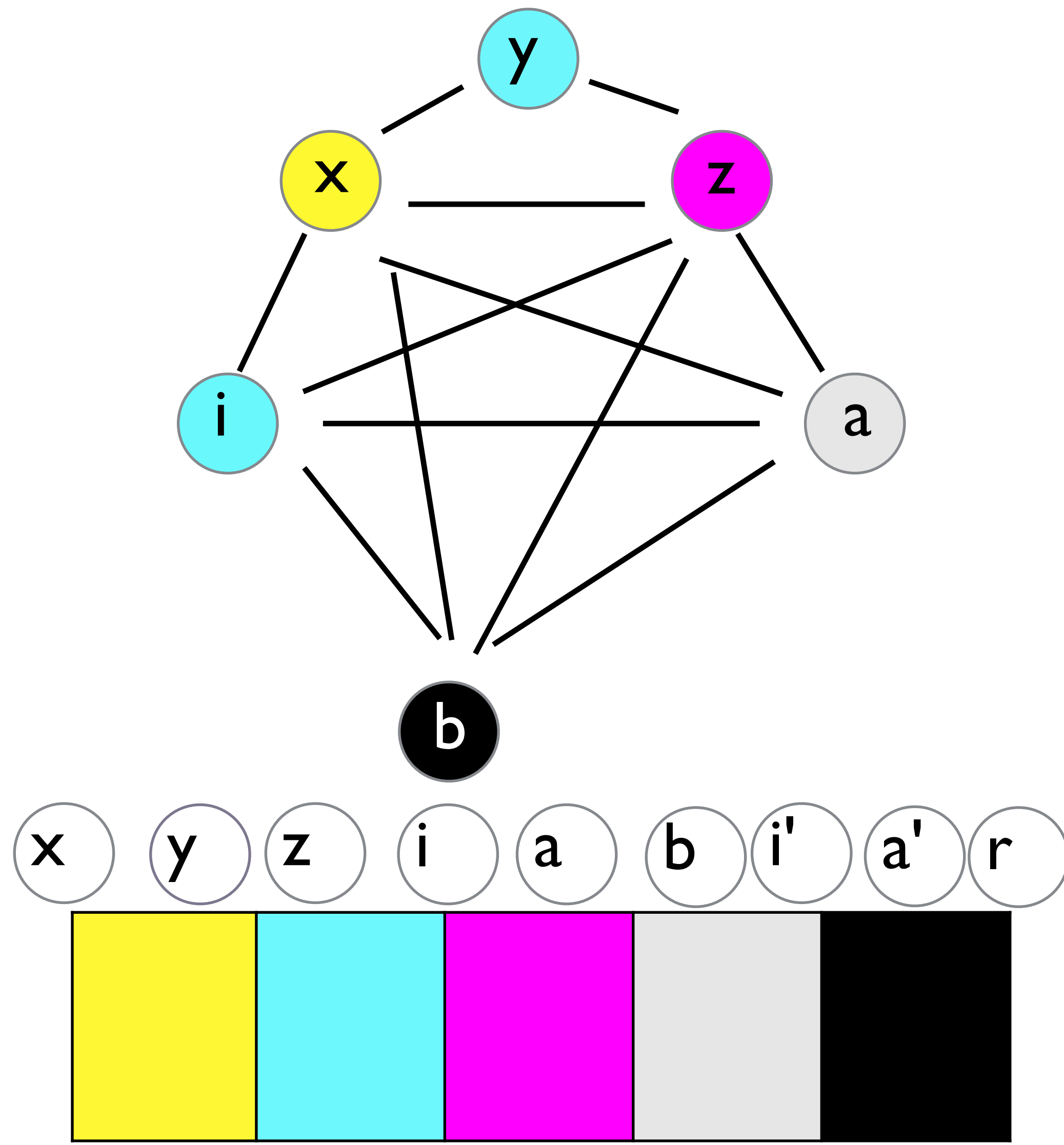




# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

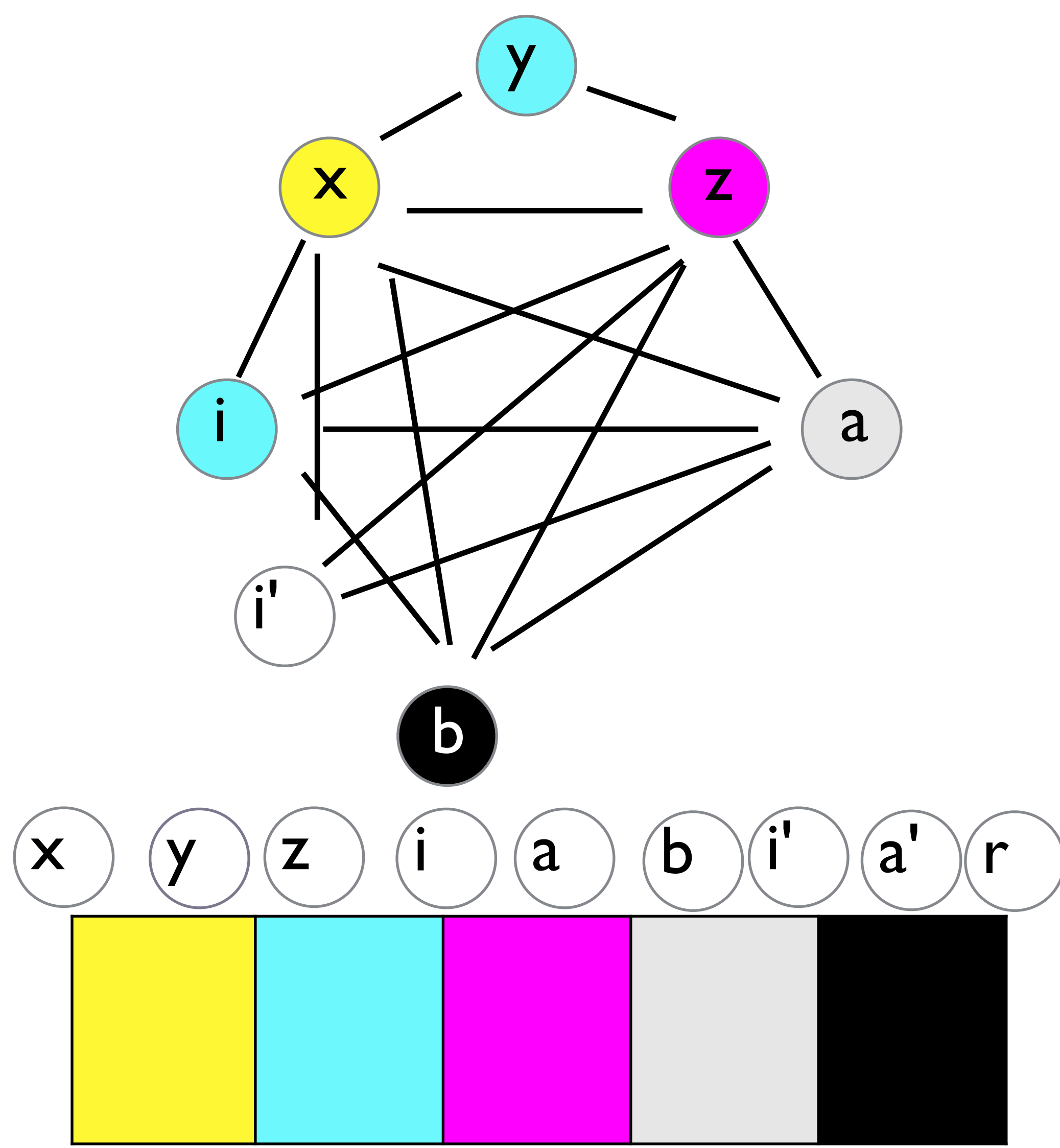
# Interference Graph



# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

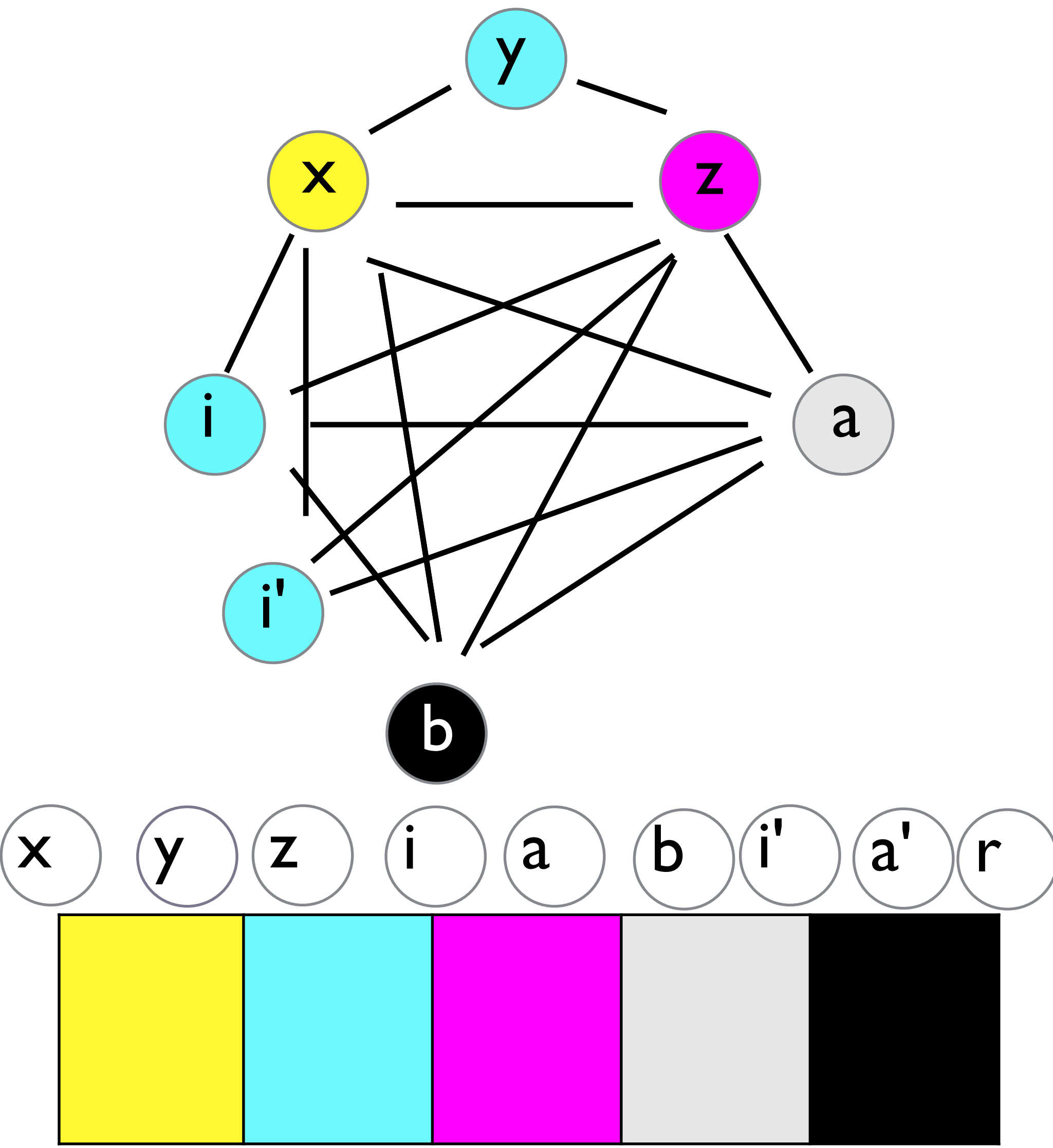
# Interference Graph



# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

# Interference Graph

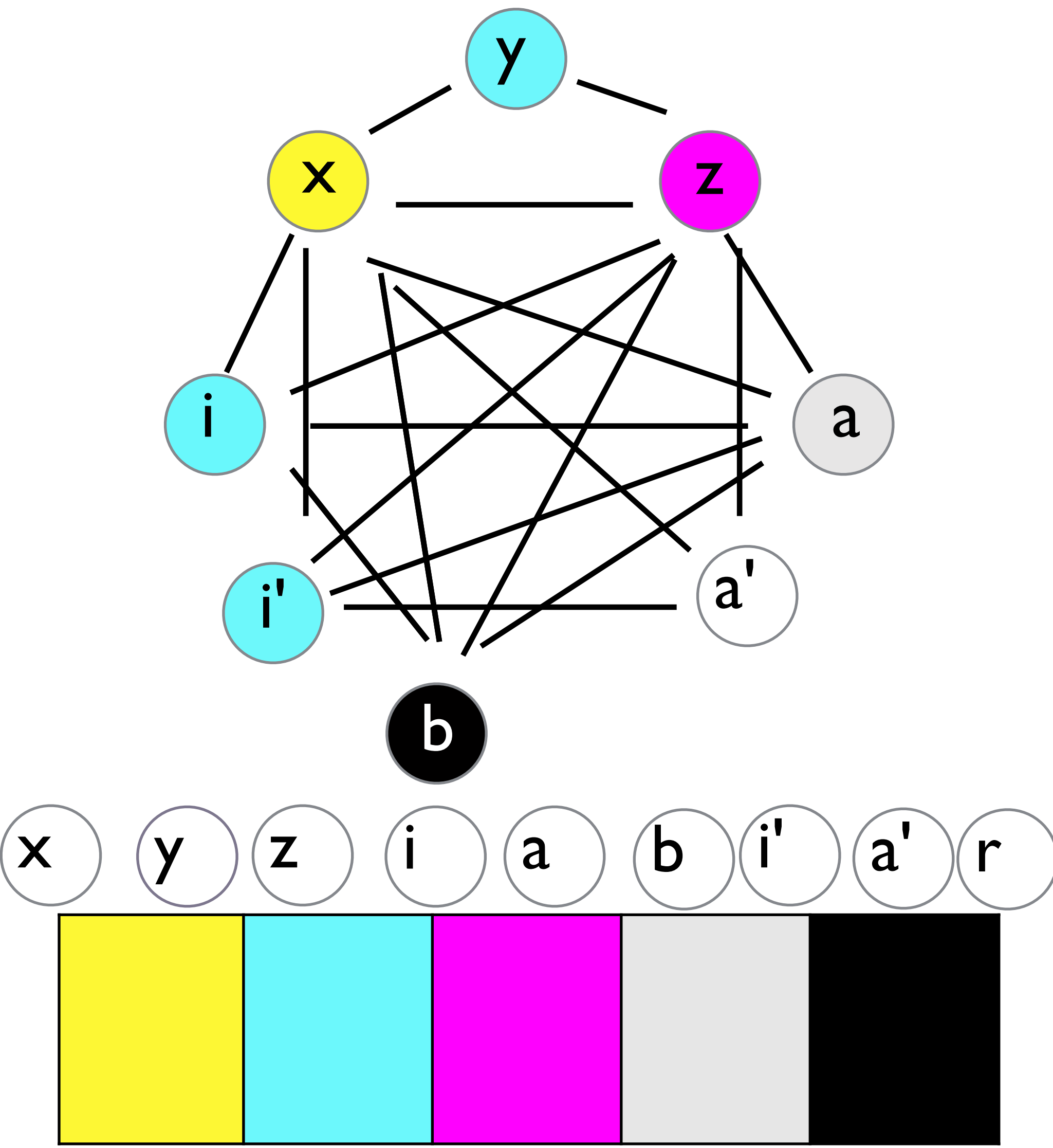




# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

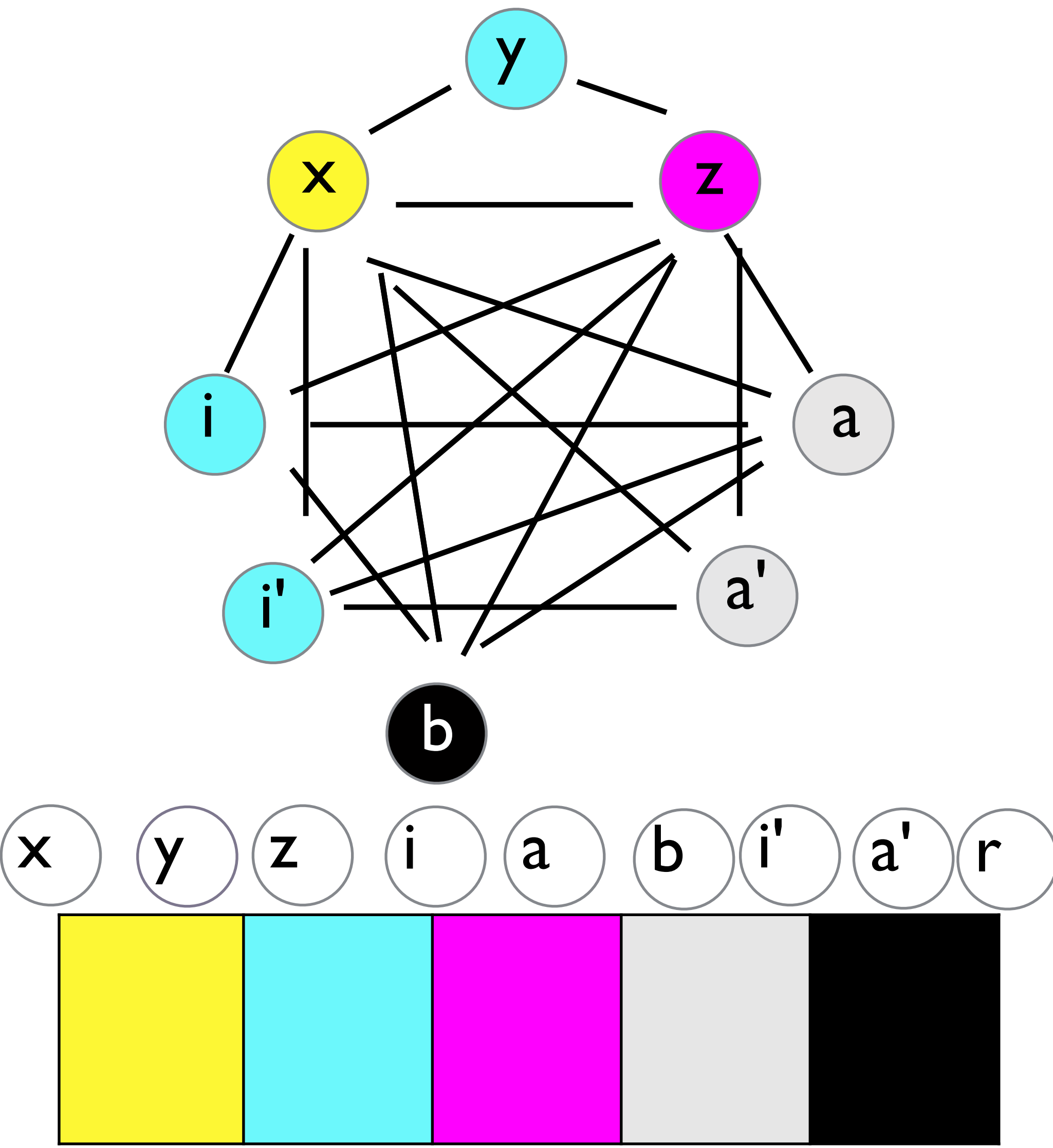
# Interference Graph



# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

# Interference Graph

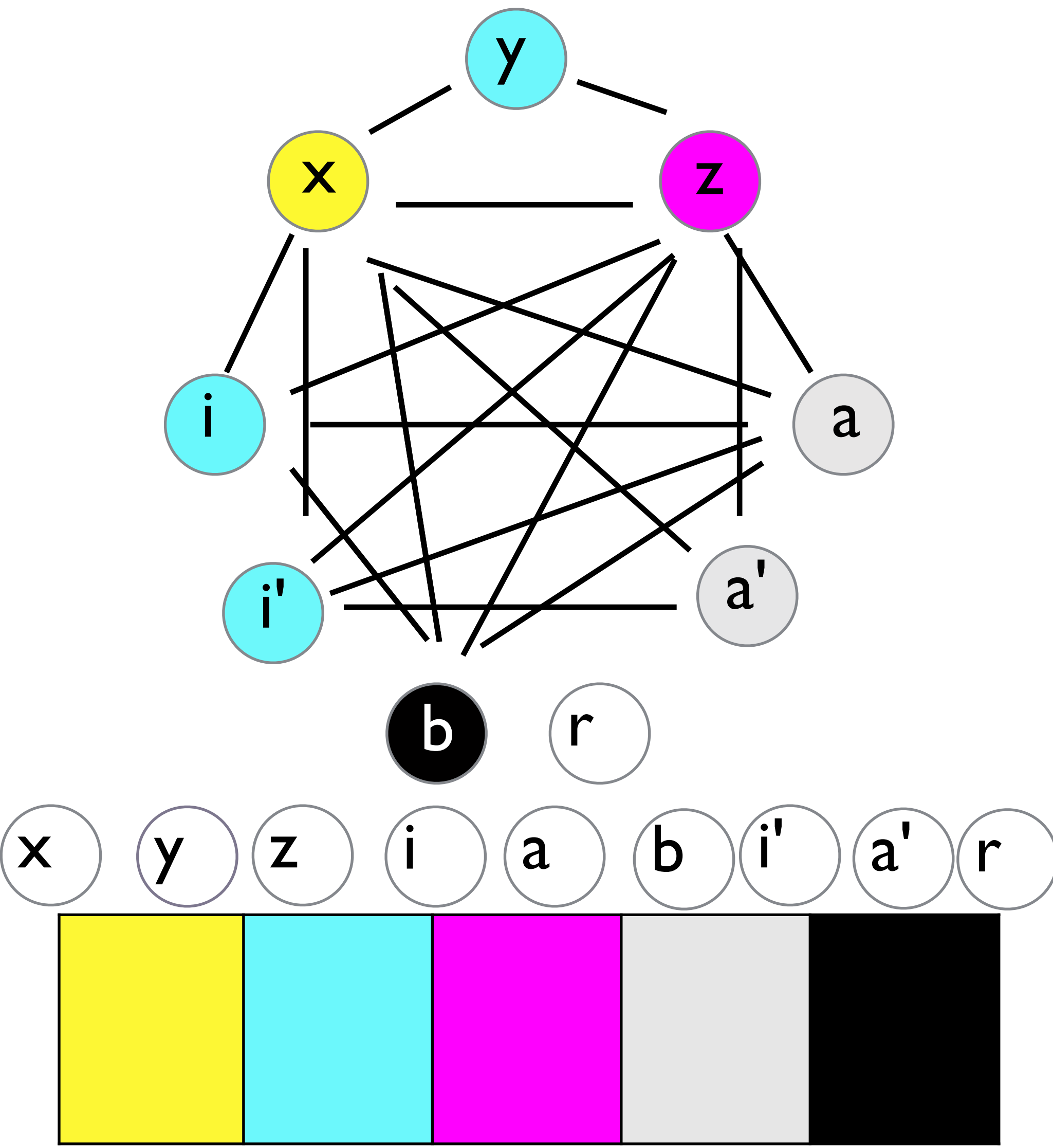




# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

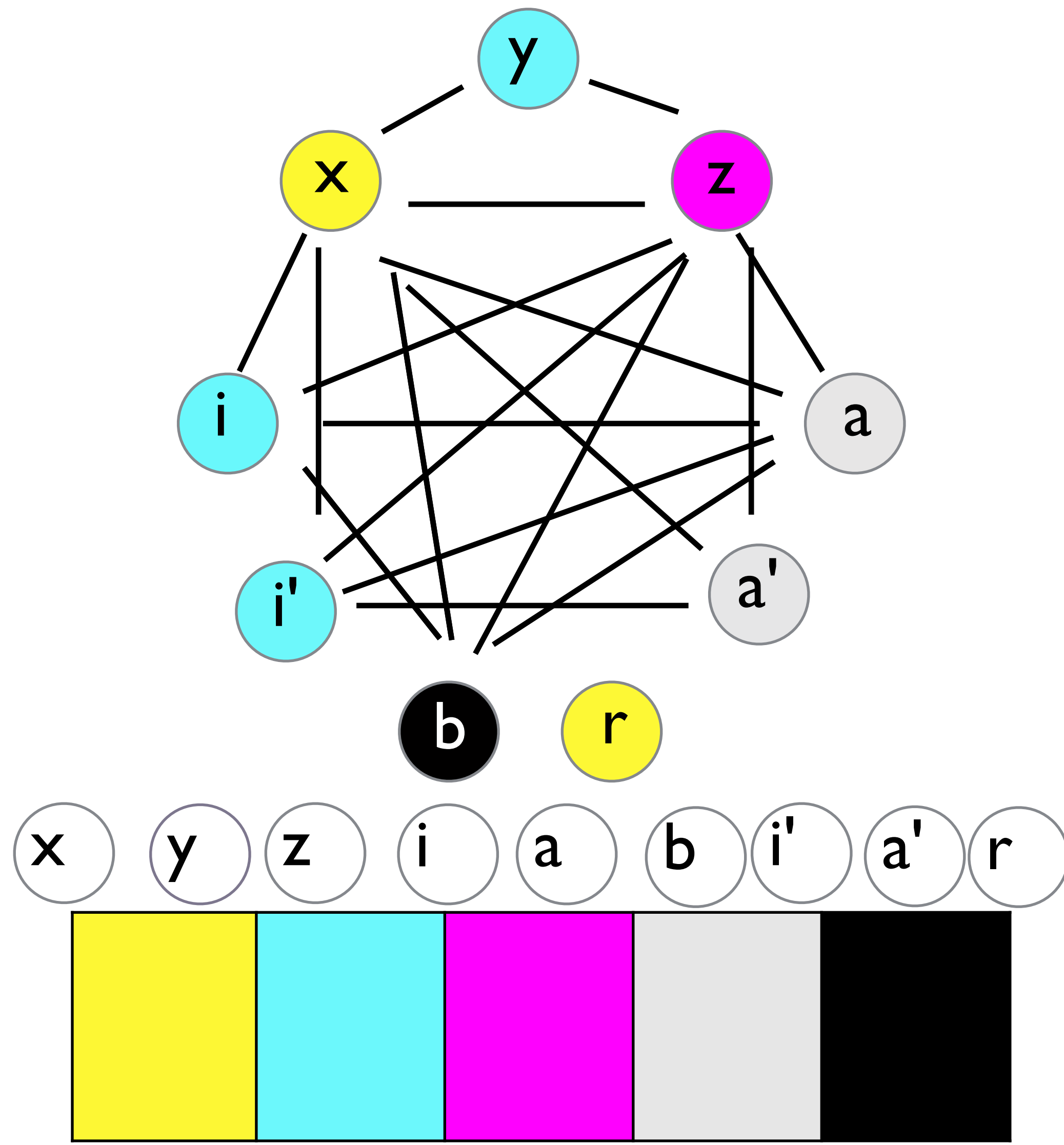
# Interference Graph



# SSA Program

```
f(x,y,z):  
  loop(i,a):  
    thn():  
      r = a * z  
      ret r  
    els():  
      i' = i - 1  
      a' = a + x  
      br loop(i', a')  
  b = i == 0  
  cbr b thn() els()  
br loop(y, 0)
```

# Interference Graph



# Incorporating Register Allocation

---

Perform register allocation on all the **blocks** of our SSA program. Treat function blocks specially.



# Effects on Codegen

---

Store results directly in the output register

$x = y - z$

mov rx, ry

sub rx, rz

Does this always work? Need to be careful if output register is the same as one of the input registers (e.g.,  $rx = rz$ )

Can either use a scratch register, or op-specific tricks:

sub rx, ry

imul rx, -1

# Spilling

---

If a variable is picked to be spilled:

- When it is assigned to, store the result in memory
- When it is used, access memory

One issue:

$$x = y + z$$

If all x,y,z are spilled, cannot implement this without a register.

Easy solution: reserve one scratch register for this purpose:

```
mov r, [rsp - off(y)]
```

```
add r, [rsp - off(z)]
```

```
mov [rsp - off(x)], r
```

# Implementing Branch with Arguments

---

```
f(a,b,c): ...  
...  
br f(x,y,z)
```

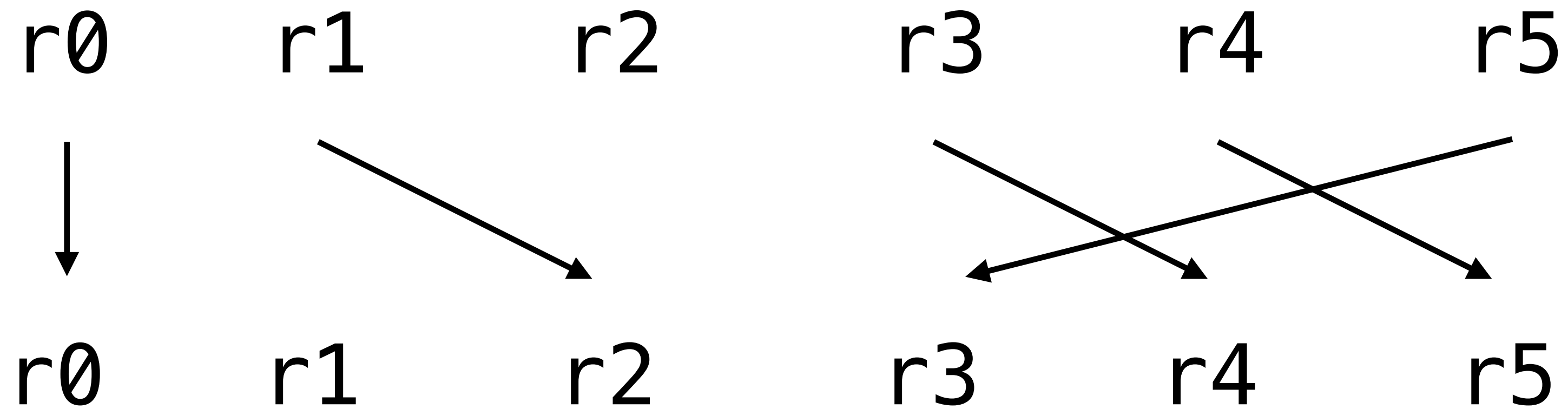
```
mov r_a, r_x  
mov r_b, r_y  
mov r_c, r_z  
jmp f
```

what if a,b,c registers and  
x,y,z registers overlap?



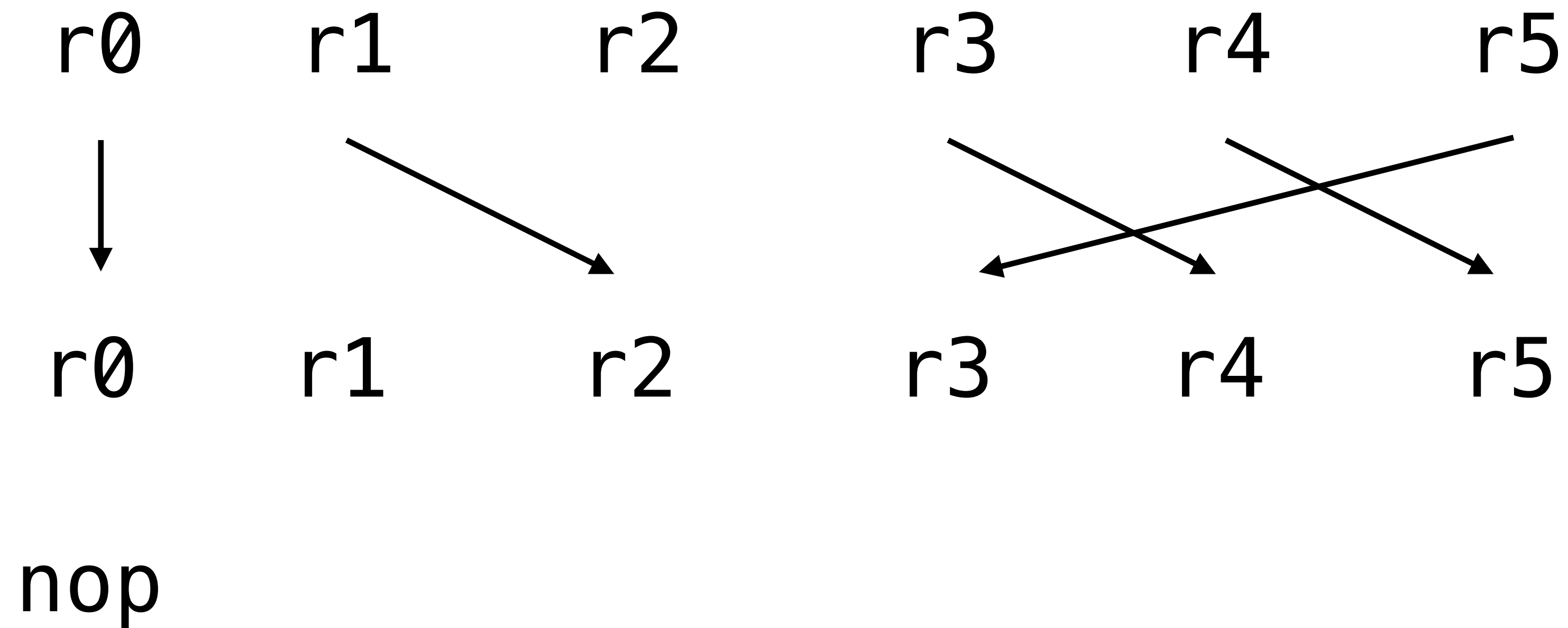
# Implementing Branch with Arguments

---



# Implementing Branch with Arguments

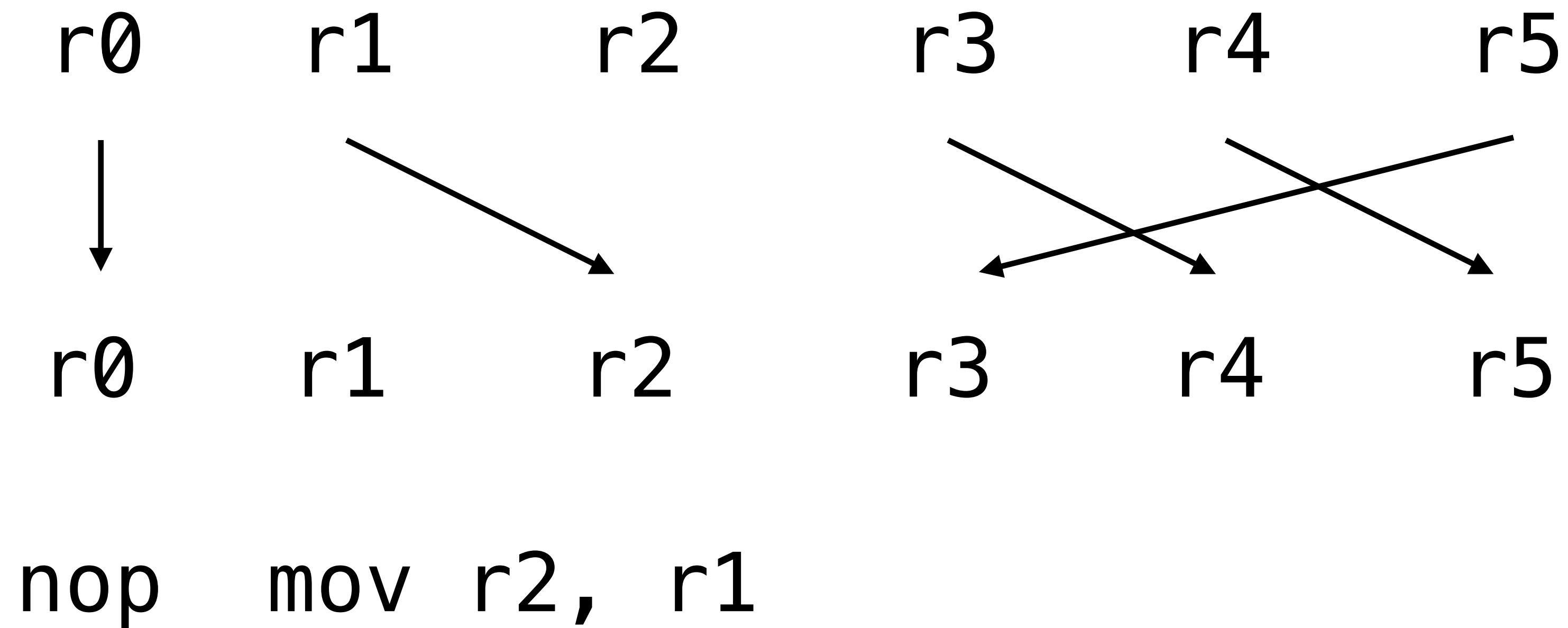
---





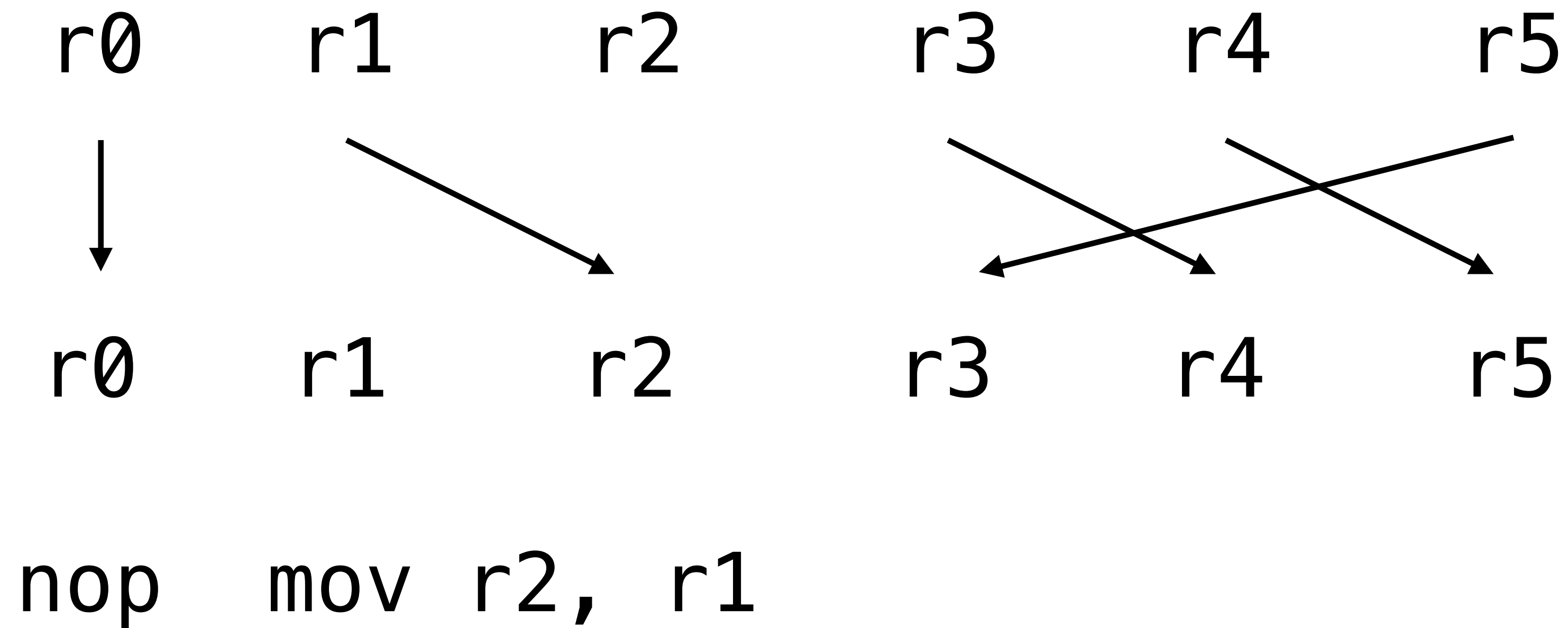
# Implementing Branch with Arguments

---



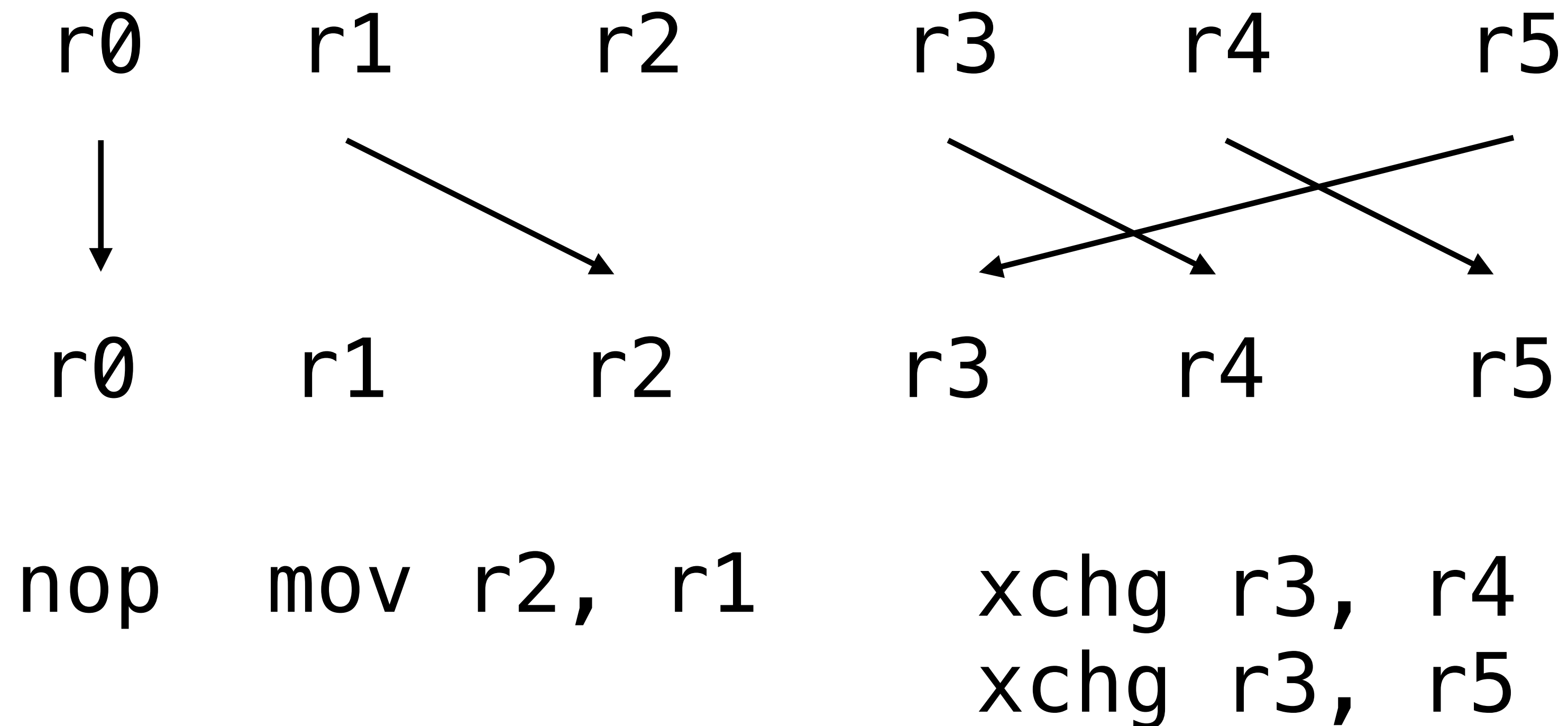
# Implementing Branch with Arguments

---



# Implementing Branch with Arguments

---



xchg is like mov, but **exchanges** the values without need for an extra register.  
Faster than xor swapping

SSA reg allocation is polytime, but minimizing the resulting number of movs/xchg is NP hard



# Register Allocation vs Calling Conventions

---

Now that we are using registers we need to take care to respect treatment of registers in the calling conventions we use.

# Register Allocation vs Calling Conventions

---

Now that we are using registers we need to take care to respect treatment of registers in the calling conventions we use.

In System V AMD 64 Calling convention, registers are divided into two classes:

- **volatile** aka **caller-save**: when you make a call, the value of these registers may change when the callee returns
- **non-volatile** aka **callee-save**: when you make a call, the value of these registers will be the same when the callee returns

# Volatile/Caller Save registers

---

**volatile** aka **caller-save**

$x = \dots$

$y = f(z)$

$z = x + y$

if **x** is stored in a volatile register, its value may be overwritten by the function **f**.

- Simple solution: save all live volatiles to the stack before a call, restore after the call
- Better solution: add nodes to interference graph for volatile registers, add conflicts at every non-tail call



# Non-volatile/Callee Save registers

---

$y = \dots$

$z = x + y$

ret z

if **y** is stored in a **non-volatile** register, the value of the register must be **saved** on entry and **restored** when we return

- Solution: save all used non-volatiles to the stack at the beginning of every global function def, restore them before every return/external tail call
- Start spilled variables **after** the saved non-volatile registers

# Implementing function blocks

---

```
fun f_fun(a,b,c,...):  
    br f_tail(a,b,c,...)
```

1. Save all used non-volatiles/callee-save registers
2. Treat f\_fun's args are pre-determined by the calling convention, otherwise similar to any branch with args

```
mov [rsp - 8], rbx  
mov [rsp - 16], rbp  
...  
; br f_tail(...)
```



# Implementing ret

---

ret x

- Move x into rax
- Restore non-volatile/callee-saves

mov rax, loc(x)

mov rbx, [rsp - 8]

mov rbp, [rsp - 16]

...

**Optimization, Possible Values/  
Assertion Removal, Dataflow Analysis**

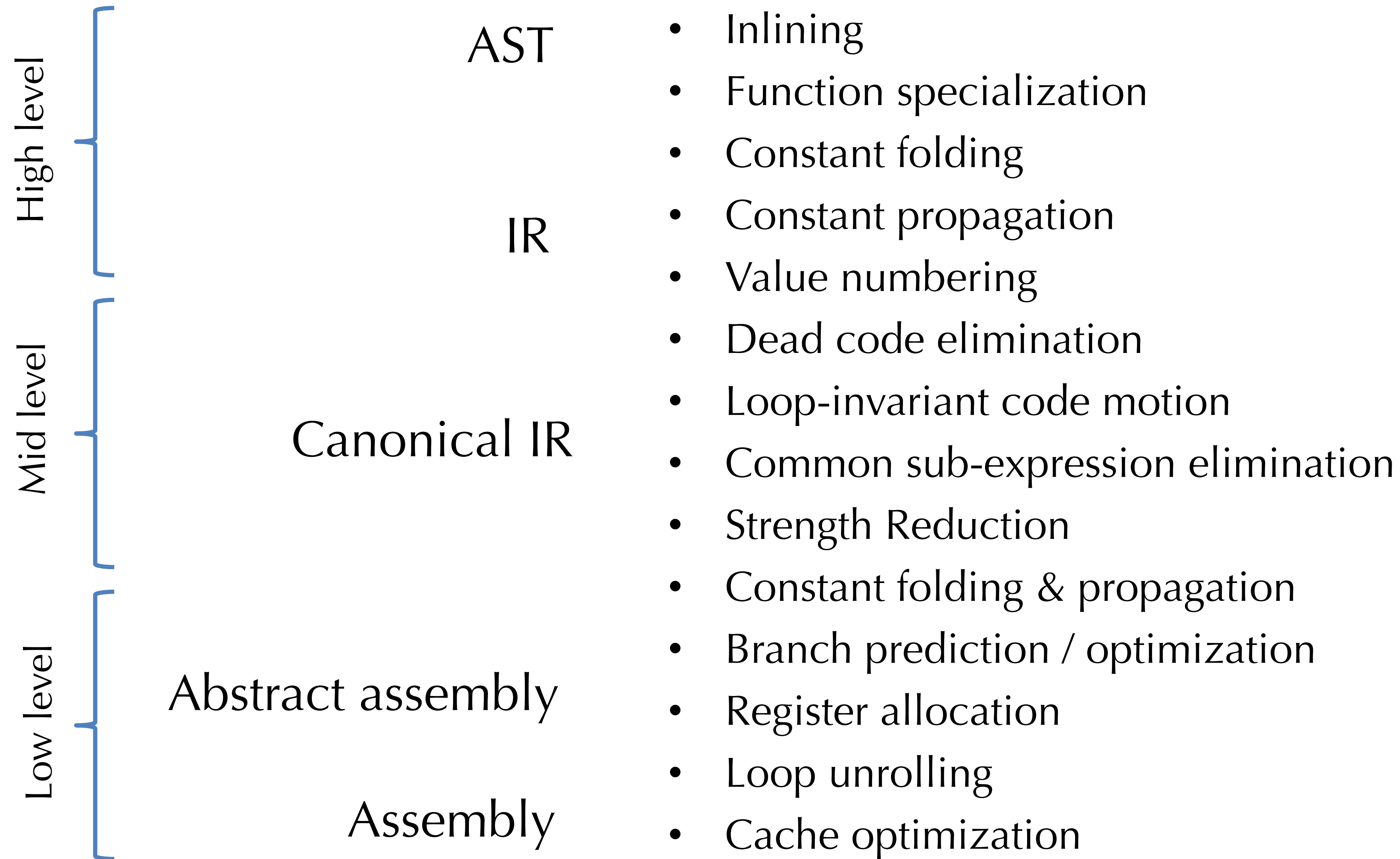


Why optimize?

# OPTIMIZATIONS, GENERALLY



# When to apply optimization



# Safety

- Whether an optimization is *safe* depends on the programming language semantics.
  - Languages that provide weaker guarantees to the programmer permit more optimizations but have more ambiguity in their behavior.
  - e.g., In C, loading from uninitialized memory is undefined, so the compiler can do anything if a program reads uninitialized data.
  - e.g., In Java tail-call optimization (which turns recursive function calls into loops) is not valid because of “stack inspection”.
- Example: *loop-invariant code motion*
  - Idea: hoist invariant code out of a loop

```
while (b) {  
  z = y/x;  
  ...  
}
```

// y, x not updated



```
z = y/x;  
while (b) {  
  ...  
}
```

// y, x not updated

- Is this more efficient?
- Is this safe?



A high-level tour of a variety of optimizations.

# BASIC OPTIMIZATIONS



# Constant Folding

- Idea: If operands are known at compile time, perform the operation statically.

`int x = (2 + 3) * y` → `int x = 5 * y`

`b & false` → `false`

- Performed at every stage of optimization...
- Why?
  - Constant expressions can be created by translation or earlier optimizations

Example: `A[2]` might be compiled to:

`MEM[MEM[A] + 2 * 4]` → `MEM[MEM[A] + 8]`

# Constant Folding Conditionals

|                      |      |
|----------------------|------|
| if (true) S          | → S  |
| if (false) S         | → ;  |
| if (true) S else S'  | → S  |
| if (false) S else S' | → S' |
| while (false) S      | → ;  |
| if (2 > 3) S         | →    |
| if (false) S         | → ;  |



# Algebraic Simplification

- More general form of constant folding
  - Take advantage of mathematically sound simplification rules
- *Mathematical identities:*
  - $a * 1 \rightarrow a$                        $a * 0 \rightarrow 0$
  - $a + 0 \rightarrow a$                        $a - 0 \rightarrow a$
  - $b \mid \text{false} \rightarrow b$                    $b \& \text{true} \rightarrow b$
- *Reassociation & commutativity:*
  - $(a + 1) + 2 \rightarrow a + (1 + 2) \rightarrow a + 3$
  - $(2 + a) + 4 \rightarrow (a + 2) + 4 \rightarrow a + (2 + 4) \rightarrow a + 6$
- *Strength reduction:* (replace expensive op with cheaper op)
  - $a * 4 \rightarrow a \ll 2$
  - $a * 7 \rightarrow (a \ll 3) - a$
  - $a / 32767 \rightarrow (a \gg 15) + (a \gg 30)$
- *Note 1:* must be careful with floating point (due to rounding) and integer arithmetic (due to overflow/underflow)
- *Note 2:* iteration of these optimizations is useful... how much?
- *Note 3:* must be sure that rewrites terminate:
  - commutativity apply like:  $(x + y) \rightarrow (y + x) \rightarrow (x + y) \rightarrow (y + x) \rightarrow \dots$

# Constant Propagation

- If a variable is known to be a constant, replace the use of the variable by the constant
- Value of the variable must be propagated forward from the point of assignment
  - This is a *substitution* operation

Example:

```
int x = 5;  
int y = x * 2;  
int z = a[y];
```

→

```
int y = 5 * 2;  
int z = a[y];
```

→

```
int y = 10;  
int z = a[y];
```

→

```
int z = a[10];
```

- To be most effective, constant propagation should be interleaved with constant folding

# Copy Propagation

- If one variable is assigned to another, replace uses of the assigned variable with the copied variable.
- Need to know where copies of the variable propagate.
- Interacts with the scoping rules of the language.

- Example:

```
x = y;  
if (x > 1) {  
    x = x * f(x - 1);  
}
```

→

```
x = y;  
if (y > 1) {  
    x = y * f(y - 1);  
}
```

- Can make the first assignment to x **dead code** (that can be eliminated).



# Dead Code Elimination

- If a side-effect free statement can never be observed, it is safe to eliminate the statement.

```
x = y * y // x is dead!  
...      // x never used  
x = z * z
```

```
→      ...  
x = z * z
```

- A variable is **dead** if it is never used after it is defined.
  - Computing such *definition* and *use* information is an important component of program analysis
- Dead variables can be created by other optimizations...

# Unreachable/Dead Code

- Basic blocks not reachable by any trace leading from the starting basic block are *unreachable* and can be deleted.
  - Performed at the IR or assembly level
  - Improves cache, TLB performance
- Dead code: similar to unreachable blocks.
  - A value might be computed but never subsequently used.
- Code for computing the value can be dropped
- But only if it's **pure**, i.e., it has *no externally visible side effects*
  - Externally visible effects: raising an exception, modifying a global variable, going into an infinite loop, printing to standard output, sending a network packet, launching a rocket
  - Note: Pure functional languages (e.g., Haskell) make reasoning about the safety of optimizations (and code transformations in general) easier!

# Inlining

- Replace a call to a function with the body of the function itself with arguments rewritten to be local variables:
- Example in C: inline `pow` into `g`

```
int g(int x) { return x + pow(x); }  
int pow(int a) {  
    var b = 1; var x = 0;  
    while (x < a) {b = 2 * b; x = x + 1}  
    return b;  
}
```



```
int g(int x) {  
    int a = x;  
    int b = 1; int x2 = 0;  
    while (x2 < a) {b = 2 * b; x2 = x2 + 1};  
    tmp = b;  
    return x + tmp;  
}
```

note: renaming

- May need to rename variables to avoid *capture*
- Best done at the AST or relatively high-level IR.
- When is it profitable?
  - Eliminates the stack manipulation, jump, etc.
  - Can increase code size.
  - Enables further optimizations



# Code Specialization

- Idea: create specialized versions of a function that is called from different places with different arguments.
- Example: specialize function `f` in:

```
class A implements I { int m() {...} }  
class B implements I { int m() {...} }  
int f(I x) { x.m(); }           // don't know which m  
A a = new A(); f(a);           // know it's A.m  
B b = new B(); f(b);           // know it's B.m
```

- `f_A` would have code specialized to dispatch to `A.m`
- `f_B` would have code specialized to dispatch to `B.m`
- You can also inline methods when the run-time type is known statically
  - Often just one class implements a method.

# Common Subexpression Elimination

- *fold redundant computations together*
  - in some sense, it's the opposite of inlining
- Example:

```
a[i] = a[i] + 1
```

compiles to:

```
[a + i*4] = [a + i*4] + 1
```

Common subexpression elimination removes the redundant add and multiply:

```
t = a + i*4; [t] = [t] + 1
```

- For safety, you must be sure that the shared expression always has the same value in both places!



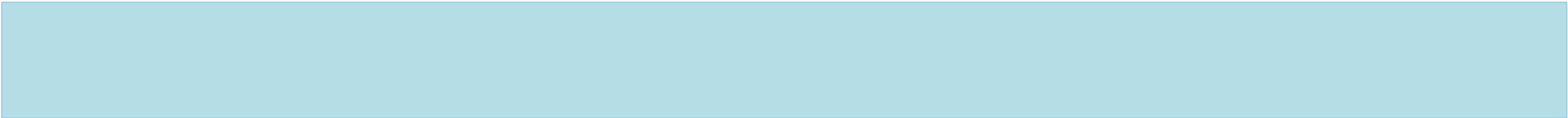
# Unsafe Common Subexpression Elimination

- Example: consider this C function:

```
unit f(int[] a, int[] b, int[] c) {  
    var j = ...; var i = ...; var k = ...;  
    b[j] = a[i] + 1;  
    c[k] = a[i];  
    return;  
}
```

- The optimization that shares the expression `a[i]` is unsafe... why?

```
unit f(int[] a, int[] b, int[] c) {  
    var j = ...; var i = ...; var k = ...;  
    t = a[i];  
    b[j] = t + 1;  
    c[k] = t;  
    return;  
}
```



# LOOP OPTIMIZATIONS

# Loop Optimizations

- Program hot spots often occur in loops.
  - Especially inner loops
  - Not always: consider operating systems code or compilers vs. a computer game or word processor
- Most program execution time occurs in loops.
  - The 90/10 rule of thumb holds here too.  
(90% of the execution time is spent in 10% of the code)
- Loop optimizations are very important, effective, and numerous
  - Also, concentrating effort to improve loop body code is usually a win



# Loop Invariant Code Motion (revisited)

- Another form of redundancy elimination.
- If the result of a statement or expression does not change during the loop *and* it's pure, it can be hoisted outside the loop body.
- Often useful for array element addressing code
  - Invariant code not visible at the source level

```
for (i = 0; i < a.length; i++) {  
    /* a not modified in the body */  
}
```



```
t = a.length;  
for (i = 0; i < t; i++) {  
    /* same body as above */  
}
```

Hoisted loop-  
invariant  
expression

# Strength Reduction (revisited)

- Strength reduction can work for loops too
- Idea: replace expensive operations (multiplies, divides) by cheap ones (adds and subtracts)
- For loops, create a *dependent induction variable*:
- Example:

```
for (int i = 0; i < n; i++) { a[i*3] = 1; } // stride by 3
```



```
int j = 0;  
for (int i = 0; i < n; i++) {  
    a[j] = 1;  
    j = j + 3; // replace multiply by add  
}
```

# Loop Unrolling (revisited)

- Branches can be expensive, unroll loops to avoid them.

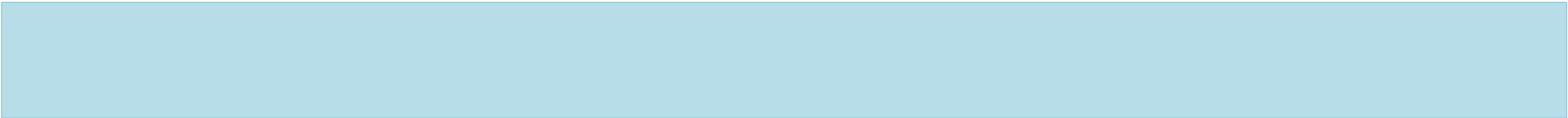
```
for (int i=0; i<n; i++) { S }
```



```
for (int i=0; i<n-3; i+=4) {S;S;S;S};  
for (      ; i<n; i++) { S } // left over iterations
```

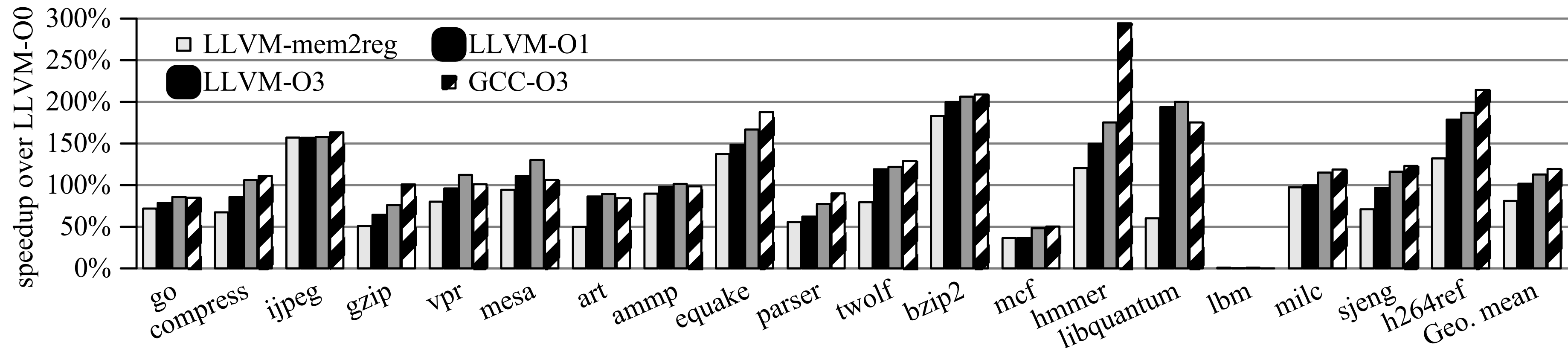
- With k unrollings, eliminates  $(k-1)/k$  conditional branches
  - So for the above program, it eliminates  $3/4$  of the branches
- Space-time tradeoff:
  - Not a good idea for large S or small n
- Interacts with instruction caching, branch prediction





**EFFECTIVENESS?**

# Optimization Effectiveness?



$$\% \text{speedup} = \left[ \frac{\text{base time}}{\text{optimized time}} - 1 \right] \times 100\%$$

Example:

base time = 2s

optimized time = 1s

⇒ 100% speedup

Example:

base time = 1.2s

optimized time = 0.87s

⇒ 38% speedup

Graph taken from:

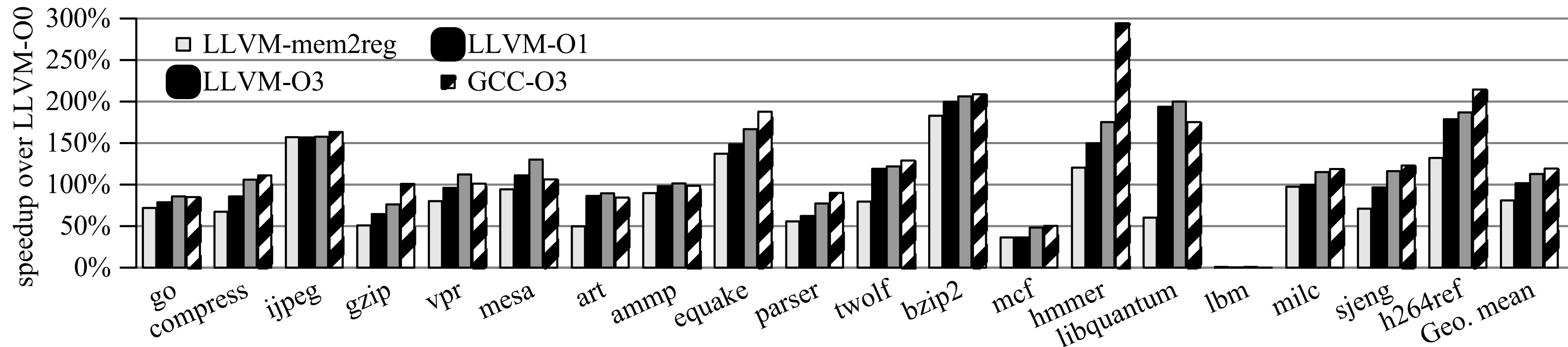
Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic.

Formal Verification of SSA-Based Optimizations for LLVM.

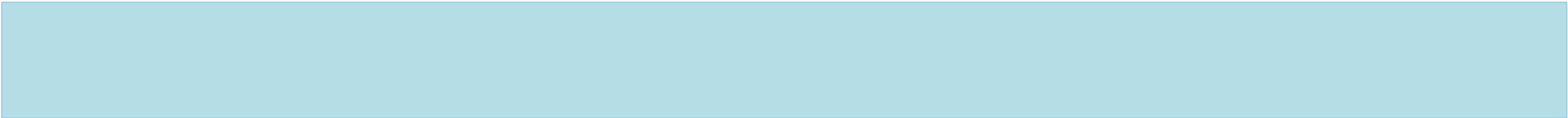
In Proc. 2013 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI), 2013



# Optimization Effectiveness?



- mem2reg: promotes alloca'd stack slots to temporaries to enable register allocation
- Analysis:
  - mem2reg alone (+ back-end optimizations like register allocation) yields ~78% speedup on average
  - -O1 yields ~100% speedup (so all the rest of the optimizations combined account for ~22%)
  - -O3 yields ~120% speedup
- Hypothetical program that takes 10 sec. (base time):
  - Mem2reg alone: expect ~5.6 sec
  - -O1: expect ~5 sec
  - -O3: expect ~4.5 sec



# CODE ANALYSIS

# Assertion Removal

- Dynamic typing adds many runtime assertions into our program.
- ```
let x = f() in
let y = x + 2 in
let z = y * x in
...
```
- Current compilation always adds assertions that inputs are integers
- ```
x = f()
assertInt(x)
y = x + 2
assertInt(y)
assertInt(x)
y2 = y >> 1
z = y2 * x
...
```
- Which assertions can we remove?



# Assertion Removal

- Dynamic typing adds many runtime assertions into our program.
- ```
let x = f() in
let y = x + 2 in
let z = y * x in
...
```
- Current compilation always adds assertions that inputs are integers
- ```
x = f()
assertInt(x)
y = x + 2
assertInt(y)
assertInt(x)
y2 = y >> 1
z = y2 * x
...
```
- **Which assertions can we remove?**

# Assertion Removal

- When is it correct to remove an assertion from our SSA program?

```
...  
assertInt(x)  
...
```

When we are **sure** that the assertion will succeed

In this case, if we are **sure** that  $x$  can only ever be a (tagged) integer at runtime.

- Appropriate analysis: determine what possible values  $x$  can take at runtime.

# Possible Values Analysis

- To perform assertion removal, we need to figure out what possible values variables take at runtime.
- Perform an analysis that says at every program point, the set of possible values that every variable might have at that point in the program.
- Remove assertions that always would succeed on the possible values
- Rice's theorem applies: it's impossible to compute the exact correct sets. So we must approximate.

Which way should we approximate?

- Underapproximate: produce a **subset** of the true possible values. But might miss some
- Overapproximate: produce a **superset** of the true possible values. But might include some that never happen
- For assertion removal, we need to **overapproximate**.
  - If our set is a superset of the true possible values, and still contains only tagged integers, then at runtime the possible value is definitely a tagged integer.
  - Might miss out on some assertion removals, but that's unavoidable.



# Possible Values Analysis

- What do we mean by "possible values"?
- Performing this analysis at the SSA level. In SSA, a value is a 64-bit integer.
- So we can represent a set of possible SSA values as a `HashSet<i64>` in Rust.

```
x = f()  
assertInt(x)  
y = x + 2  
assertInt(y)  
assertInt(x)  
y2 = y >> 1  
z = y2 * x
```

Problem: after `x = f()`, assuming `f` is an extern function, `x` may take on any value. That would be a huge set.

In general, a set of `i64` values would take  $2^{64}$  bits to represent!

# Abstract Interpretation

- To keep space manageable, we need a different representation of sets, one that takes much less space than  $(2^{64})$  bits.
- This **inherently** means we are missing out on precision! But most of those sets are never going to come up in our analysis anyway.
- We design an "abstract domain" of possible value sets that is good enough to perform our analysis.
- To start, let's just worry about removing assertInt.
  - A simple abstract domain is to have just three elements:
    - Any (aka Top): this represents the set of all possible 64-bit integers
    - Even (aka TaggedInt): this represents the even 64-bit integers
    - None aka Empty aka Bottom: the represents the empty set



# Flow Functions

- At each program point, for each variable associate an approximation of what the possible values are.
- For each instruction, define a "flow function" that says how the possible values are affected by performing the operation

# Flow Functions

- At each program point, for each variable associate an approximation of what the possible values are.
- For each instruction, define a "flow function" that says how the possible values are affected by performing the operation

$x = y + z$

What is the **most precise information** we know about the possible values of  $x$  based on the possible values of  $y$  and  $z$ ?

$\text{Poss}(x) = \text{Flow}[+](\text{Poss}(y), \text{Poss}(z)) \sim \sim \{ y + z \mid y \text{ in } \text{Poss}(y), z \text{ in } \text{Poss}(z) \}$

$\text{Flow}[+](\text{Any}, \text{Any}) = \text{Any}$

$\text{Flow}[+](\text{Any}, \text{Even}) = \text{Flow}[+](\text{Even}, \text{Any}) = \text{Any}$

$\text{Flow}[+](\text{Even}, \text{Even}) = \text{Even}$

$\text{Flow}[+](\text{None}, Q) = \text{None}$

$\text{Flow}[+](P, \text{None}) = \text{None}$

- Why is this correct? We output the most precise approximation of the set of all values that result from adding values in the input sets.
- $\text{None} + Q = \{ y + z \mid y \text{ in } \text{EmptySet}, z \text{ in } Q \} = \text{EmptySet}$

# Flow Functions

- At each program point, for each variable associate an approximation of what the possible values are.
- For each instruction, define a "flow function" that says how the possible values are affected by performing the operation

$x = y \ll n$  where  $n \geq 1$

$\text{Poss}(x) = \text{Flow}[\ll n](\text{Poss}(y)) \sim \sim \{ y \ll n \mid y \text{ in } \text{Poss}(y) \}$

$\text{Flow}[\ll n](\text{Any}) = \text{Even}$

$\text{Flow}[\ll n](\text{Even}) = \text{Even}$

$\text{Flow}[\ll n](\text{None}) = \text{None}$

- Note here that the case for Even **loses** precision:
  - $\{ y \ll 1 \mid y \text{ in } \text{Even} \} = \text{Multiples of 4 subset Even}$



# Flow Functions

- At each program point, for each variable associate an approximation of what the possible values are.
- For each instruction, define a "flow function" that says how the possible values are affected by performing the operation

$x = y * z$

What is the **most precise information** we know about the possible values of  $x$  based on the possible values of  $y$  and  $z$ ?

$\text{Poss}(x) = \text{Flow}[*](\text{Poss}(y), \text{Poss}(z)) \sim \sim \{ y * z \mid y \text{ in } \text{Poss}(y), z \text{ in } \text{Poss}(z) \}$

$\text{Flow}[*](\text{Any}, \text{Any}) = \text{Any}$

$\text{Flow}[*](\text{Any}, \text{Even}) = \text{Flow}[*](\text{Even}, \text{Any}) = \text{Even}$

$\text{Flow}[*](\text{Even}, \text{Even}) = \text{Even}$

$\text{Flow}[*](\text{None}, Q) = \text{None}$

$\text{Flow}[*](P, \text{None}) = \text{None}$

# Flow Functions

- At each program point, for each variable associate an approximation of what the possible values are.
- For each instruction, define a "flow function" that says how the possible values are affected by performing the operation

`assertInt(x)`

`Poss(x) = Flow[assertInt](Poss(x))`

`Flow[assertInt](Any) = Even`

`Flow[assertInt](Even) = Even`

`Flow[assertInt](None) = None`

# Straightline Code Example

```
0 x = f()  
1 assertInt(x)  
2 y = x + 2  
3 assertInt(y)  
4 assertInt(x)  
5 y2 = y >> 1  
6 z = y2 * x  
7
```

```
0:  
1: {x: Any}  
2: {x: Even}  
3: {x: Even, y: Even}  
4: {x: Even, y: Even}  
5: {x: Even, y: Even}  
6: {x: Even, y: Even, y2: Any}  
7: {x: Even, y: Even, y2: Any, z: Even}
```



# Straightline Code Example

```
0 x = f()  
1 assertInt(x)  
2 y = x + 2  
3 assertInt(y)  
4 assertInt(x)  
5 y2 = y >> 1  
6 z = y2 * x  
7
```

```
0:  
1: {x: Any}  
2: {x: Even}  
3: {x: Even, y: Even}  
4: {x: Even, y: Even}  
5: {x: Even, y: Even}  
6: {x: Even, y: Even, y2: Any}  
7: {x: Even, y: Even, y2: Any, z: Even}
```

# Tag-checking Analysis

- At each program point, for each variable associate an approximation of what the possible values are.
- For each instruction, update that information accordingly  
To do a complete analysis: extend this to all SSA operations
- What about **blocks** and **functions**?

```
f(x):  
    assertInt(x)  
    assertInt(y)  
    ...
```

What info do we have about x? about y?

Collect the info from all the places that **branch to f**, taking a "union"

We call these the **predecessors of f**, because they are the incoming edges of the control-flow graph.

Because we can have loops, **f** can be a predecessor of itself, so we have a similar circularity that we did in liveness.

Same solution: initialize the information to be minimal (bottom in this case) and update iteratively

For functions, the predecessors are places that **call f**.

For main, there is a special implicit predecessor which is the entry point. This sets the input variable to Any because the program input is an array.



# Loop Example

```
extern g
def main(y):
    def loop(i,a):
        if i == 0:
            a
        else:
            loop(i - 1, a + g())
    in
    loop(y, 0)
```

```
main(y):
    loop(i,a):
        thn():
            ret a
        els():
            assertInt(i)
            i' = i - 2
            x = g()
            assertInt(a)
            assertInt(x)
            a' = a + x
            br loop(i', a')
        b = i == 0
        cbr b thn() els()
    br loop(y, 0)
```

```

main(y):
  loop(i,a):
    thn():
      3 ret a
    els():
      4 assertInt(i)
      5 i' = i - 2
      6 x = g()
      7 assertInt(a)
      8 assertInt(x)
      9 a' = a + x
      10 br loop(i', a')
  1 b = i == 0
  2 cbr b thn() els()
  0 br loop(y, 0)

```

Initialize the blocks: main entry point arguments are Any, other blocks everything is None.

```

0: {y:Any}
1: {y,i,a:None}
2:
3: {y,i,a:None}
4: {y,i,a:None}
5:
6:
7:
8:
9:
10:

```

```

main(y):
  loop(i,a):
    thn():
      3 ret a
    els():
      4 assertInt(i)
      5 i' = i - 2
      6 x = g()
      7 assertInt(a)
      8 assertInt(x)
      9 a' = a + x
      10 br loop(i', a')
  1 b = i == 0
  2 cbr b thn() els()
  0 br loop(y, 0)

```

Next: apply the flow functions to update the internal nodes

```

0: {y:Any}
1: {y,i,a:None}
2: {y,i,a,b:None}
3: {y,i,a:None}
4: {y,i,a:None}
5: {y,i,a:None}
6: {y,i,a,i':None}
7: {y,i,a,i':None,x:Any}
8: {y,i,a,i':None,x:Any}
9: {y,i,a,i':None,x:Even}
10: {y,i,a,i',a':None,x:Even}

```



```
main(y):
  loop(i,a):
    thn():
      3 ret a
    els():
      4 assertInt(i)
      5 i' = i - 2
      6 x = g()
      7 assertInt(a)
      8 assertInt(x)
      9 a' = a + x
      10 br loop(i', a')
  1 b = i == 0
  2 cbr b thn() els()
  0 br loop(y, 0)
```

To start a new iteration, initialize blocks based on the previous round's info about predecessors

| Previous Round                | Next Round |
|-------------------------------|------------|
| 0: {y:Any}                    | 0: ?       |
| 1: {y,i,a:None}               | 1: ?       |
| 2: {y,i,a,b:None}             | 2:         |
| 3: {y,i,a:None}               | 3: ?       |
| 4: {y,i,a:None}               | 4: ?       |
| 5: {y,i,a:None}               | 5:         |
| 6: {y,i,a,i':None}            | 6:         |
| 7: {y,i,a,i':None,x:Any}      | 7:         |
| 8: {y,i,a,i':None,x:Any}      | 8:         |
| 9: {y,i,a,i':None,x:Even}     | 9:         |
| 10: {y,i,a,i',a':None,x:Even} | 10:        |

```
main(y):
  loop(i,a):
    thn():
      3 ret a
    els():
      4 assertInt(i)
      5 i' = i - 2
      6 x = g()
      7 assertInt(a)
      8 assertInt(x)
      9 a' = a + x
      10 br loop(i', a')
  1 b = i == 0
  2 cbr b thn() els()
  0 br loop(y, 0)
```

To start a new iteration, initialize blocks based on the previous round's info about predecessors

Previous Round

- 0: {y:Any}
- 1: {y,i,a:None}
- 2: {y,i,a,b:None}
- 3: {y,i,a:None}
- 4: {y,i,a:None}
- 5: {y,i,a:None}
- 6: {y,i,a,i':None}
- 7: {y,i,a,i':None,x:Any}
- 8: {y,i,a,i':None,x:Any}
- 9: {y,i,a,i':None,x:Even}
- 10: {y,i,a,i',a':None,x:Even}

Next Round

- 0: {y:Any}
- 1: {y:Any U None, i:Any U None, a: Even U None}
- 2:
- 3: ?
- 4: ?
- 5:
- 6:
- 7:
- 8:
- 9:
- 10:

the loop(i,a) body 1 has two predecessors:  
br loop(y, 0)  
br loop(i', a')  
Take the "union" of their information

```

main(y):
  loop(i,a):
    thn():
      3 ret a
    els():
      4 assertInt(i)
      5 i' = i - 2
      6 x = g()
      7 assertInt(a)
      8 assertInt(x)
      9 a' = a + x
      10 br loop(i', a')
  1 b = i == 0
  2 cbr b thn() els()
  0 br loop(y, 0)

```

To start a new iteration, initialize blocks based on the previous round's info about predecessors

Previous Round

Next Round

|                               |                         |
|-------------------------------|-------------------------|
| 0: {y:Any}                    | 0: {y:Any}              |
| 1: {y,i,a:None}               | 1: {y:Any,i:Any,a:Even} |
| 2: {y,i,a,b:None}             | 2:                      |
| 3: {y,i,a:None}               | 3: ?                    |
| 4: {y,i,a:None}               | 4: ?                    |
| 5: {y,i,a:None}               | 5:                      |
| 6: {y,i,a,i':None}            | 6:                      |
| 7: {y,i,a,i':None,x:Any}      | 7:                      |
| 8: {y,i,a,i':None,x:Any}      | 8:                      |
| 9: {y,i,a,i':None,x:Even}     | 9:                      |
| 10: {y,i,a,i',a':None,x:Even} | 10:                     |

the loop(i,a) body 1 has two predecessors:

br loop(y, 0)

br loop(i', a')

Take the "union" of their information



```

main(y):
  loop(i,a):
    thn():
      3 ret a
    els():
      4 assertInt(i)
      5 i' = i - 2
      6 x = g()
      7 assertInt(a)
      8 assertInt(x)
      9 a' = a + x
      10 br loop(i', a')
  1 b = i == 0
  2 cbr b thn() els()
  0 br loop(y, 0)

```

To start a new iteration, initialize blocks based on the previous round's info about predecessors

Previous Round

```

0: {y:Any}
1: {y,i,a:None}
2: {y,i,a,b:None}
3: {y,i,a:None}
4: {y,i,a:None}
5: {y,i,a:None}
6: {y,i,a,i':None}
7: {y,i,a,i':None,x:Any}
8: {y,i,a,i':None,x:Any}
9: {y,i,a,i':None,x:Even}
10: {y,i,a,i',a':None,x:Even}

```

Next Round

```

0: {y:Any}
1: {y:Any,i:Any,a:Even}
2:
3: {y,i,a,b:None}
4: {y,i,a,b:None}
5:
6:
7:
8:
9:
10:

```

thn() and els() each have one predecessor  
 cbr b thn() els()

```

main(y):
  loop(i,a):
    thn():
      3 ret a
    els():
      4 assertInt(i)
      5 i' = i - 2
      6 x = g()
      7 assertInt(a)
      8 assertInt(x)
      9 a' = a + x
      10 br loop(i', a')
  1 b = i == 0
  2 cbr b thn() els()
  0 br loop(y, 0)

```

Update the rest of the internal nodes

Current Round

```

0: {y:Any}
1: {y:Any,i:Any,a:Even}
2:
3: {y,i,a,b:None}
4: {y,i,a,b:None}
5:
6:
7:
8:
9:
10:

```



```

main(y):
  loop(i,a):
    thn():
      3 ret a
    els():
      4 assertInt(i)
      5 i' = i - 2
      6 x = g()
      7 assertInt(a)
      8 assertInt(x)
      9 a' = a + x
      10 br loop(i', a')
  1 b = i == 0
  2 cbr b thn() els()
  0 br loop(y, 0)

```

Update the rest of the internal nodes

Current Round

```

0: {y:Any}
1: {y:Any,i:Any,a:Even}
2: {y:Any,i:Any,a:Even,b:Any}
3: {y,i,a,b:None}
4: {y,i,a,b:None}
5: {y,i,a:None}
6: {y,i,a,i':None}
7: {y,i,a,i':None,x:Any}
8: {y,i,a,i':None,x:Any}
9: {y,i,a,i':None,x:Even}
10: {y,i,a,i',a':None,x:Even}

```

Since the results changed, we perform another iteration

|                                |                                           |                                     |
|--------------------------------|-------------------------------------------|-------------------------------------|
| main(y):                       |                                           |                                     |
| loop(i,a):                     |                                           |                                     |
| thn():                         | Previous Round                            | Next Round                          |
| <sup>3</sup> ret a             | <sup>0</sup> : {y:Any}                    | <sup>0</sup> : {y:Any}              |
| els():                         | <sup>1</sup> : {y:Any,i:Any,a:Even}       | <sup>1</sup> : {y:Any,i:Any,a:Even} |
| <sup>4</sup> assertInt(i)      | <sup>2</sup> : {y:Any,i:Any,a:Even,b:Any} | <sup>2</sup> :                      |
| <sup>5</sup> i' = i - 2        | <sup>3</sup> : {y,i,a,b:None}             | <sup>3</sup> : ?                    |
| <sup>6</sup> x = g()           | <sup>4</sup> : {y,i,a,b:None}             | <sup>4</sup> : ?                    |
| <sup>7</sup> assertInt(a)      | <sup>5</sup> : {y,i,a:None}               | <sup>5</sup> :                      |
| <sup>8</sup> assertInt(x)      | <sup>6</sup> : {y,i,a,i':None}            | <sup>6</sup> :                      |
| <sup>9</sup> a' = a + x        | <sup>7</sup> : {y,i,a,i':None,x:Any}      | <sup>7</sup> :                      |
| <sup>10</sup> br loop(i', a')  | <sup>8</sup> : {y,i,a,i':None,x:Any}      | <sup>8</sup> :                      |
| <sup>1</sup> b = i == 0        | <sup>9</sup> : {y,i,a,i':None,x:Even}     | <sup>9</sup> :                      |
| <sup>2</sup> cbr b thn() els() | <sup>10</sup> : {y,i,a,i',a':None,x:Even} | <sup>10</sup> :                     |
| <sup>0</sup> br loop(y, 0)     |                                           |                                     |

the loop(i,a) body 1 has two predecessors:

  br loop(y, 0)

  br loop(i', a')

Take the "union" of their information

```
main(y):
  loop(i,a):
    thn():
      3 ret a
    els():
      4 assertInt(i)
      5 i' = i - 2
      6 x = g()
      7 assertInt(a)
      8 assertInt(x)
      9 a' = a + x
      10 br loop(i', a')
  1 b = i == 0
  2 cbr b thn() els()
  0 br loop(y, 0)
```

Previous Round

- 0: {y:Any}
- 1: {y:Any,i:Any,a:Even}
- 2: {y:Any,i:Any,a:Even,b:Any}
- 3: {y,i,a,b:None}
- 4: {y,i,a,b:None}
- 5: {y,i,a:None}
- 6: {y,i,a,i':None}
- 7: {y,i,a,i':None,x:Any}
- 8: {y,i,a,i':None,x:Any}
- 9: {y,i,a,i':None,x:Even}
- 10: {y,i,a,i',a':None,x:Even}

Next Round

- 0: {y:Any}
- 1: {y:Any,i:Any,a:Even}
- 2:
- 3: {y:Any,i:Any,a:Even,b:Any}
- 4: {y:Any,i:Any,a:Even,b:Any}
- 5:
- 6:
- 7:
- 8:
- 9:
- 10:

thn() and els() each have one predecessor  
cbr b thn() els()



```

main(y):
  loop(i,a):
    thn():
      3 ret a
    els():
      4 assertInt(i)
      5 i' = i - 2
      6 x = g()
      7 assertInt(a)
      8 assertInt(x)
      9 a' = a + x
      10 br loop(i', a')
      1 b = i == 0
      2 cbr b thn() els()
      0 br loop(y, 0)

```

Update the rest of the internal nodes

Current Round

```

0: {y:Any}
1: {y:Any,i:Any,a:Even}
2:
3: {y:Any,i:Any,a:Even,b:Any}
4: {y:Any,i:Any,a:Even,b:Any}
5:
6:
7:
8:
9:
10:

```

```

main(y):
  loop(i,a):
    thn():
      3 ret a
    els():
      4 assertInt(i)
      5 i' = i - 2
      6 x = g()
      7 assertInt(a)
      8 assertInt(x)
      9 a' = a + x
      10 br loop(i', a')
  1 b = i == 0
  2 cbr b thn() els()
  0 br loop(y, 0)

```

Update the rest of the internal nodes

Current Round

```

0: {y:Any}
1: {y:Any,i:Any,a:Even}
2: {y:Any,i:Any,a:Even,b:Any}
3: {y:Any,i:Any,a:Even,b:Any}
4: {y:Any,i:Any,a:Even,b:Any}
5: {y:Any,i:Even,a:Even,b:Any}
6: {y:Any,i:Even,a:Even,b:Any,i':Even}
7: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Any}
8: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Any}
9: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Int}
10: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Int,a':Int}

```

If we repeat one more iteration, we get the same result, and we have our final analysis

```

main(y):
  loop(i,a):
    thn():
      3 ret a
    els():
      4 assertInt(i)
      5 i' = i - 2
      6 x = g()
      7 assertInt(a)
      8 assertInt(x)
      9 a' = a + x
      10 br loop(i', a')
  1 b = i == 0
  2 cbr b thn() els()
  0 br loop(y, 0)

```

Update the rest of the internal nodes

Current Round

```

0: {y:Any}
1: {y:Any,i:Any,a:Even}
2: {y:Any,i:Any,a:Even,b:Any}
3: {y:Any,i:Any,a:Even,b:Any}
4: {y:Any,i:Any,a:Even,b:Any}
5: {y:Any,i:Even,a:Even,b:Any}
6: {y:Any,i:Even,a:Even,b:Any,i':Even}
7: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Any}
8: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Any}
9: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Int}
10: {y:Any,i:Even,a:Even,b:Any,i':Even,x:Int,a':Int}

```

With all of this work, we can remove 1 assertion: assertInt(a)

We can't prove that assertInt(i) succeeds because the initial value of y might not be an integer...

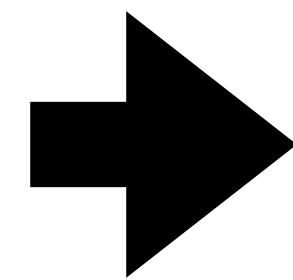


```

extern g
def main(y):
    def loop(i,a):
        if i == 0:
            a - z
        else:
            loop(i - 1, a + g())
    in
    loop(y, 0)

```

inline once



```

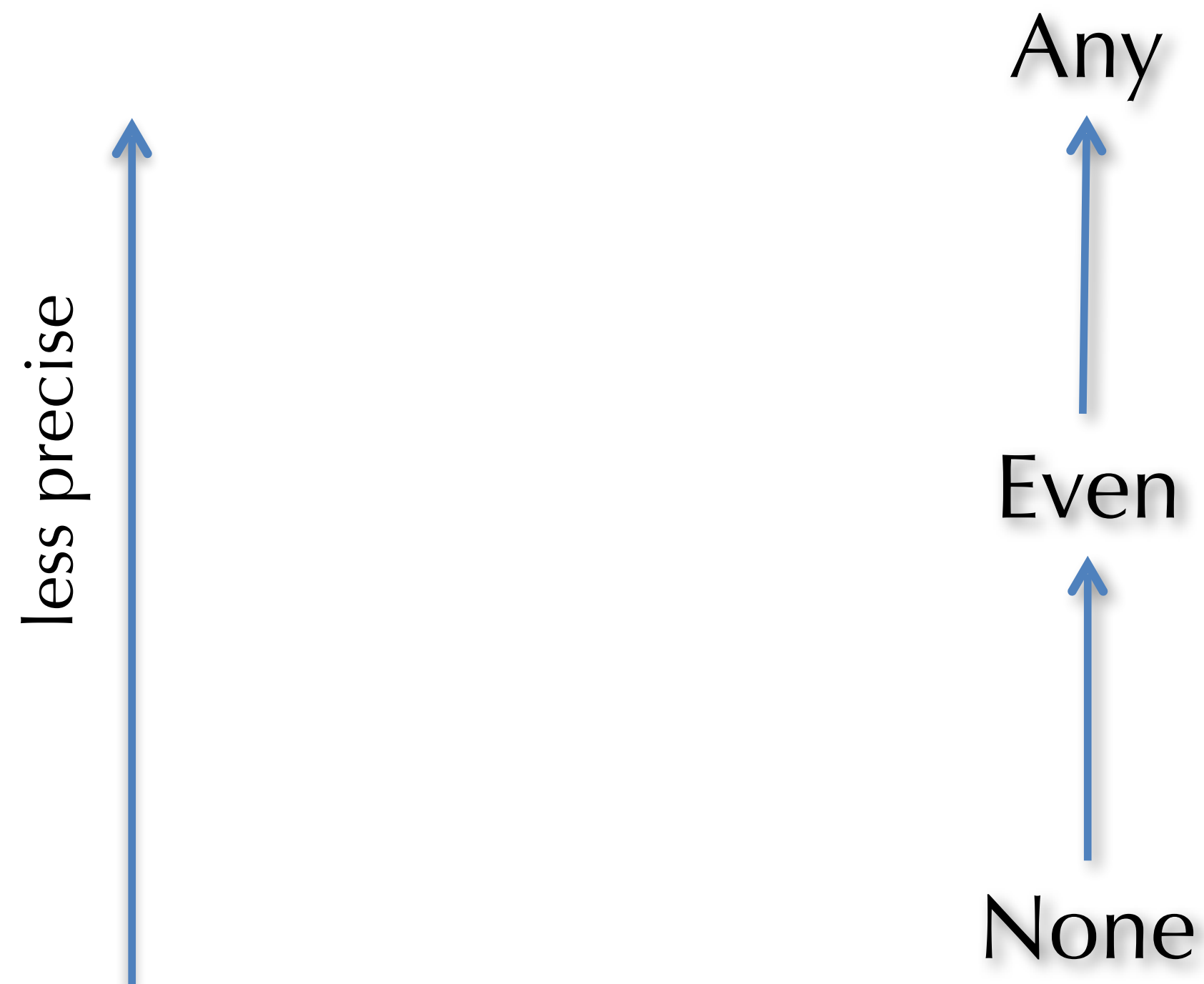
extern g
def main(y):
    def loop(i,a):
        if i == 0:
            a
        else:
            loop(i - 1, a + g())
    in
    if y == 0:
        0
    else:
        loop(y - 1, 0 + g())

```

If we re-do the analysis, no i is always an Int

# Abstract Interpretation

- We used a simple interpretation for just removing assertInt
- A simple abstract domain is to have just three elements:
  - Any (aka Top): this represents the set of all possible 64-bit integers
  - Even (aka TaggedInt): this represents the even 64-bit integers
  - None aka Empty aka Bottom: this represents the empty set



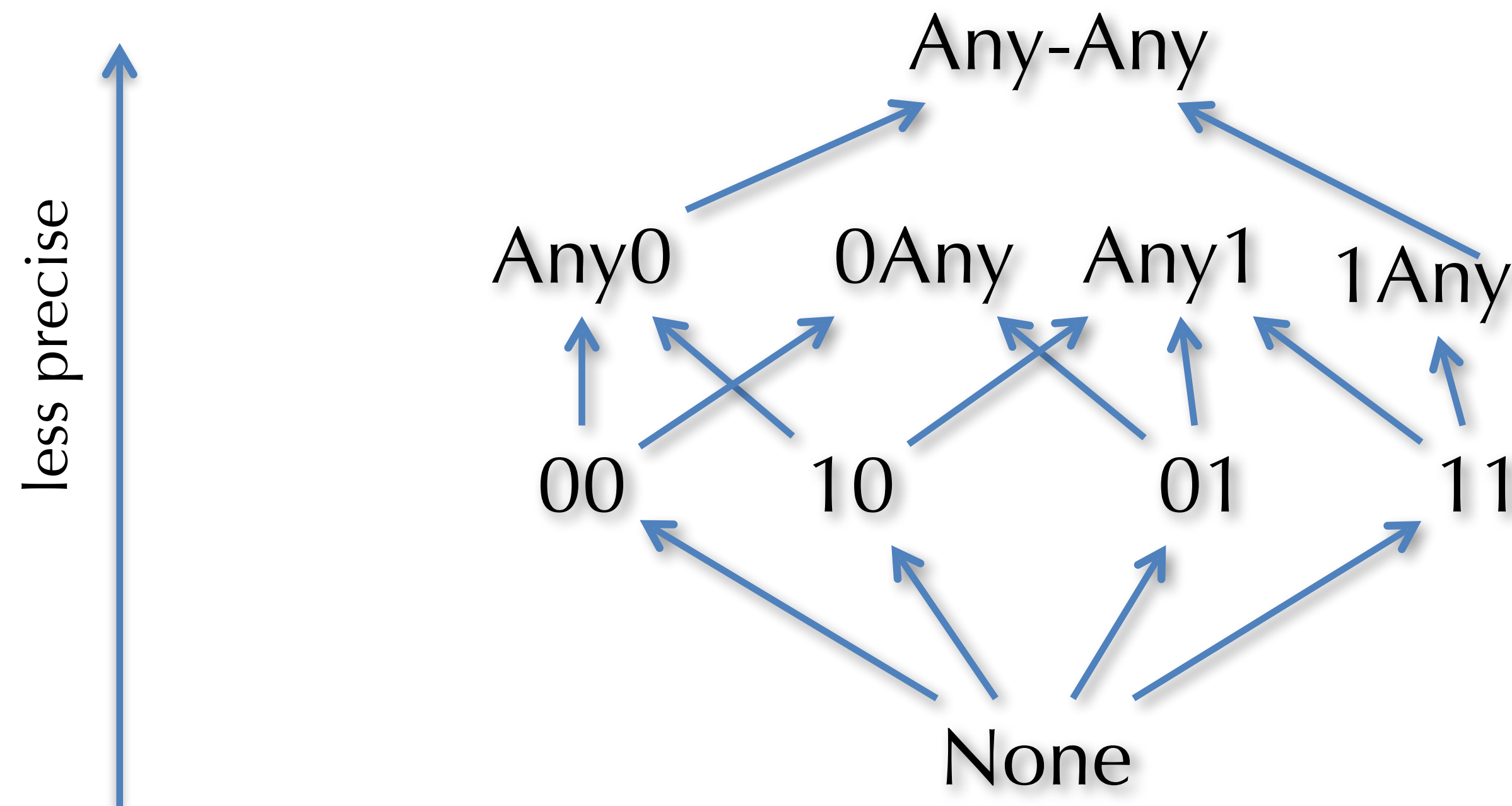


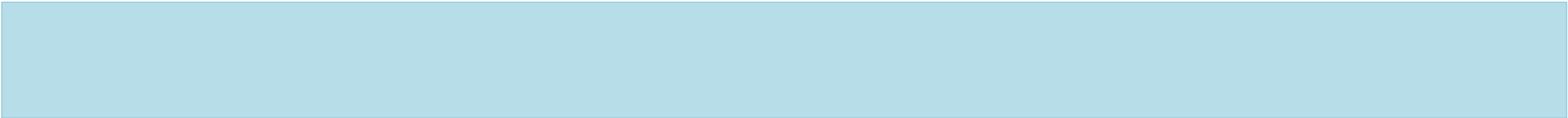
# Abstract Interpretation

- We used a simple interpretation for just removing assertInt
  - What about for Booleans and Arrays?
  - Abstract domain for the lowest two bits:

# Abstract Interpretation

- We used a simple interpretation for just removing assertInt
  - What about for Booleans and Arrays?
  - Abstract domain for the lowest two bits:





# GENERAL DATAFLOW ANALYSIS



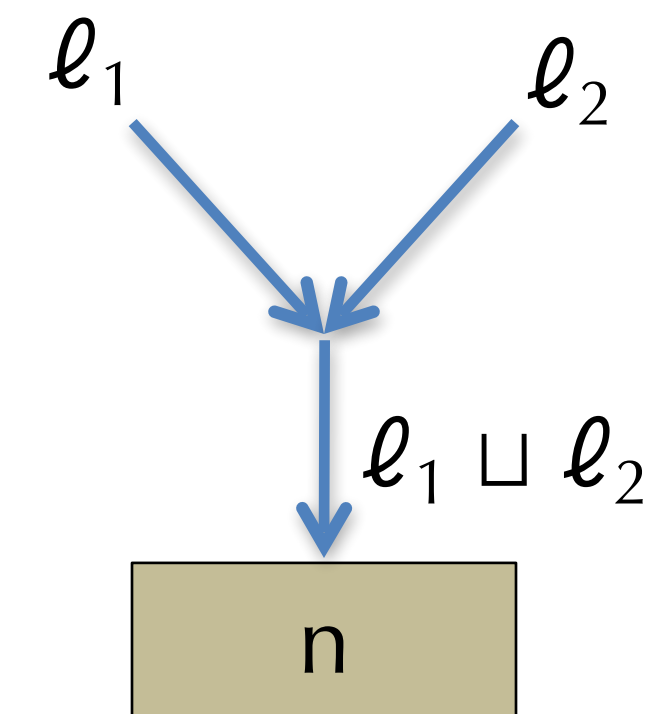
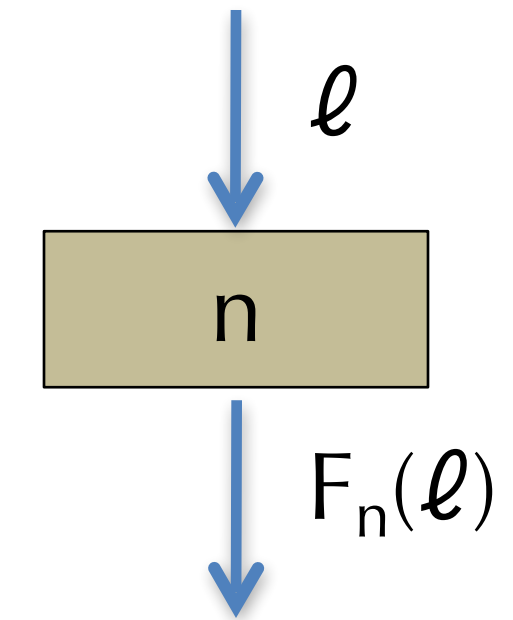
# Common Features

- Liveness and Possible Values Analyses had similarities
  - In both, we have some **domain** of information we attach to each point in the program
    - Liveness, the domain is sets of variables
    - Possible values, the domain is maps from variables to (abstractions of sets of values)
  - Each analysis has a notion of **flow function**
    - How do we update the information based on each operation in the program.
  - But they propagate information in opposite directions
    - Liveness is Backwards: if I use a variable now, it is live in previous program points
    - Possible values is Forwards: if I learn a variables value now, I know it later as well
  - Each analysis aggregates information at control flow boundaries
    - Liveness takes the union of successors at a conditional branch
    - Possible values takes the union of predecessors at a block/function
  - Perform the analysis by starting from incorrect information and iterating until we get the same result, a fixed point.

# (Forward) Dataflow Analysis Framework

A forward dataflow analysis can be characterized by:

1. A domain of dataflow values  $\mathcal{L}$ 
  - e.g.  $\mathcal{L}$  = the powerset of all variables
  - Think of  $\ell \in \mathcal{L}$  as a property, then “ $x \in \ell$ ” means “ $x$  has the property”
2. For each node  $n$ , a flow function  $F_n : \mathcal{L} \rightarrow \mathcal{L}$ 
  - “If  $\ell$  is a property that holds before the node  $n$ , then  $F_n(\ell)$  holds after  $n$ ”
3. A combining operator  $\sqcup$ 
  - “If we know *either*  $\ell_1$  or  $\ell_2$  holds on entry to node  $n$ , we know at most  $\ell_1 \sqcup \ell_2$ ”
  - $\text{in}[n] := \bigsqcup_{n' \in \text{pred}[n]} \text{out}[n']$



# Generic Iterative (Forward) Analysis

for all  $n$ ,  $\text{in}[n] := \perp$ ,  $\text{out}[n] := \perp$

repeat until no change

  for all  $n$

$\text{in}[n] := \bigsqcup_{n' \in \text{pred}[n]} \text{out}[n']$

$\text{out}[n] := F_n(\text{in}[n])$

  end

end

- Here,  $\perp \in \mathcal{L}$  (“bottom”) represents having the “most precise” constraint
  - Having “more precise” information enables more optimizations
  - “most precise” amount could be inconsistent with the constraints.
  - Iteration refines the answer, eliminating inconsistencies, producing less precise results



# Structure of $\mathcal{L}$

- The domain has structure that reflects the “amount” of information contained in each dataflow value.
- Some dataflow values are more informative than others:
  - Write  $\ell_1 \sqsubseteq \ell_2$  whenever  $\ell_1$  provides at least as much information as  $\ell_2$ .
  - The dataflow value  $\ell_1$  is “better” for enabling optimizations.
- Example 1: for liveness and possible values analysis, *smaller* sets of variables are more informative.
  - Having smaller sets of variables live across an edge means that there are fewer conflicts for register allocation assignments.
  - So:  $\ell_1 \sqsubseteq \ell_2$  if and only if  $\ell_1 \subseteq \ell_2$
- Example 2: for available expressions analysis, larger sets of nodes are more informative.
  - Having a larger set of nodes (equivalently, expressions) available means that there is more opportunity for common subexpression elimination.
  - So:  $\ell_1 \sqsubseteq \ell_2$  if and only if  $\ell_1 \supseteq \ell_2$

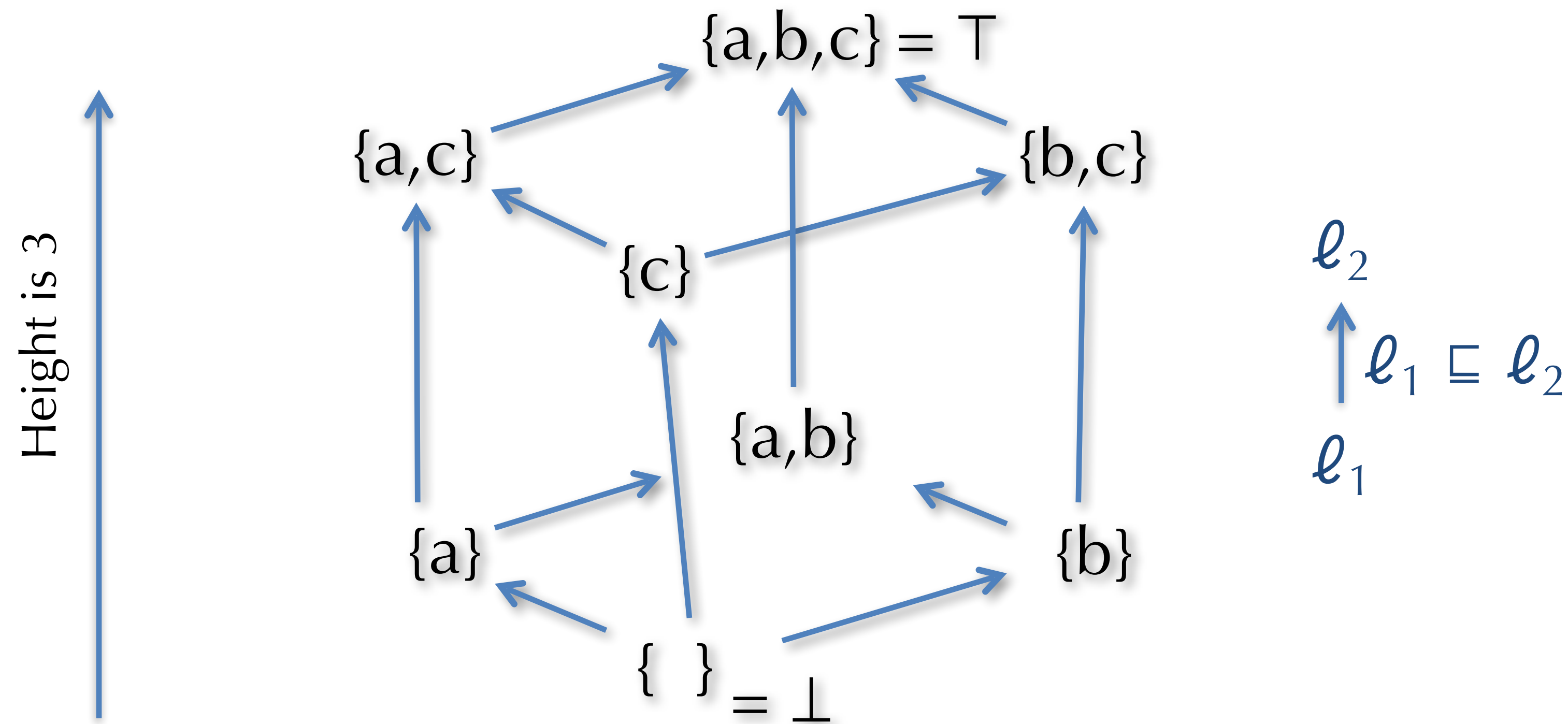
# $\mathcal{L}$ as a Partial Order

- $\mathcal{L}$  is a *partial order* defined by the ordering relation  $\sqsubseteq$ .
- A partial order is an ordered set.
- Some of the elements might be *incomparable*.
  - That is, there might be  $\ell_1, \ell_2 \in \mathcal{L}$  such that neither  $\ell_1 \sqsubseteq \ell_2$  nor  $\ell_2 \sqsubseteq \ell_1$
- Properties of a partial order:
  - *Reflexivity*:  $\ell \sqsubseteq \ell$
  - *Transitivity*:  $\ell_1 \sqsubseteq \ell_2$  and  $\ell_2 \sqsubseteq \ell_3$  implies  $\ell_1 \sqsubseteq \ell_3$
  - *Anti-symmetry*:  $\ell_1 \sqsubseteq \ell_2$  and  $\ell_2 \sqsubseteq \ell_1$  implies  $\ell_1 = \ell_2$
- Examples:
  - Integers ordered by  $\leq$
  - Sets ordered by  $\subseteq$  or  $\supseteq$



# Subsets of $\{a,b,c\}$ ordered by $\subseteq$

Partial order presented as a Hasse diagram.



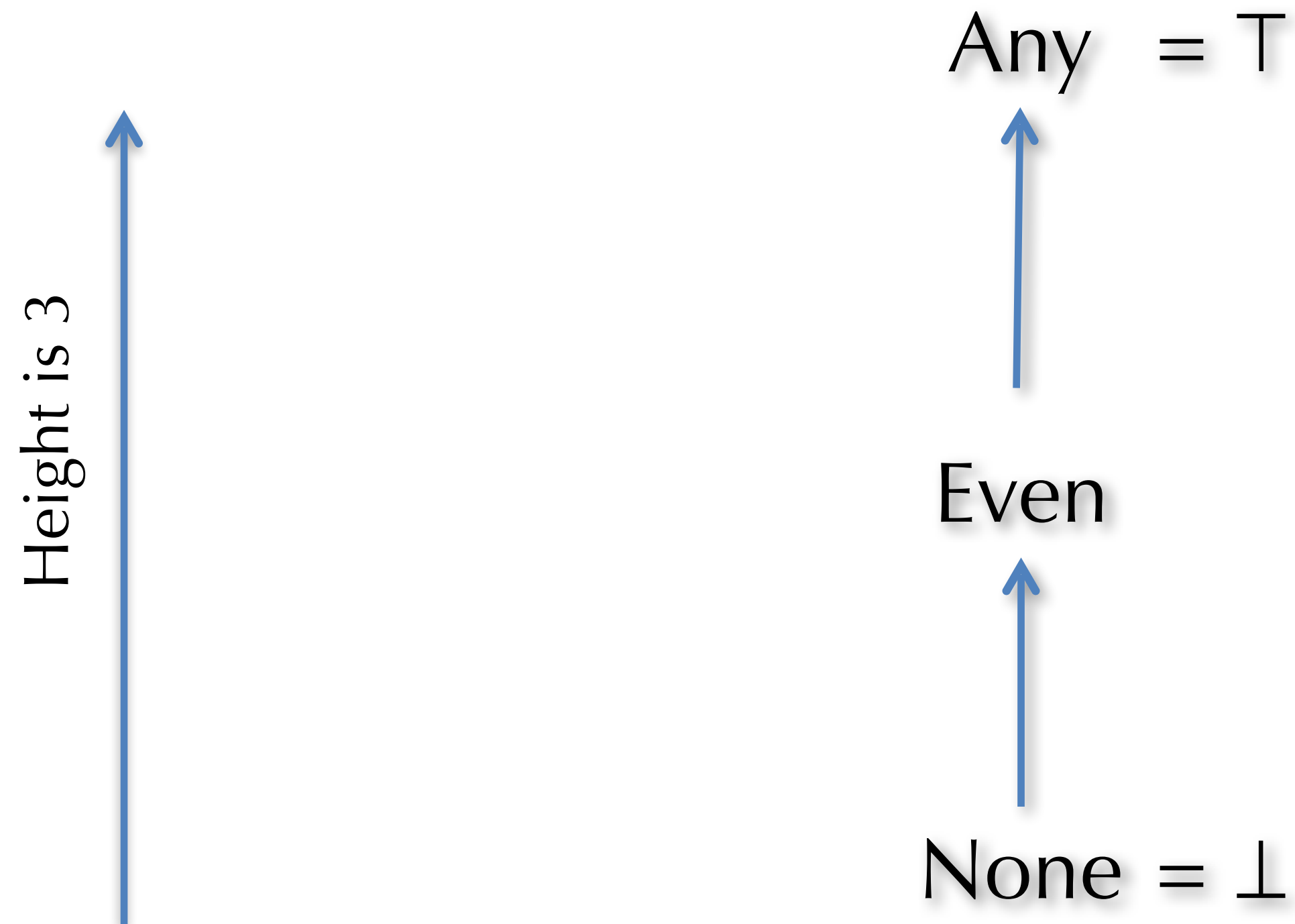
order  $\sqsubseteq$  is  $\subseteq$

meet  $\sqcap$  is  $\cap$

join  $\sqcup$  is  $\cup$

# Possible Values

Partial order presented as a Hasse diagram.



$$\text{Any} \sqcup \text{Any} = \text{Any} \sqcup \text{Even} = \text{Any} \sqcup \text{None} = \text{Any}$$

$$\text{Even} \sqcup \text{Even} = \text{Even} \sqcup \text{None} = \text{Even}$$

$$\text{None} \sqcup \text{None} = \text{None}$$

# Meets and Joins

- The combining operator,  $\sqcup$  operator is called the “join” operation.
- It constructs the *least upper bound*:
  - $\ell_1 \sqsubseteq \ell_1 \sqcup \ell_2$  and  $\ell_2 \sqsubseteq \ell_1 \sqcup \ell_2$   
“the join is an upper bound”
  - If  $\ell_1 \sqsubseteq \ell$  and  $\ell_2 \sqsubseteq \ell$  then  $\ell_1 \sqcup \ell_2 \sqsubseteq \ell$   
“there is no smaller upper bound”
- The dual operator  $\sqcap$  is called the “meet” operation.
- It constructs the *greatest lower bound*:
  - $\ell_1 \sqcap \ell_2 \sqsubseteq \ell_1$  and  $\ell_1 \sqcap \ell_2 \sqsubseteq \ell_2$   
“the meet is a lower bound”
  - If  $\ell \sqsubseteq \ell_1$  and  $\ell \sqsubseteq \ell_2$  then  $\ell \sqsubseteq \ell_1 \sqcap \ell_2$   
“there is no greater lower bound”
- A partial order that has all meets and joins is called a *lattice*.
  - If it has just meets, it’s called a *meet semi-lattice*.



# Another Way to Describe the Algorithm

- Algorithm repeatedly computes (for each node  $n$ ):
- $\text{out}[n] := F_n(\text{in}[n])$
- Equivalently:  $\text{out}[n] := F_n(\bigsqcup_{n' \in \text{pred}[n]} \text{out}[n'])$ 
  - By definition of  $\text{in}[n]$
- We can write this as a simultaneous update of the vector of  $\text{out}[n]$  values:
  - let  $x_n = \text{out}[n]$
  - Let  $\mathbf{X} = (x_1, x_2, \dots, x_n)$  it's a vector of points in  $\mathcal{L}$
  - $\mathbf{F}(\mathbf{X}) = (F_1(\bigsqcup_{j \in \text{pred}[1]} \text{out}[j]), F_2(\bigsqcup_{j \in \text{pred}[2]} \text{out}[j]), \dots, F_n(\bigsqcup_{j \in \text{pred}[n]} \text{out}[j]))$
- Any solution to the constraints is a *fixpoint*  $\mathbf{X}$  of  $\mathbf{F}$ 
  - i.e.  $\mathbf{F}(\mathbf{X}) = \mathbf{X}$

# Iteration Computes Fixpoints

- Let  $\mathbf{X}_0 = (\perp, \perp, \dots, \perp)$
- Each loop through the algorithm apply  $F$  to the old vector:  
 $\mathbf{X}_1 = \mathbf{F}(\mathbf{X}_0)$   
 $\mathbf{X}_2 = \mathbf{F}(\mathbf{X}_1)$   
...
- $\mathbf{F}^{k+1}(\mathbf{X}) = \mathbf{F}(\mathbf{F}^k(\mathbf{X}))$
- A fixpoint is reached when  $\mathbf{F}^k(\mathbf{X}) = \mathbf{F}^{k+1}(\mathbf{X})$ 
  - That's when the algorithm stops.
- Wanted: a minimal fixpoint
  - Because that one is more informative/useful for performing optimizations

# Monotonicity & Termination

- Each flow function  $F_n$  maps lattice elements to lattice elements; to be sensible is should be *monotonic*:
- $F : \mathcal{L} \rightarrow \mathcal{L}$  is *monotonic* iff:  
 $\ell_1 \sqsubseteq \ell_2$  implies that  $F(\ell_1) \sqsubseteq F(\ell_2)$ 
  - Intuitively: “If you have more information entering a node, then you have more information leaving the node.”
- Monotonicity lifts point-wise to the function:  $\mathbf{F} : \mathcal{L}^n \rightarrow \mathcal{L}^n$ 
  - vector  $(x_1, x_2, \dots, x_n) \sqsubseteq (y_1, y_2, \dots, y_n)$  iff  $x_i \sqsubseteq y_i$  for each  $i$
- Since we start at, each iteration moves up the lattice that  $\mathbf{F}$  is consistent:  $\perp \sqsubseteq \mathbf{F}(\perp)$ 
  - So each iteration moves at least one step down the lattice (for some component of the vector)
  - $\perp \sqsubseteq \mathbf{F}(\perp) \sqsubseteq \mathbf{F}(\mathbf{F}(\perp)) \sqsubseteq \dots$
- Therefore, # steps needed to reach a fixpoint is at most the height  $H$  of  $\mathcal{L}$  times the number of nodes:  $O(Hn)$



# “Classic” Constant Propagation

- Constant propagation can be formulated as a dataflow analysis.

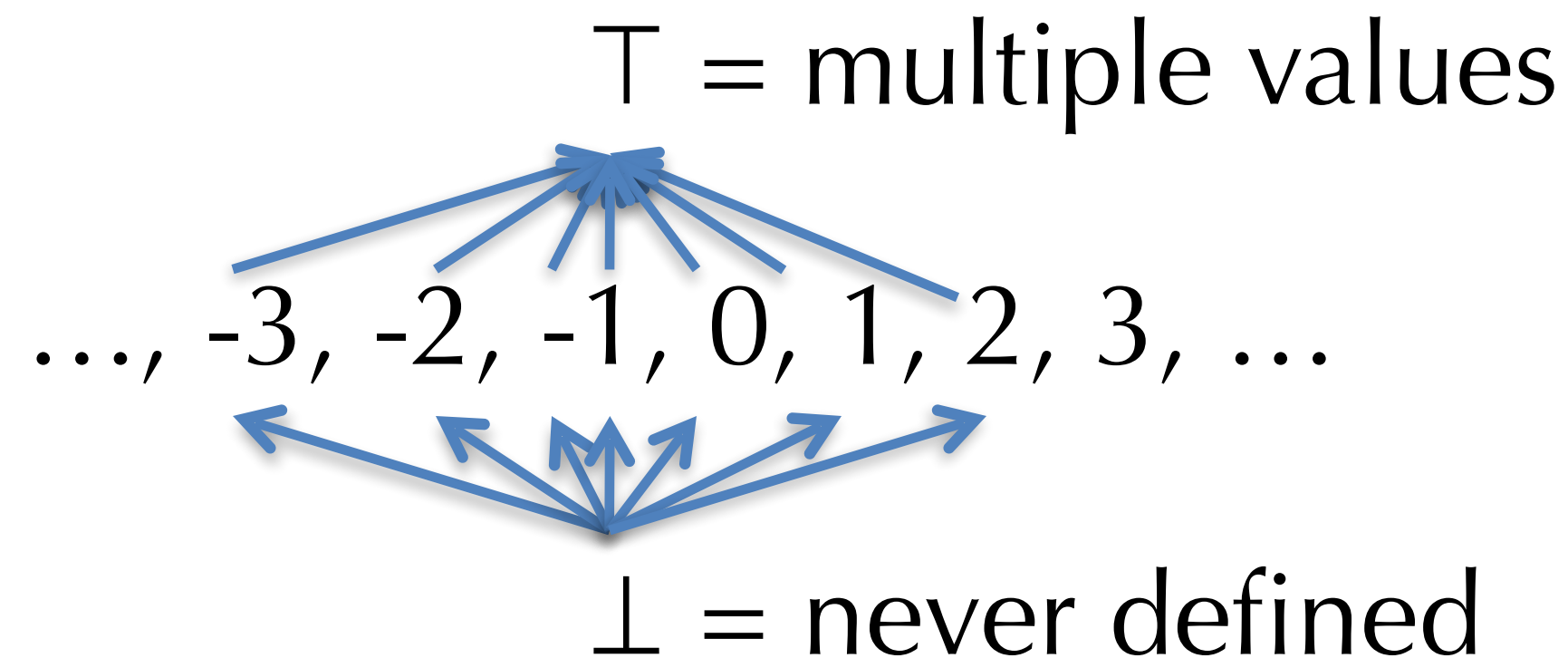
- Idea: propagate and fold integer constants in one pass:

|              |                                                                                     |           |
|--------------|-------------------------------------------------------------------------------------|-----------|
| $x = 1;$     |                                                                                     | $x = 1;$  |
| $y = 5 + x;$ |  | $y = 6;$  |
| $z = y * y;$ |                                                                                     | $z = 36;$ |

- Information about a single variable:
  - Variable is never defined.
  - Variable has a single, constant value.
  - Variable is assigned multiple values.

# Domains for Constant Propagation

- We can make a constant propagation lattice  $L$  for *one variable* like this:



- To accommodate multiple variables, we take the product lattice, with one element per variable.
  - Assuming there are three variables,  $x$ ,  $y$ , and  $z$ , the elements of the product lattice are of the form  $(\ell_x, \ell_y, \ell_z)$ .
  - Alternatively, think of the product domain as a context that maps variable names to their “*abstract interpretations*”
- What are “meet” and “join” in this product lattice?
- What is the height of the product lattice?



# Compiler Frontend: Lexing and Parsing

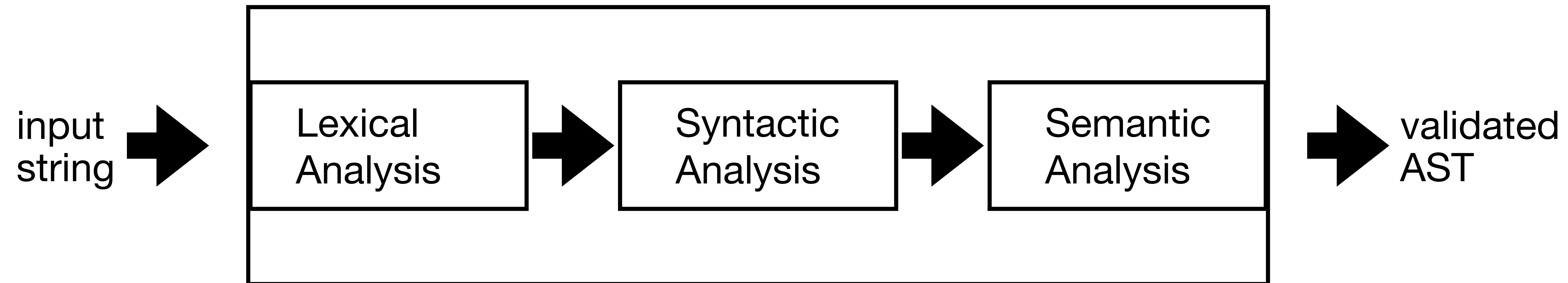
# Lexical Analysis

# Compiler Frontends

The task of the compiler frontend is take the input program as a string and

1. Validate that it is a well-formed program
2. Output an **Abstract Syntax Tree** that is more convenient for the rest of the compiler pipeline to use

## Compiler Frontend



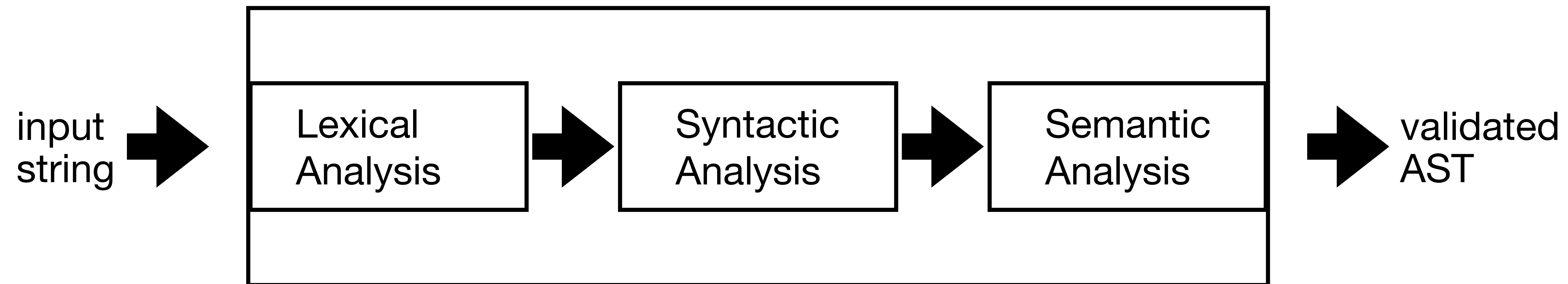


# Compiler Frontends

So far in class we have only implemented a small part of the frontend: the "semantic analysis" phase. For Snake programs this meant checking variables and functions are used properly.

Remainder of the semester: first two components of the frontend **lexing/lexical analysis** and **parsing/syntactic analysis**

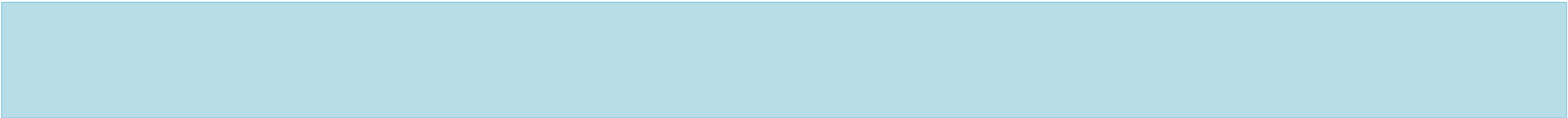
## Compiler Frontend



# Compiler Frontends

The task of the lexing and parsing phases is to **find** structure (abstract syntax trees) in an unstructured representation (strings of characters).

Works differently from passes we've seen so far, which all had tree-structured programs as inputs.



Lexical analysis, tokens, regular expressions, automata

# LEXING



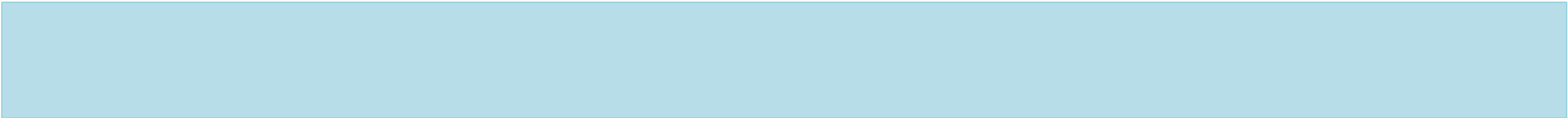
# First Step: Lexical Analysis

- Change the *character stream* "if (b == 0) { a = 0; }" into *tokens*

|    |   |   |    |   |   |   |   |   |   |   |   |
|----|---|---|----|---|---|---|---|---|---|---|---|
| if | ( | b | == | 0 | ) | { | a | = | 0 | ; | } |
|----|---|---|----|---|---|---|---|---|---|---|---|

IF; LPAREN; Ident("b"); EQEQ; Int(0); RPAREN; LBRACE;  
Ident("a"); EQ; Int(0); SEMI; RBRACE

- Token: data type that represents indivisible "chunks" of text:
  - Identifiers: a y11 elsex \_100
  - Keywords: if else while
  - Integers: 2 200 -500 5L
  - Floating point: 2.0 .02 1e5
  - Symbols: + \* ^ { } ( ) ++ << >> >>>
  - Strings: "x" "He said, \"Are you?\""
  - Comments: // 483: Project 1 ... /\* foo \*/
- Often delimited by *whitespace* (' ', \t, etc.)
  - In some languages (e.g. Python or Haskell) whitespace is significant



How hard can it be?  
handlex0.rs handlex.rs

## DEMO: LEXING BY HAND



# Lexing By Hand

- How hard can it be?
  - Tedious and painful!
- Problems:
  - Precisely define tokens
  - Matching tokens simultaneously
  - Reading too much input (need look ahead)
  - Error handling
  - **Hard to compose/interleave** tokenizer code
  - Hard to maintain



# **PRINCIPLED SOLUTION TO LEXING**

# Lexer Generators

- Lexers are
  - tedious to write
  - easy to mess up, hard to read
  - repetitive: most lexers are essentially the same algorithm but different specifics
- Solution: make a new, high-level **domain-specific language** for writing lexers
  - Easier for humans to read, write, update
  - Efficient implementation strategy implemented once and for all
    - limited computational power -> Rice's theorem no longer applies, can get "perfect" optimization
- Examples:
  - lex/flex
  - antlr
  - In Rust: logos, lalrpop



# A Lexer Compiler

Now we have reduced lexing to a mini-compiler task. So let's do what we've been doing all semester!

- Design a **language** for lexers
- Describe its **semantics**
- Transform that language into **intermediate representations**
- **Optimize** the intermediate representation
- **Generate code** that implements our optimized IR.

# A Language for Lexers

- What language should we use to describe a lexer?
- What does a lexer need to do?
- A lexer needs to specify two things
  1. What strings make up the "tokens" of our language
  2. How to turn these abstract tokens into data that our compiler pipeline can use
- 1 is the hard part: we need a **language** for describing **sets of strings**

# Formal Languages

- First we fix the "alphabet" of characters  $\Sigma$ .
  - Common alphabets  $\{ 0 , 1 \}$  for bitstrings
  - 0-255 for ASCII characters
  - very large set of Unicode "characters"
- A string (over  $\Sigma$ ) is a finite sequence of characters (i.e., elements of  $\Sigma$ )
- A **formal language** is a **subset** of strings.



# Formal Languages

- A **formal language** is a **subset** of strings.
- Examples that we use in lexing:
  - Singletons for particular keywords { "def" } {"let"} {"extern"} or syntactic tokens { ")" } { "(" } { ":" }
  - Booleans { "true" , "false" }
  - The set of all number literals { 0, -1, +1, 199239190, ... }
  - The set of all valid variable names { "x", "y", "z",... but not "def", "extern" etc }
- A lexer generator then needs a **syntax** for describing such formal languages
  - A language of expressions
  - Which are given a **semantics** as formal languages

# Regular Expressions

- Regular expressions are a syntax for defining formal languages
- A regular expression  $R$  has one of the following forms:

- $\epsilon$                        $\{ "" \}$  contains only the empty string
- $\emptyset$                        $\{ \}$  Empty set
- $'a'$                        $\{ "a" \}$  An ordinary character stands for itself
- $R_1 \mid R_2$               Alternatives, stands for **union** of  $R_1$  or  $R_2$   
 $\text{Sem}(R_1) \cup \text{Sem}(R_2)$
- $R_1 R_2$                   Concatenation, stands for  $R_1$  followed by  $R_2$   
 $\{ w_1 w_2 \mid w_1 \in \text{Sem}(R_1), w_2 \in \text{Sem}(R_2) \}$
- $R^*$                       Kleene star, stands for *zero or more* repetitions of  $R$   
 $\{ w_1 w_2 \dots \mid w_1 \in R, w_2 \in R, w_3 \in R, \dots \}$   
(includes the base case that there are no substrings)



# Regular Expressions

- *Useful extensions:*
  - “foo”      Strings, equivalent to ' f ' ' o ' ' o '
  - R+      One or more repetitions of R, equivalent to RR\*
  - R?      Zero or one occurrences of R, equivalent to (  $\epsilon$  | R )
  - [ ' a ' - ' z ' ]      One of a or b or c or ... z, equivalent to ( a | b | ... | z )
  - [ ^ ' 0 ' - ' 9 ' ]      Any character except 0 through 9

# Example Regular Expressions

- Recognize the keyword “if”:  
    `"if"`
- Recognize a digit:
  - `'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'`
  - using abbreviations: `['0'-'9']`
- Recognize an integer literal:
  - `'-'? ['0'-'9']+`
- Recognize an identifier:  
    `(['a'-'z'] | ['A'-'Z']) (['0'-'9'] | '_' | ['a'-'z'] | ['A'-'Z']) *`
- In practice, it's useful to be able to *name* regular expressions:

```
let lowercase = ['a'-'z']  
let uppercase = ['A'-'Z']  
let character = uppercase | lowercase
```

# Example Regular Expressions

- Exercise: Write a regular expression for string literals that allow escaping.
- Alphabet: just { a , b , " , \ }
- Examples of valid inputs:
  - "abababab"
  - "\\"
  - "\\\""
  - "\\a"
- Examples of invalid inputs:
  - "aba
  - abaa"
  - "\"
- Solution:
  - "(a | b | \"(a | b | \" | \"))\* \""



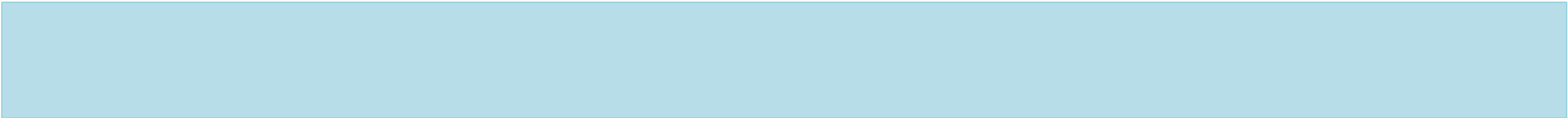
# Overlapping Regexes

- Consider the input string: `ifx = 0`
  - Could lex as: 

|                 |                |                |                |
|-----------------|----------------|----------------|----------------|
| <code>if</code> | <code>x</code> | <code>=</code> | <code>0</code> |
|-----------------|----------------|----------------|----------------|

 or as: 

|                  |                |                |
|------------------|----------------|----------------|
| <code>ifx</code> | <code>=</code> | <code>0</code> |
|------------------|----------------|----------------|
- The regular expressions alone are overlapping, need to disambiguate somehow.
- Most languages choose “longest match”
  - So the 2<sup>nd</sup> option above will be picked
- Conflicts: arise due to two tokens whose regular expressions have a shared prefix
  - Ties broken by giving some matches higher priority
  - Example: keywords have priority over identifiers
  - Usually specified by order the rules appear in the lex input file



regexlex.rs

# DEMO: REGEX-BASED LEXING

# Lexer Generators as Compilers

Lexing is tedious and error-prone to implement manually. Just like assembly code! Instead, implement a **lexer generator**, a compiler for a domain-specific language for lexers.

Just like the compilers we've been working on this semester:

1. Design a source **language** for lexers: **regular expressions** + action code
2. Describe its **semantics**: regular expressions are a syntax for **formal languages**
3. Transform into an intermediate representation: **non-deterministic finite automata**
4. **Optimize** that intermediate representation: **determinize** and **minimize** the NFA into a DFA
5. **Generate** code from the optimized IR



# Terminology

A **regular expression**  $R$  is an expression built up from epsilon, empty, single characters, disjunction, sequencing and Kleene star

The semantics of a regular expression is that it represents a **formal language** a subset of all possible input strings

A **recognizer** for a regular expression  $R$ , is a function  $\text{String} \rightarrow \text{Bool}$  that outputs true if and only if the input string is in the formal language described by the regular expression

The core of implementing a lexer is implementing **recognizers** for regular expressions. But it's not the entirety: we also need to be able to find the **longest match** for multiple regular expressions.

# Recognizing Regular Languages

How can we efficiently implement a recognizer for a regular language?

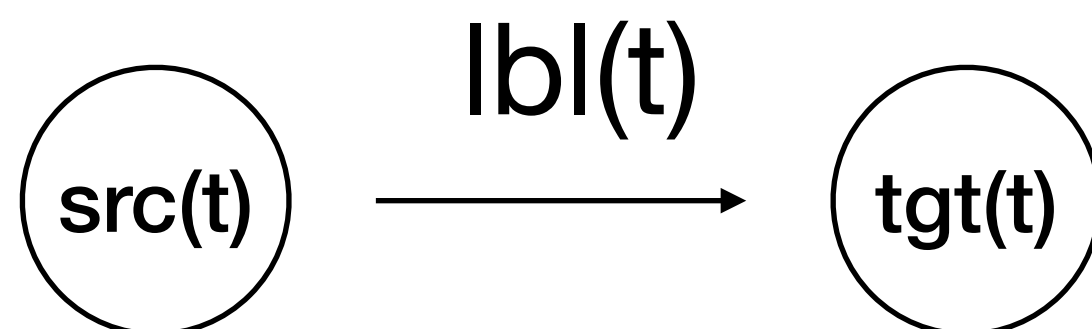
- Finite Automata
- DFA (Deterministic Finite Automata)
- NFA (Non-deterministic Finite Automata)



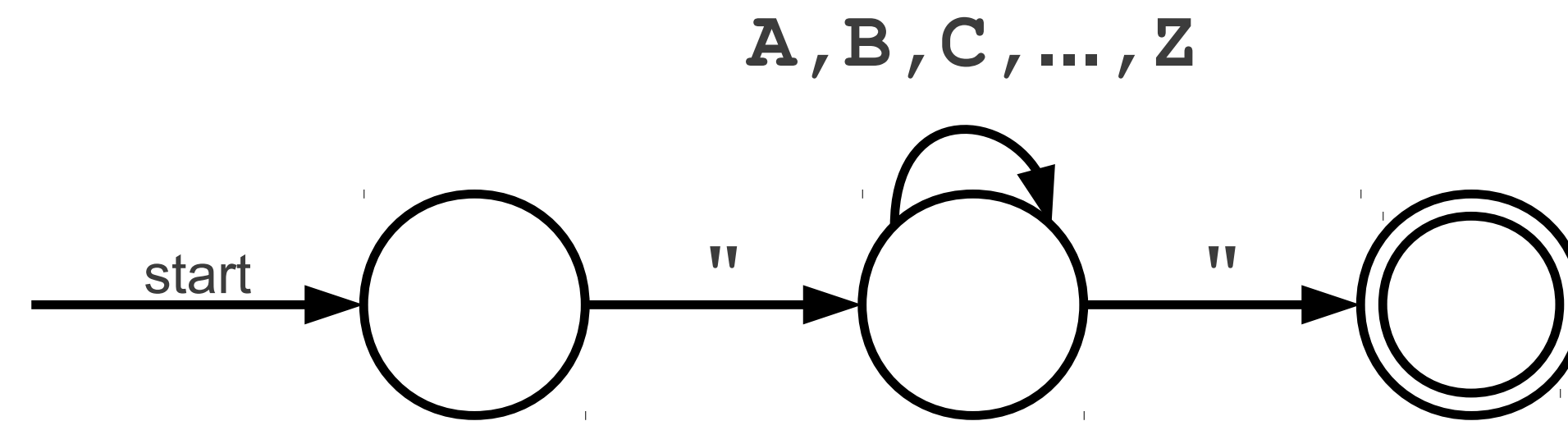
# Finite State Automata

A (non-deterministic) finite state automaton over an alphabet  $\Sigma$  consists of

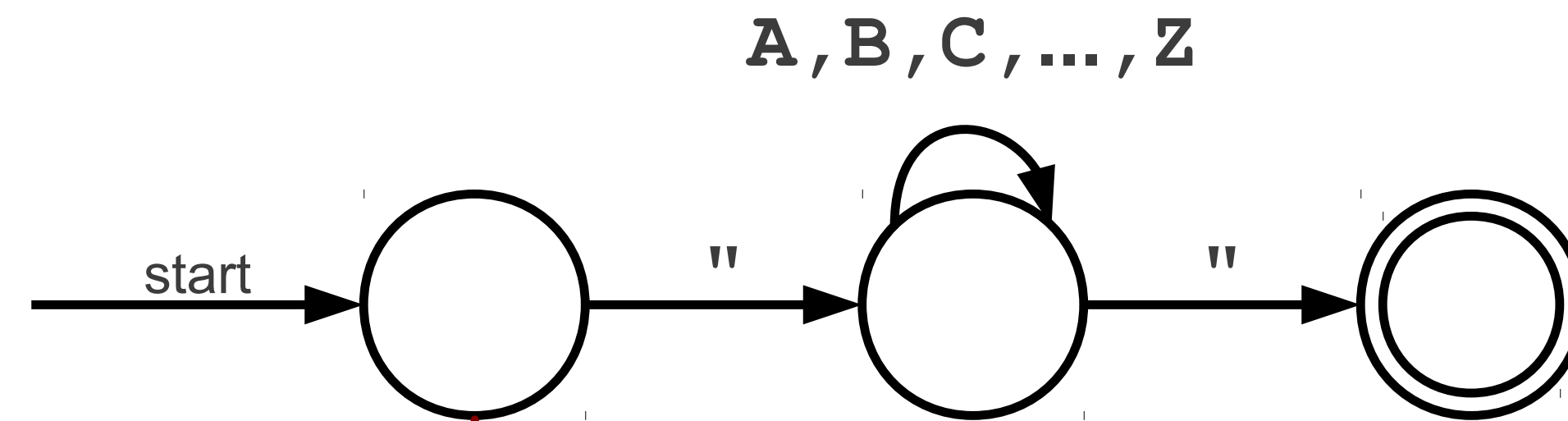
- A finite set of states **S**
- A distinguished start state **s<sub>0</sub> ∈ S**
- A subset of accepting states **Acc ⊆ S**
- A set of transitions  $\delta$ , where each transition  $t \in \delta$  has
  - a source state **src(t)**
  - a target state **tgt(t)**
  - a label **lbl(t)**, which is either a character  $c \in \Sigma$  or  $\epsilon$



# A Simple Automaton



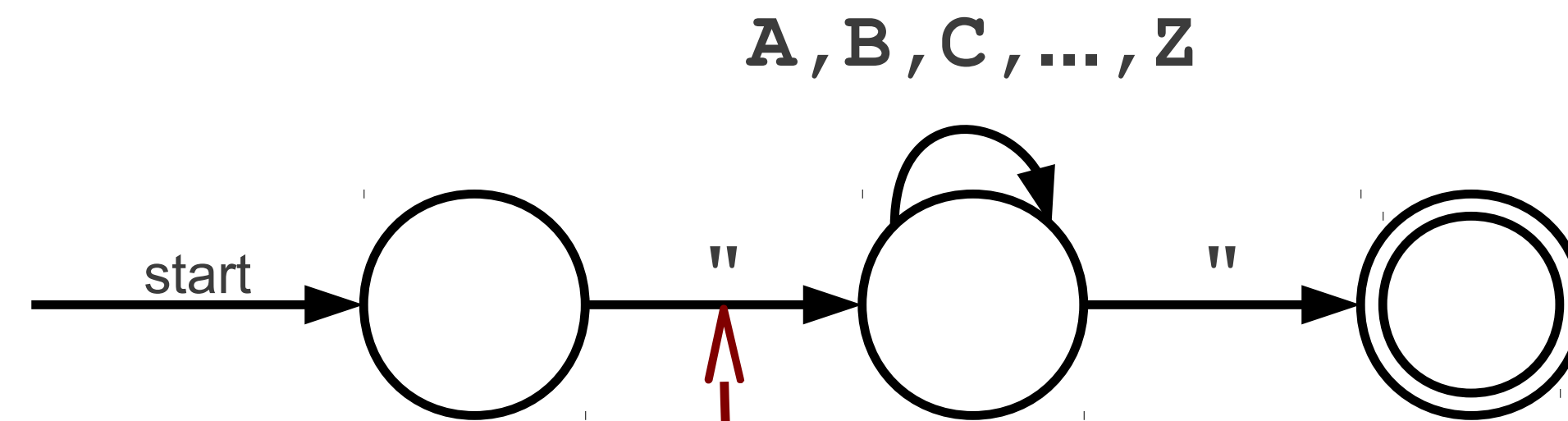
# A Simple Automaton



Each circle is a **state** of the automaton. The automaton's configuration is determined by what state(s) it is in.

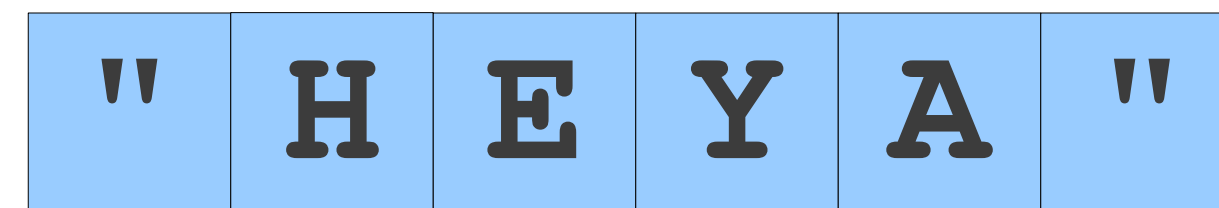
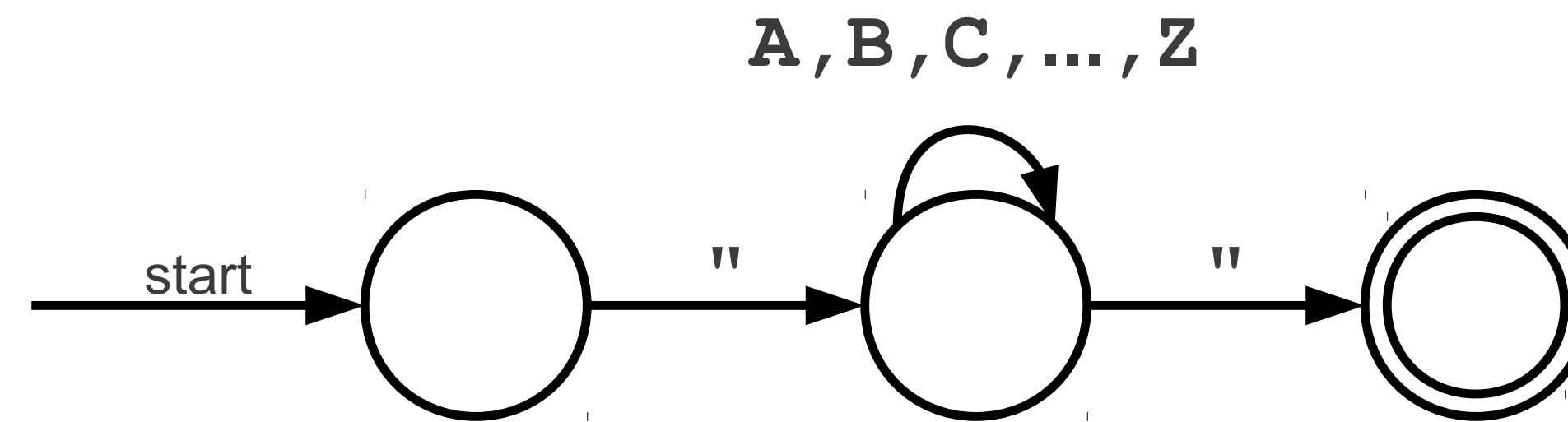


# A Simple Automaton



These arrows are called **transitions**. The automaton changes which state(s) it is in by following transitions.

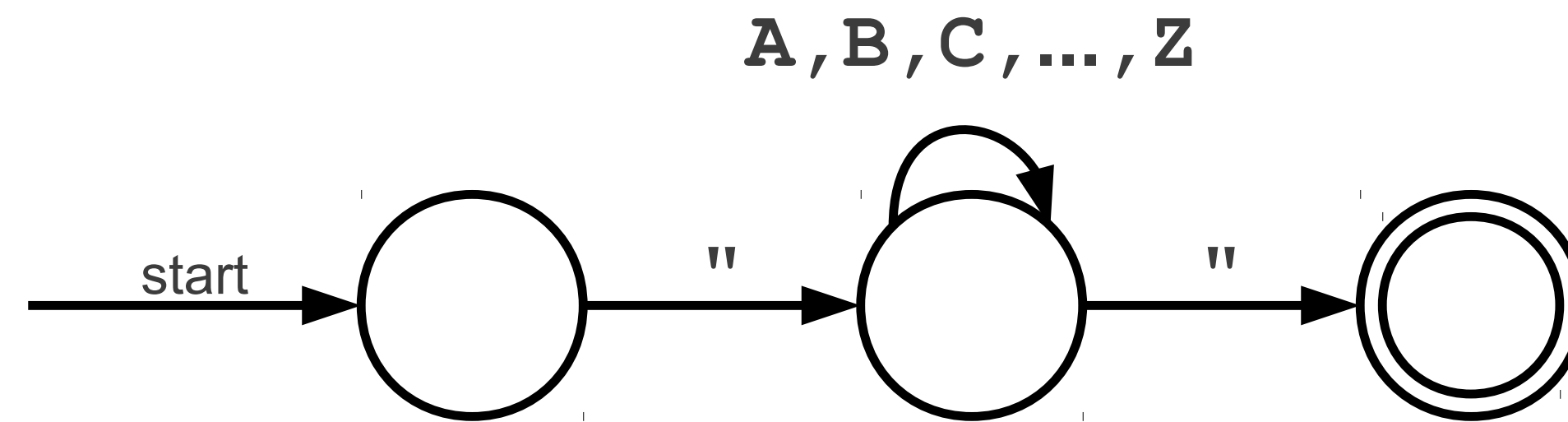
# A Simple Automaton



Finite Automata: Takes an input string and determines whether it's a valid sentence of a language

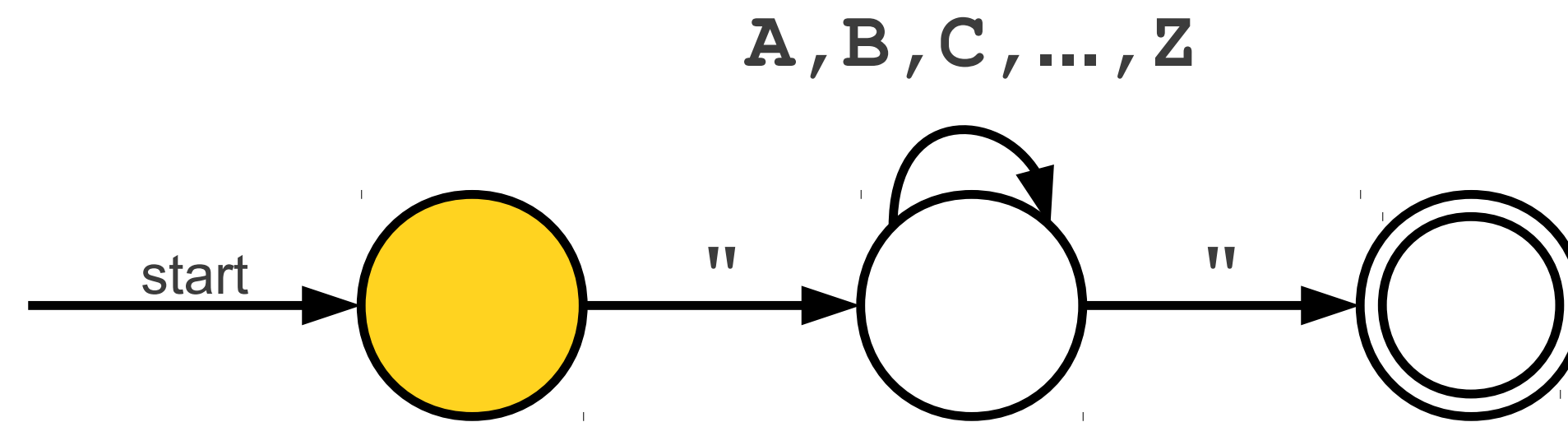
accept or reject

# A Simple Automaton

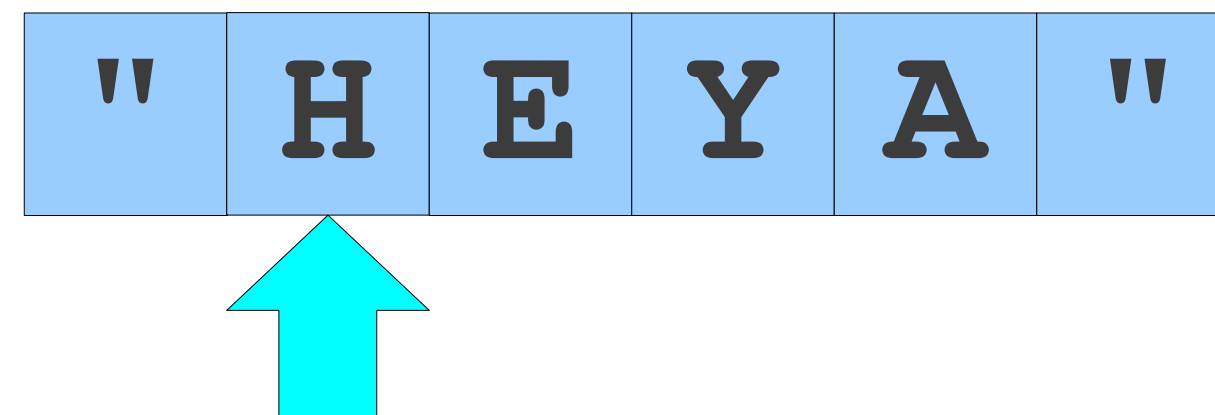
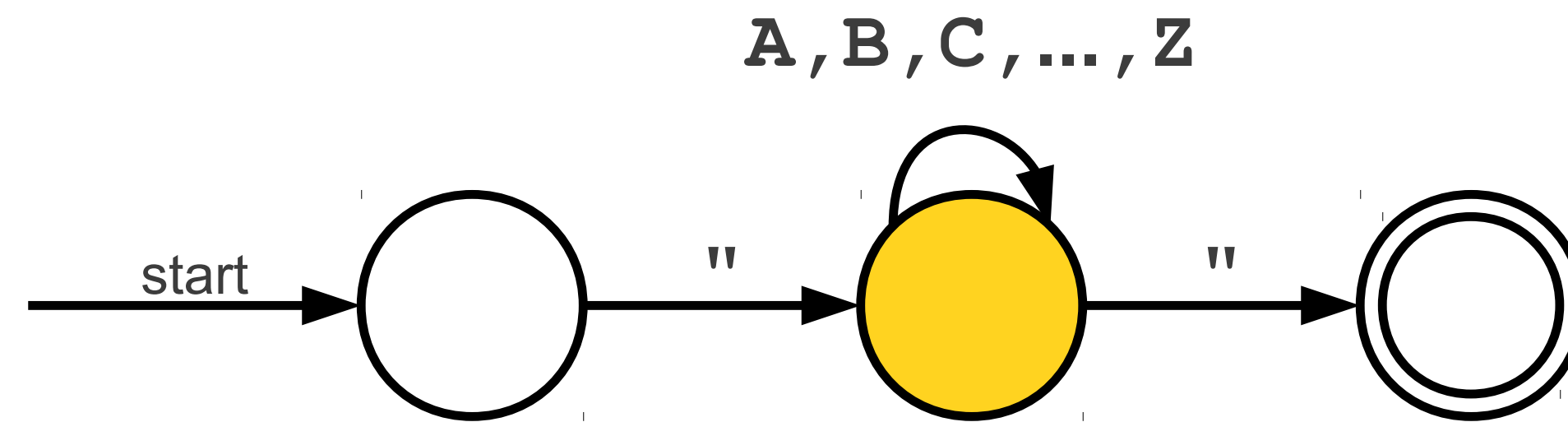




# A Simple Automaton

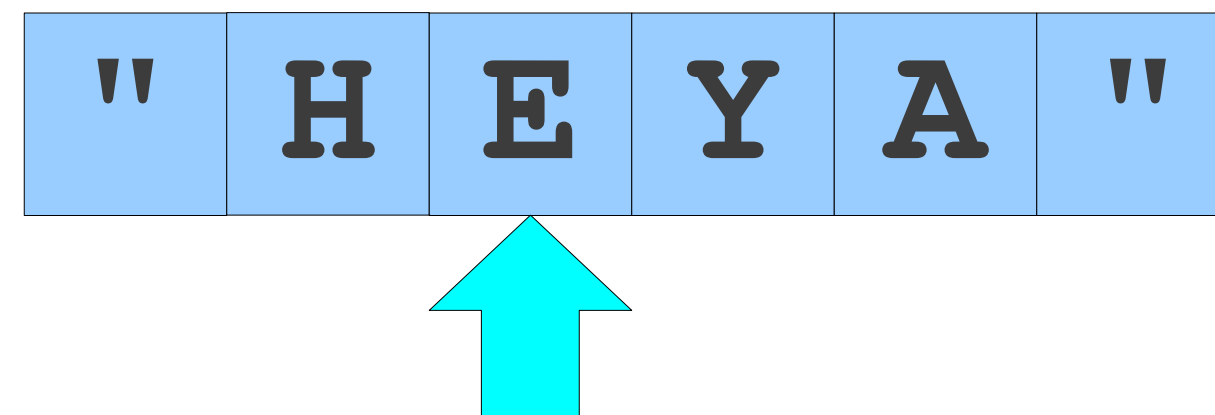
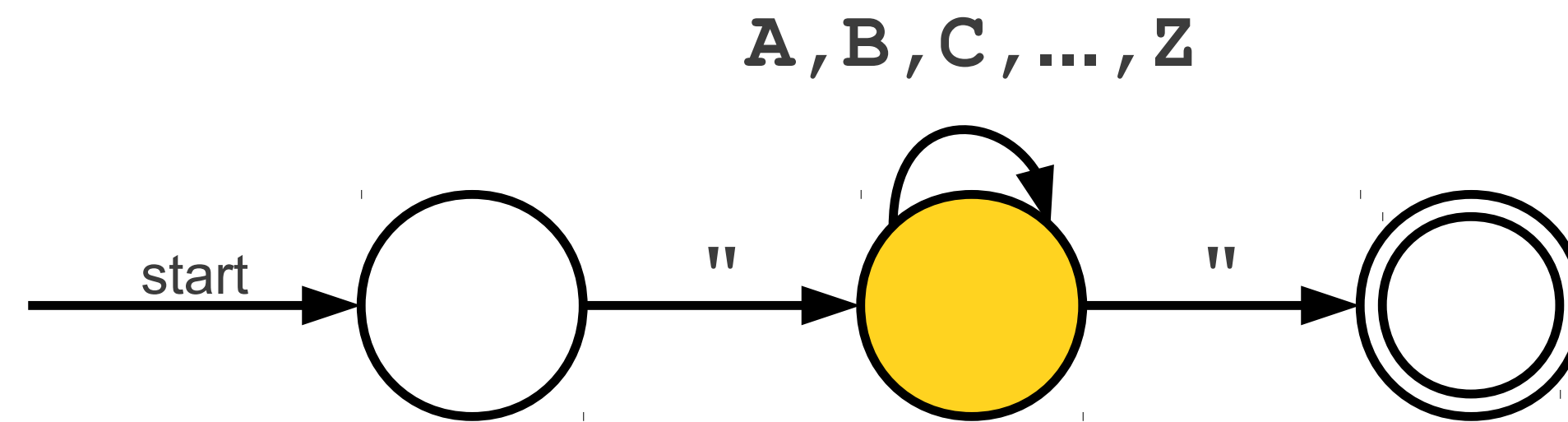


# A Simple Automaton

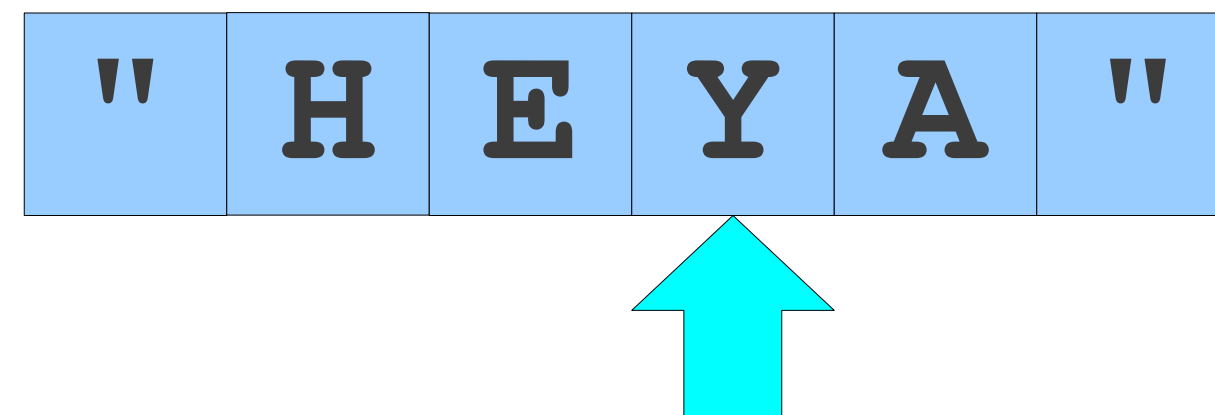
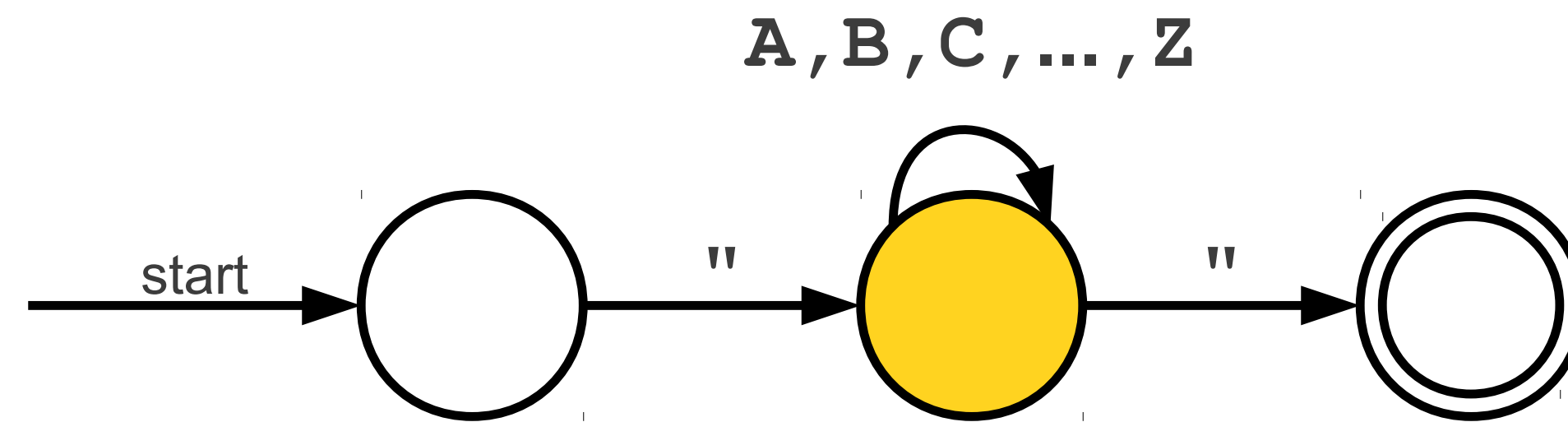




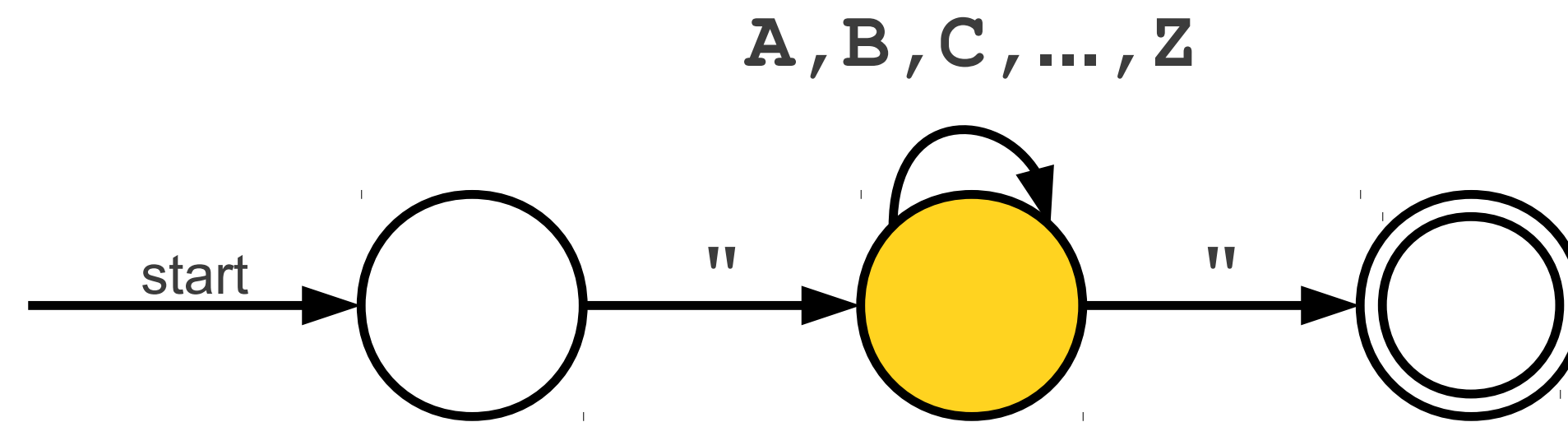
# A Simple Automaton



# A Simple Automaton

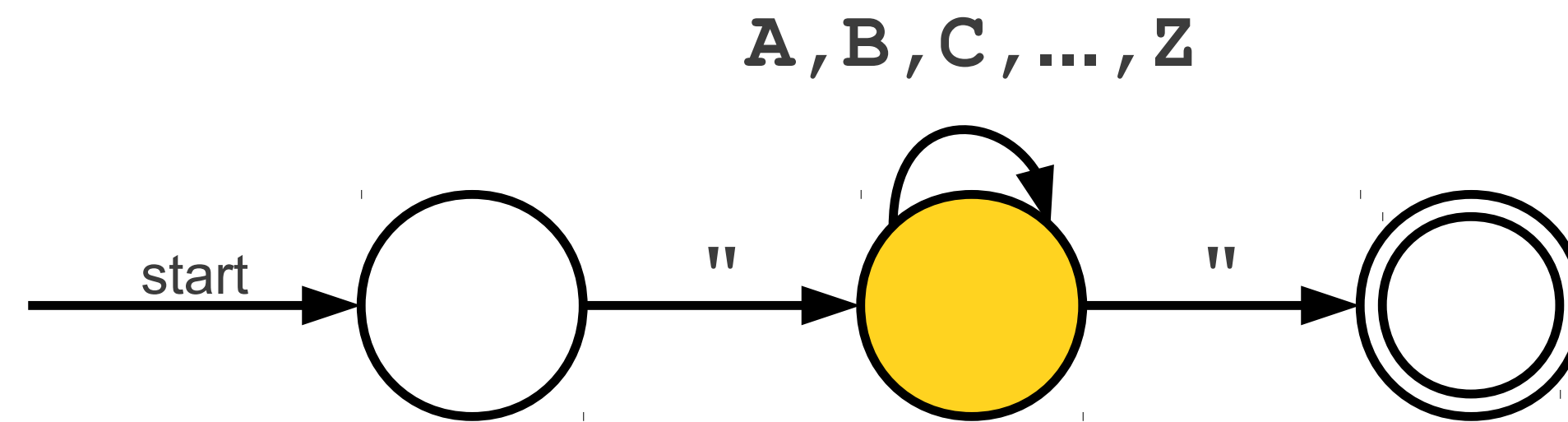


# A Simple Automaton

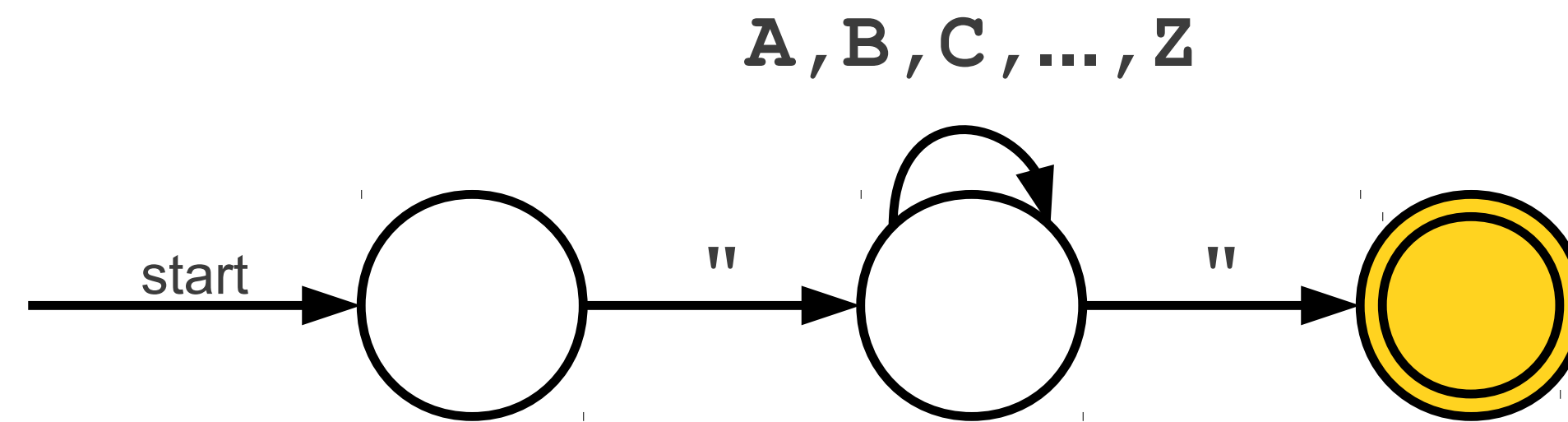




# A Simple Automaton

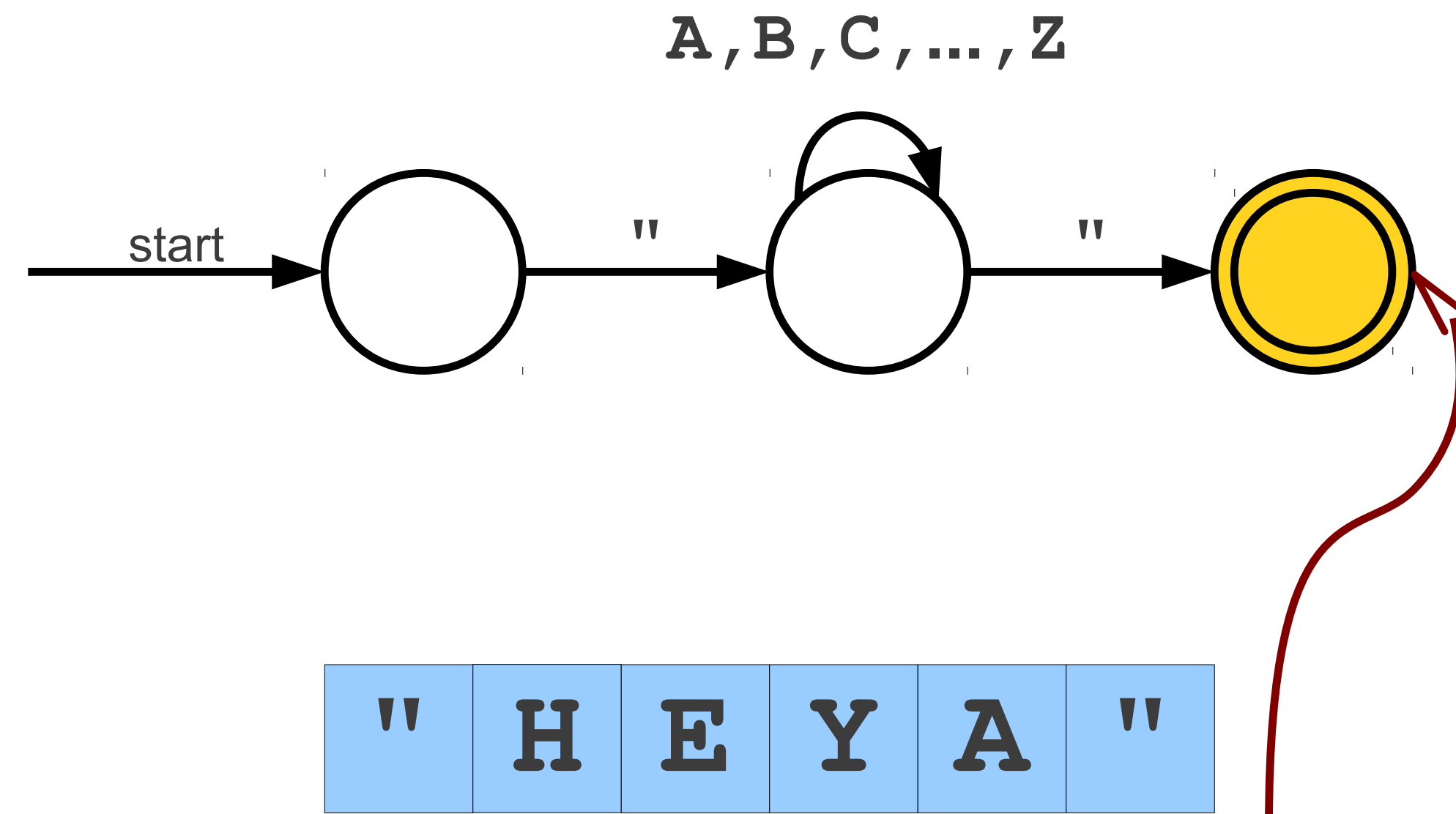


# A Simple Automaton



|   |   |   |   |   |   |
|---|---|---|---|---|---|
| " | H | E | Y | A | " |
|---|---|---|---|---|---|

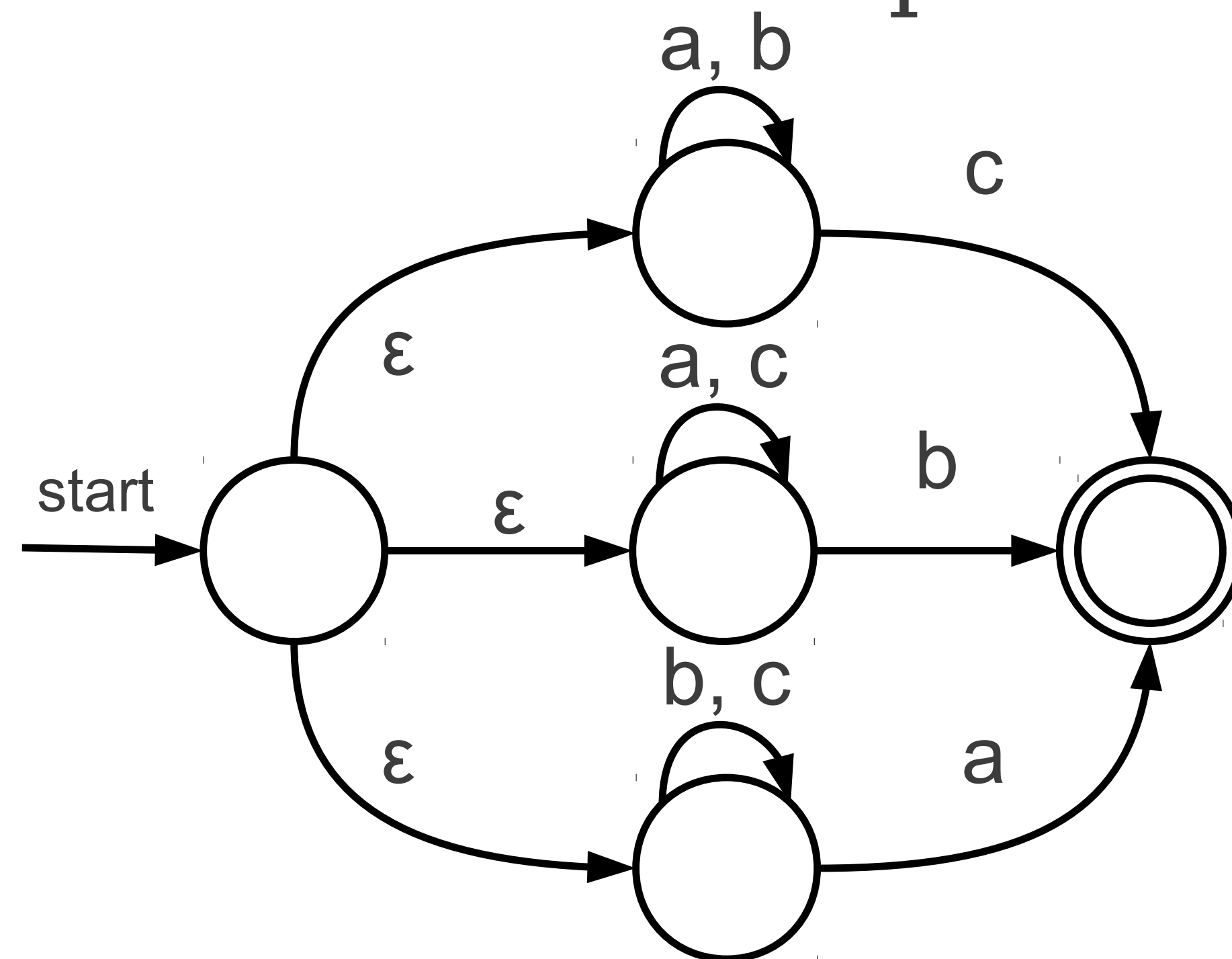
# A Simple Automaton



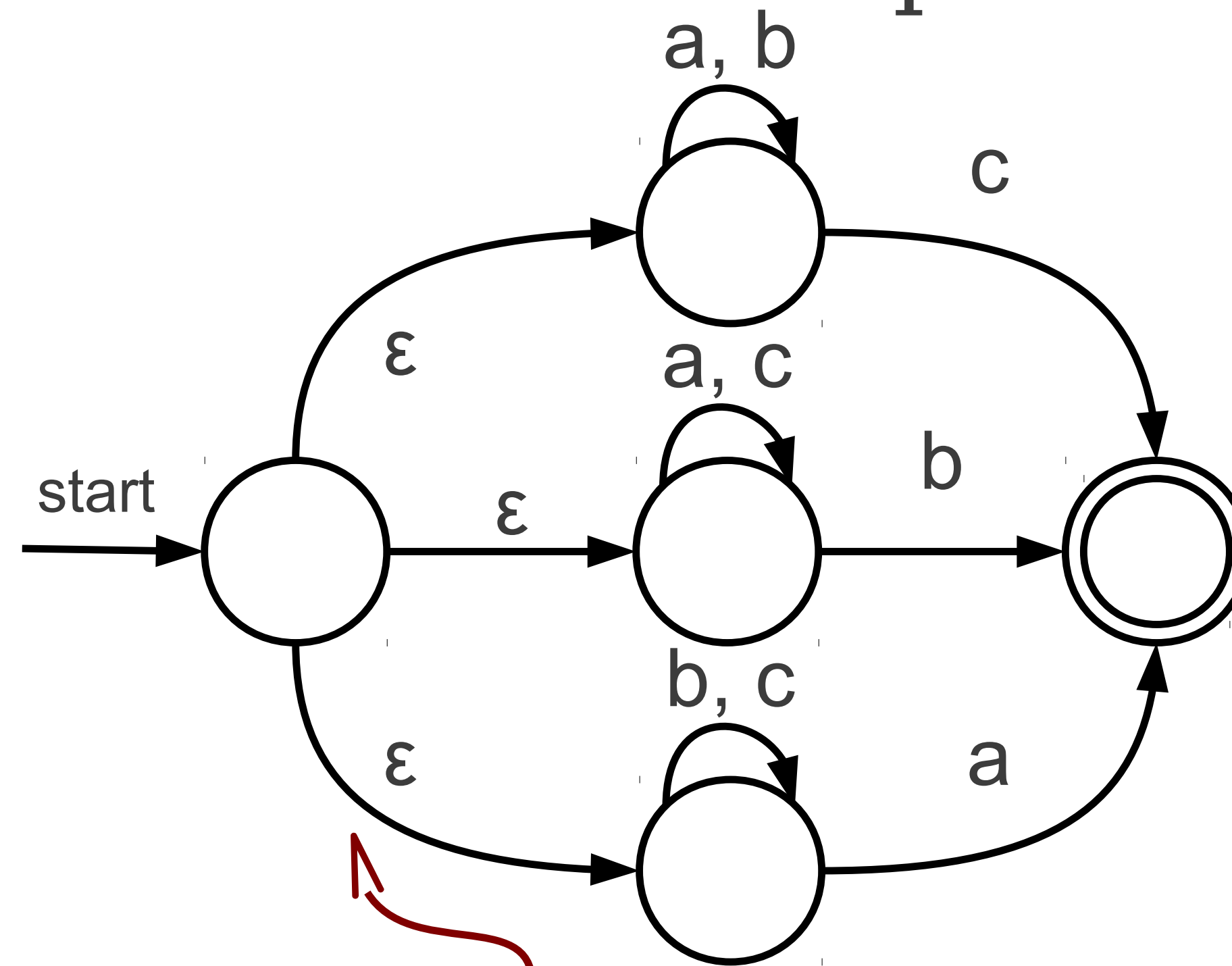
The double circle indicates that this state is an **accepting state**. The automaton accepts the string if it ends in an accepting state.



# An Even More Complex Automaton



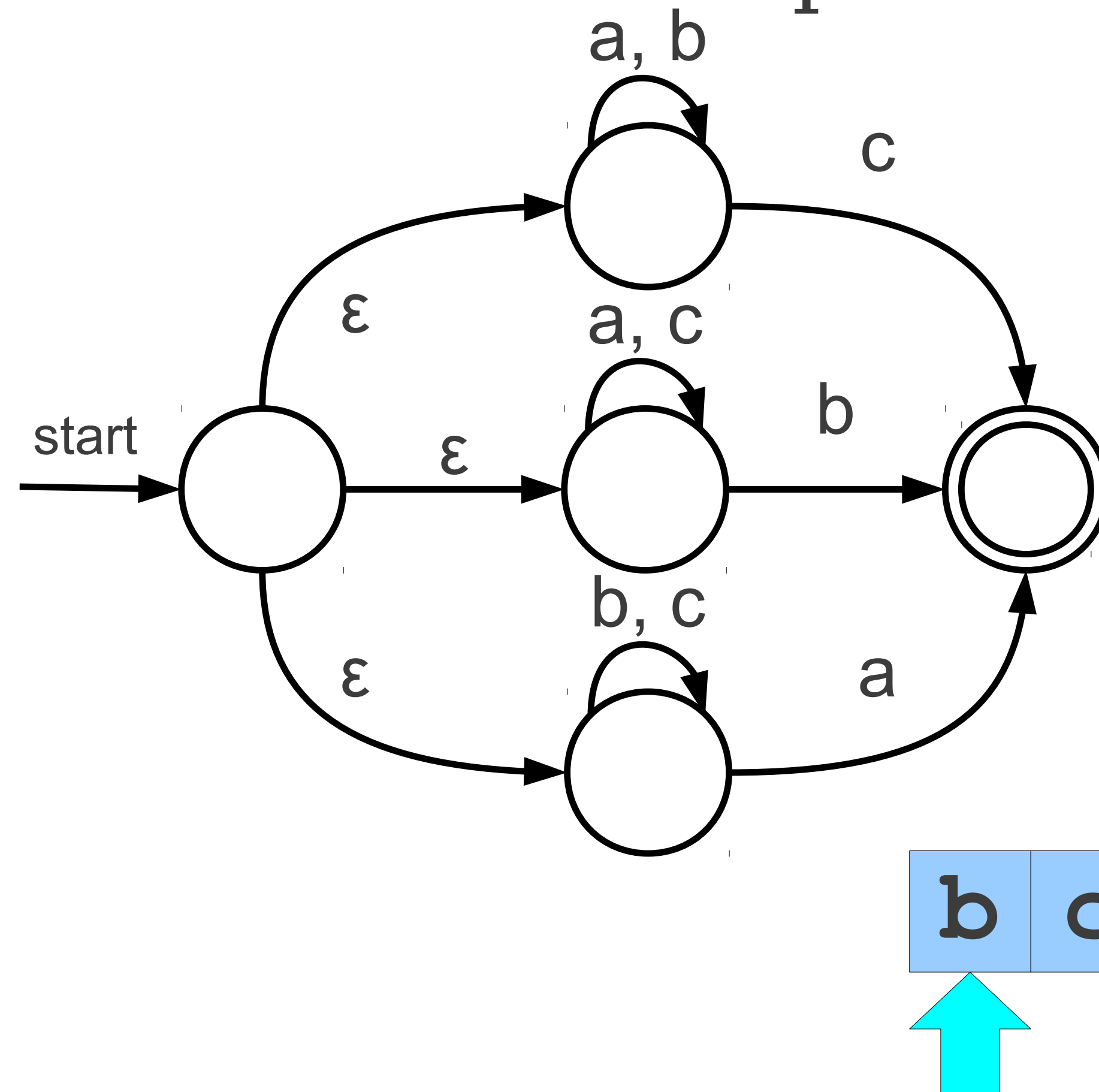
# An Even More Complex Automaton



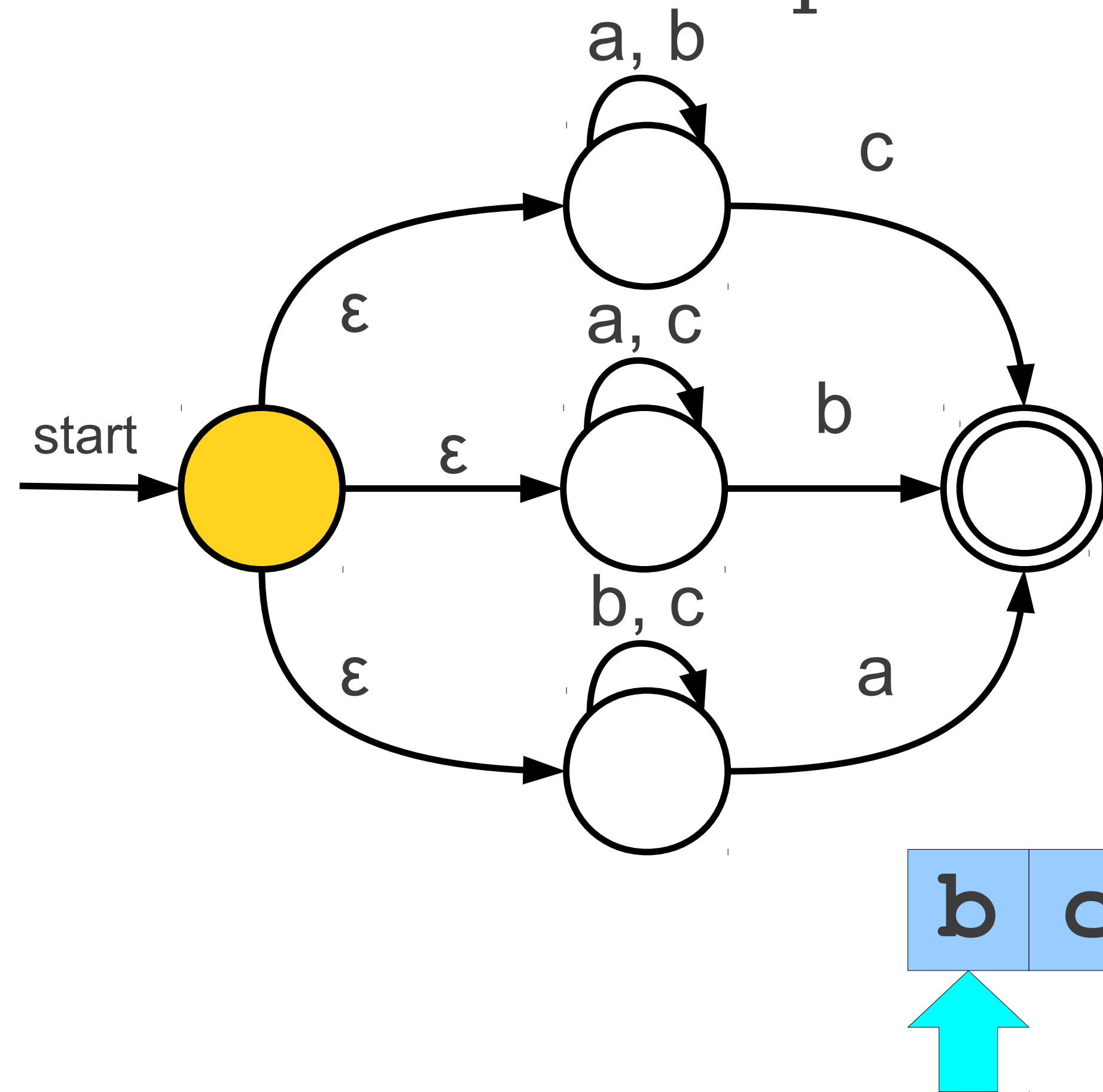
These are called  $\epsilon$ -transitions. These transitions are followed automatically and without consuming any input.



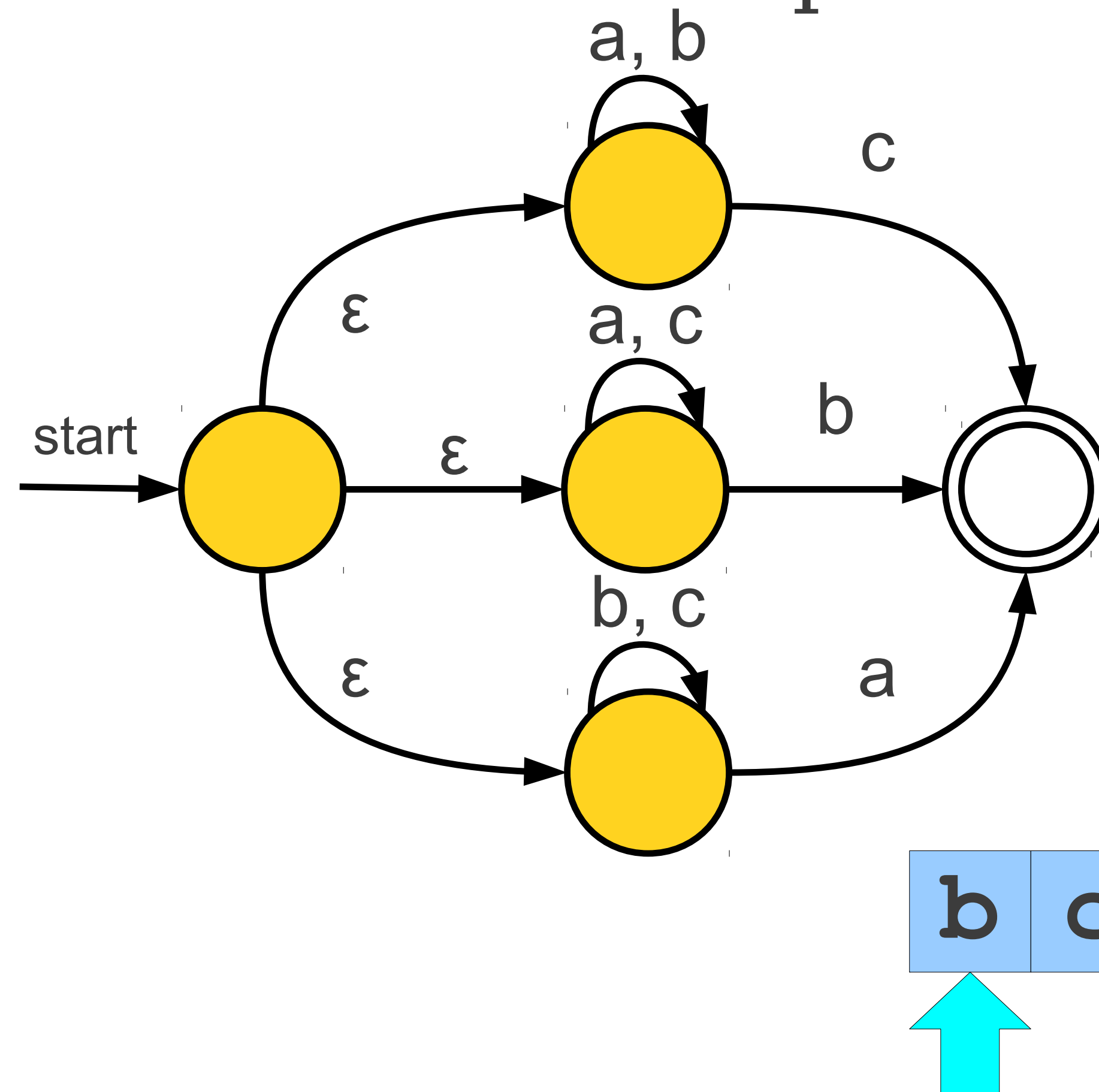
# An Even More Complex Automaton



# An Even More Complex Automaton

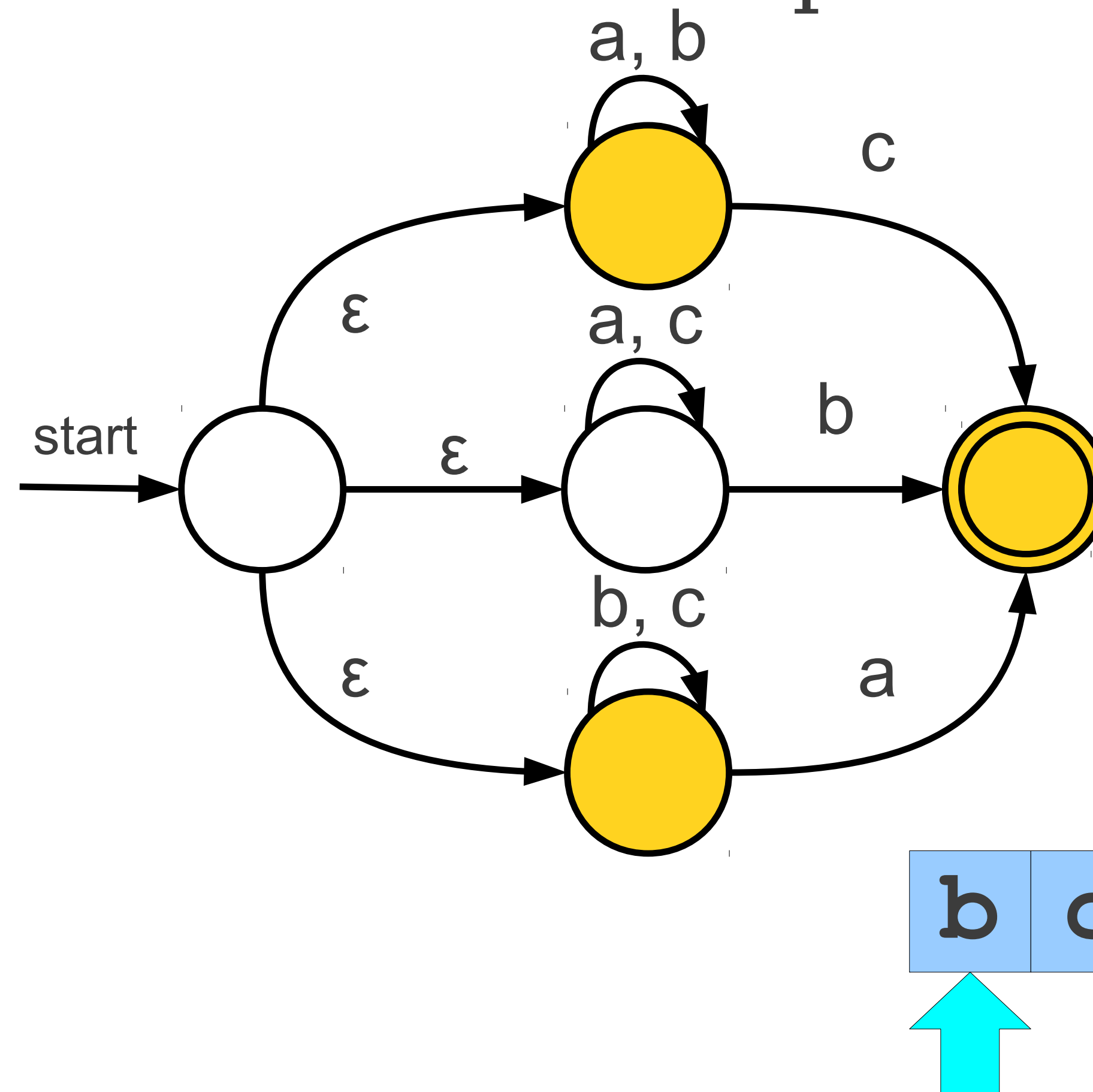


# An Even More Complex Automaton

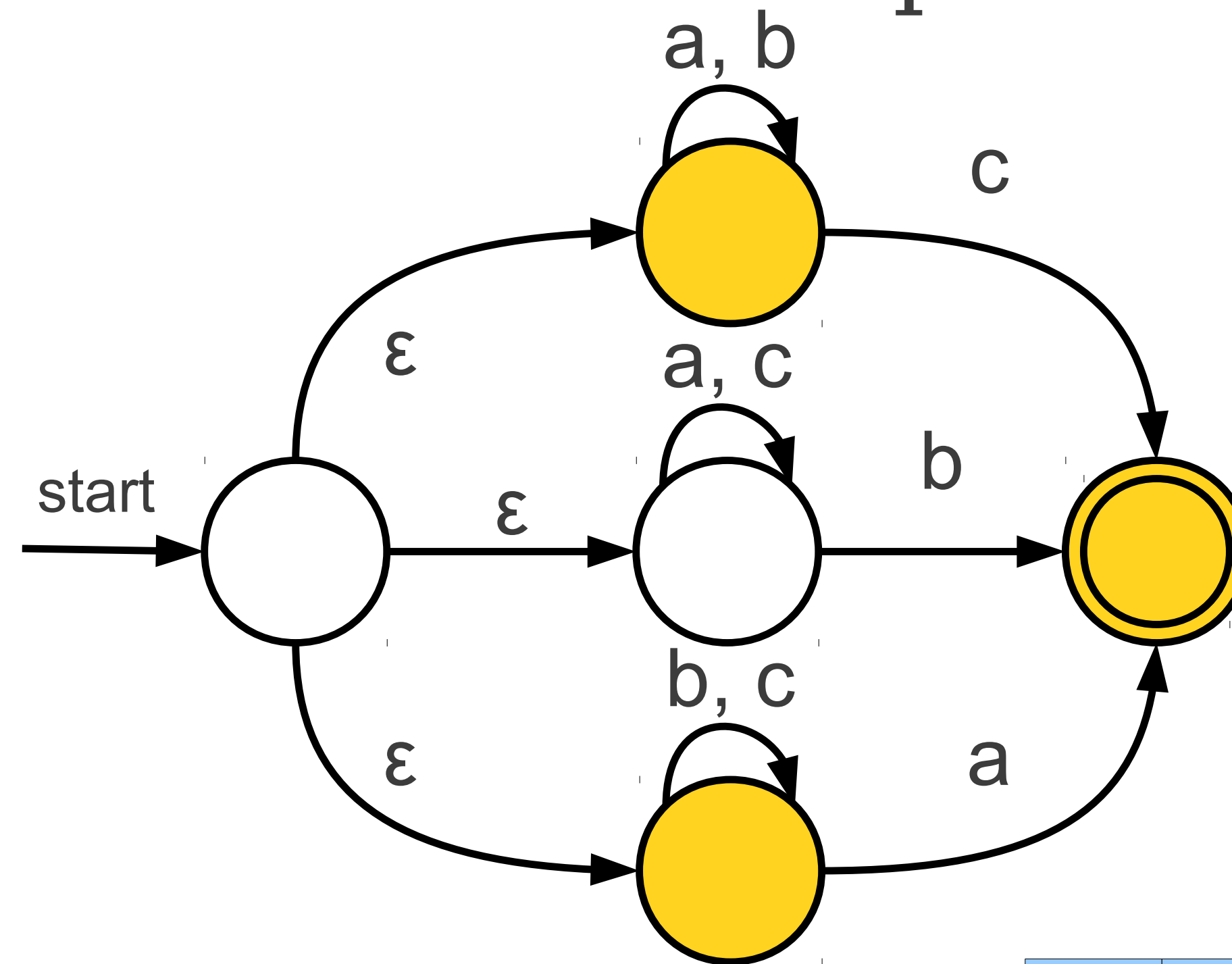




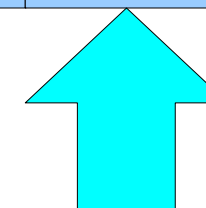
# An Even More Complex Automaton



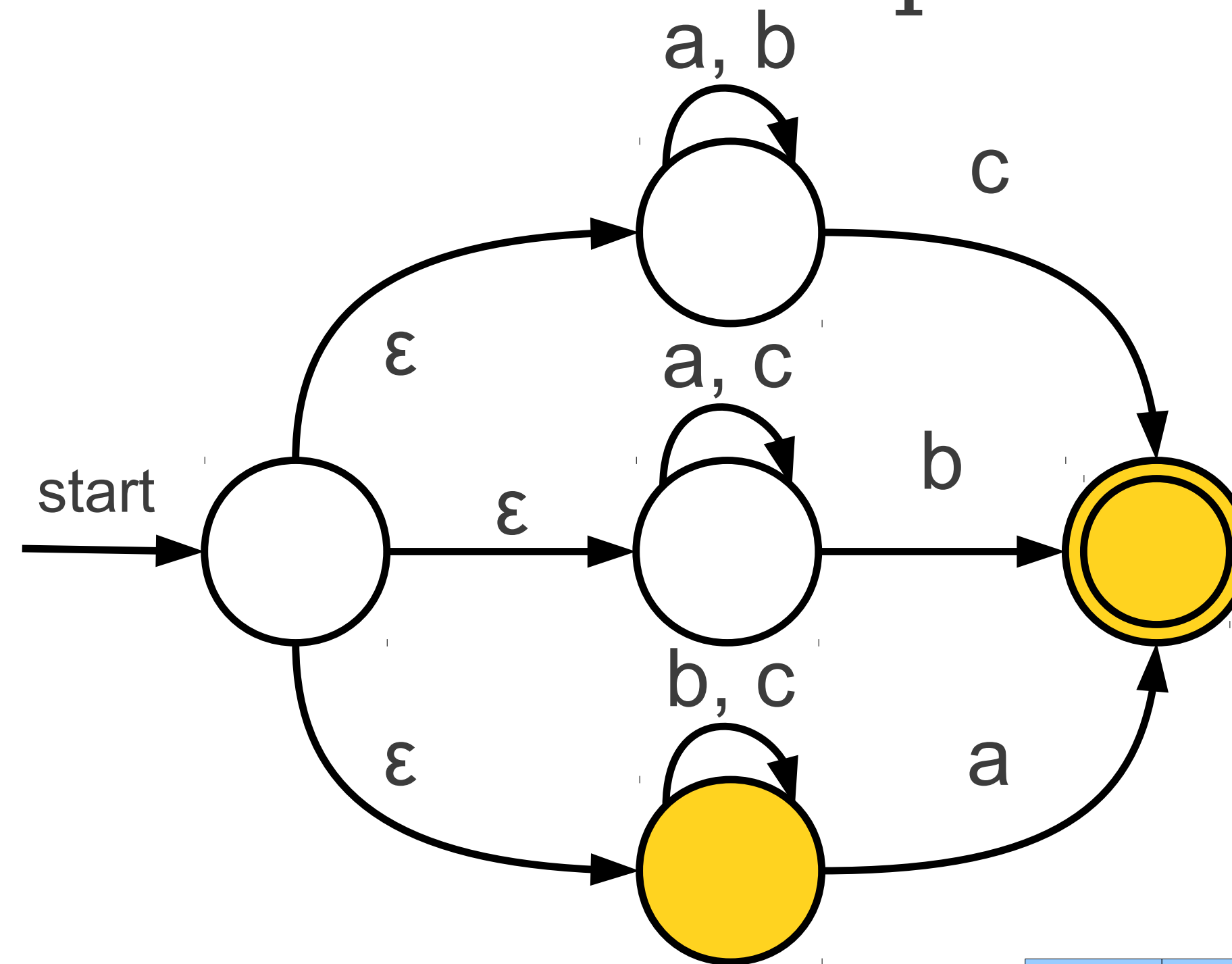
# An Even More Complex Automaton



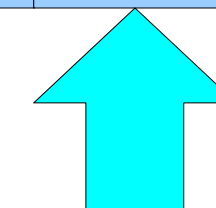
|   |   |   |   |
|---|---|---|---|
| b | c | b | a |
|---|---|---|---|



# An Even More Complex Automaton

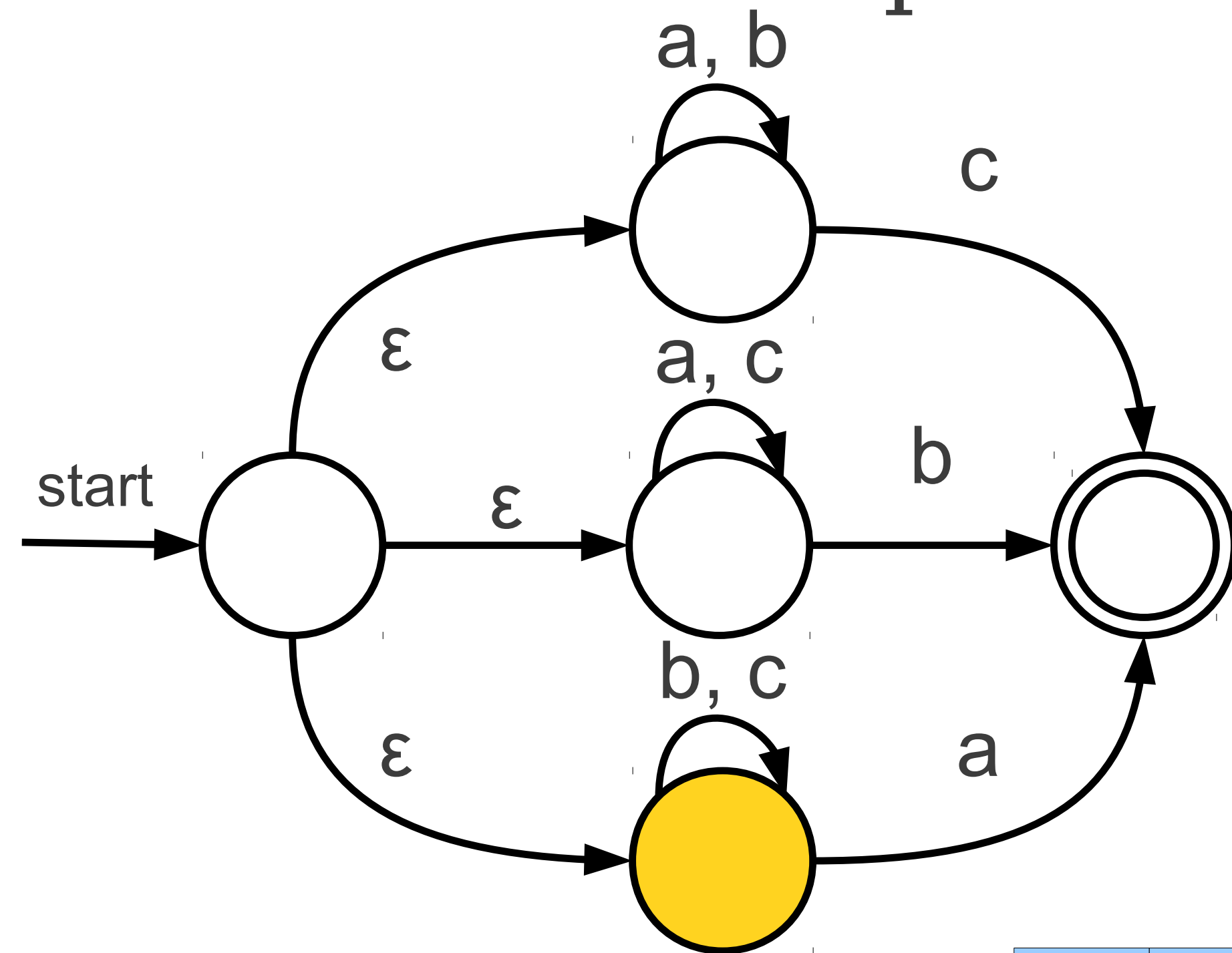


|   |   |   |   |
|---|---|---|---|
| b | c | b | a |
|---|---|---|---|

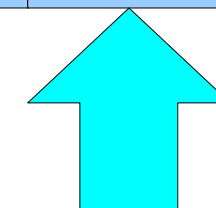




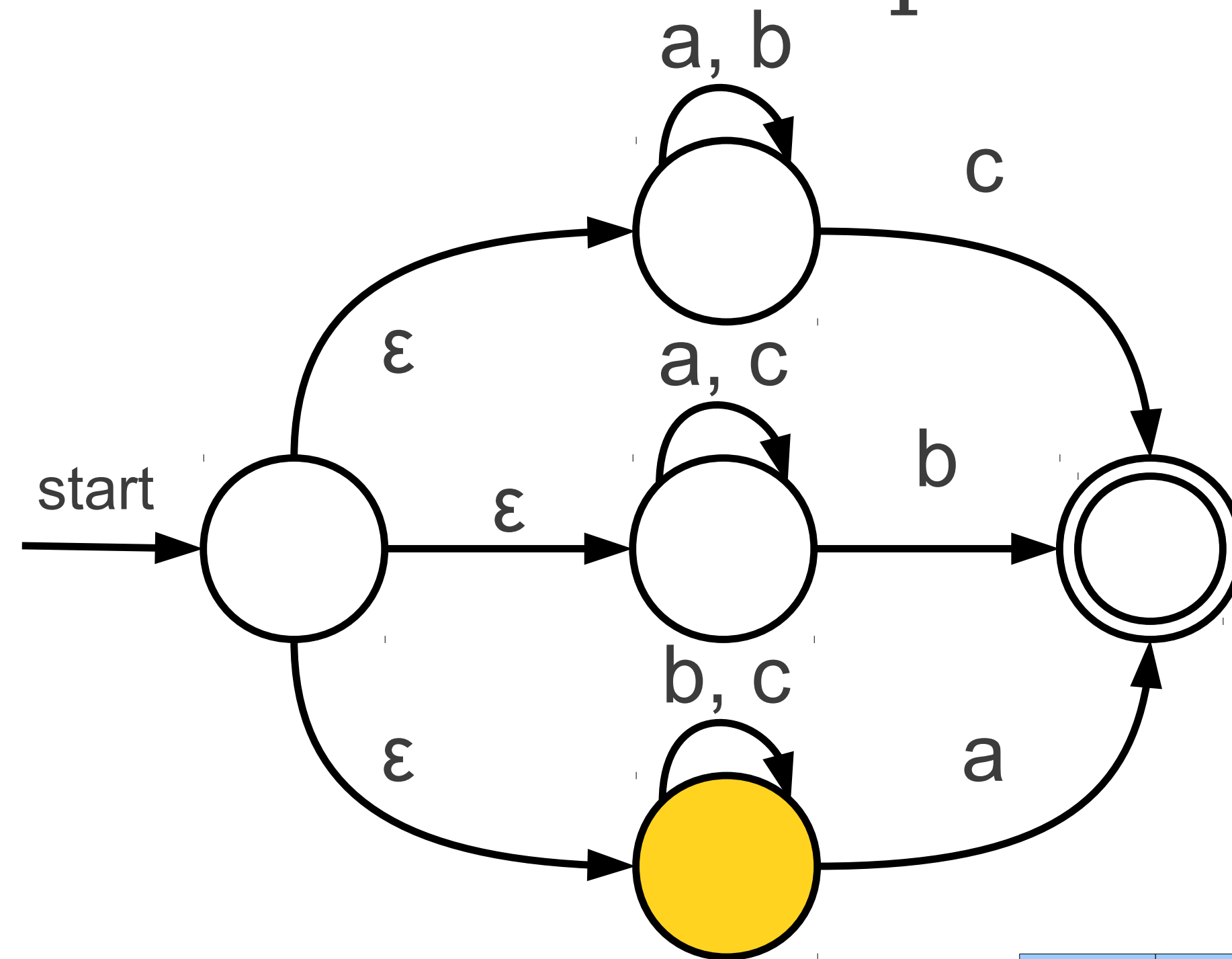
# An Even More Complex Automaton



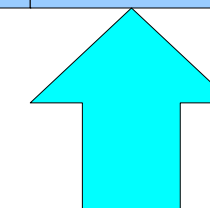
|   |   |   |   |
|---|---|---|---|
| b | c | b | a |
|---|---|---|---|



# An Even More Complex Automaton

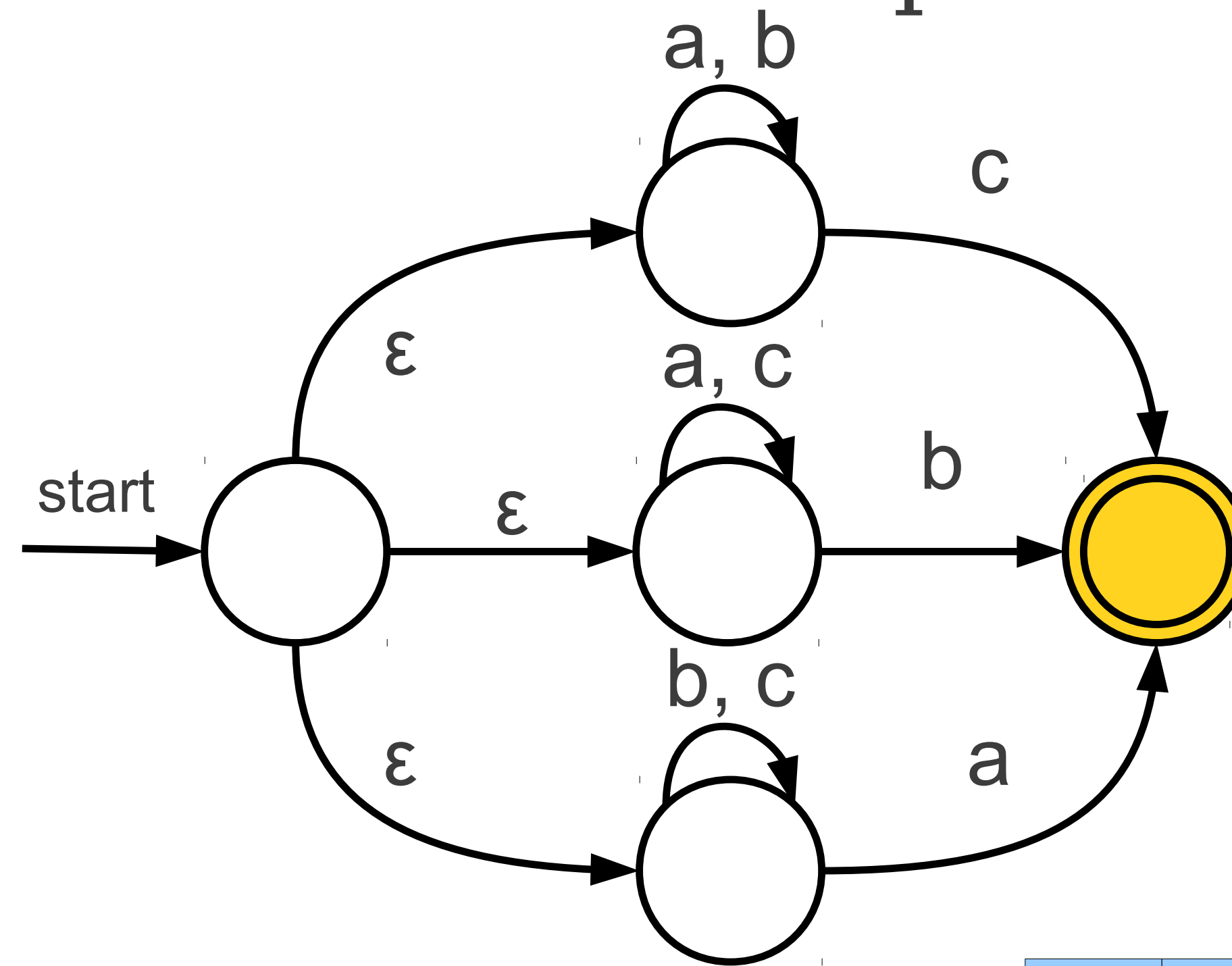


|   |   |   |   |
|---|---|---|---|
| b | c | b | a |
|---|---|---|---|

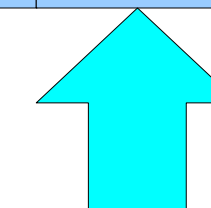




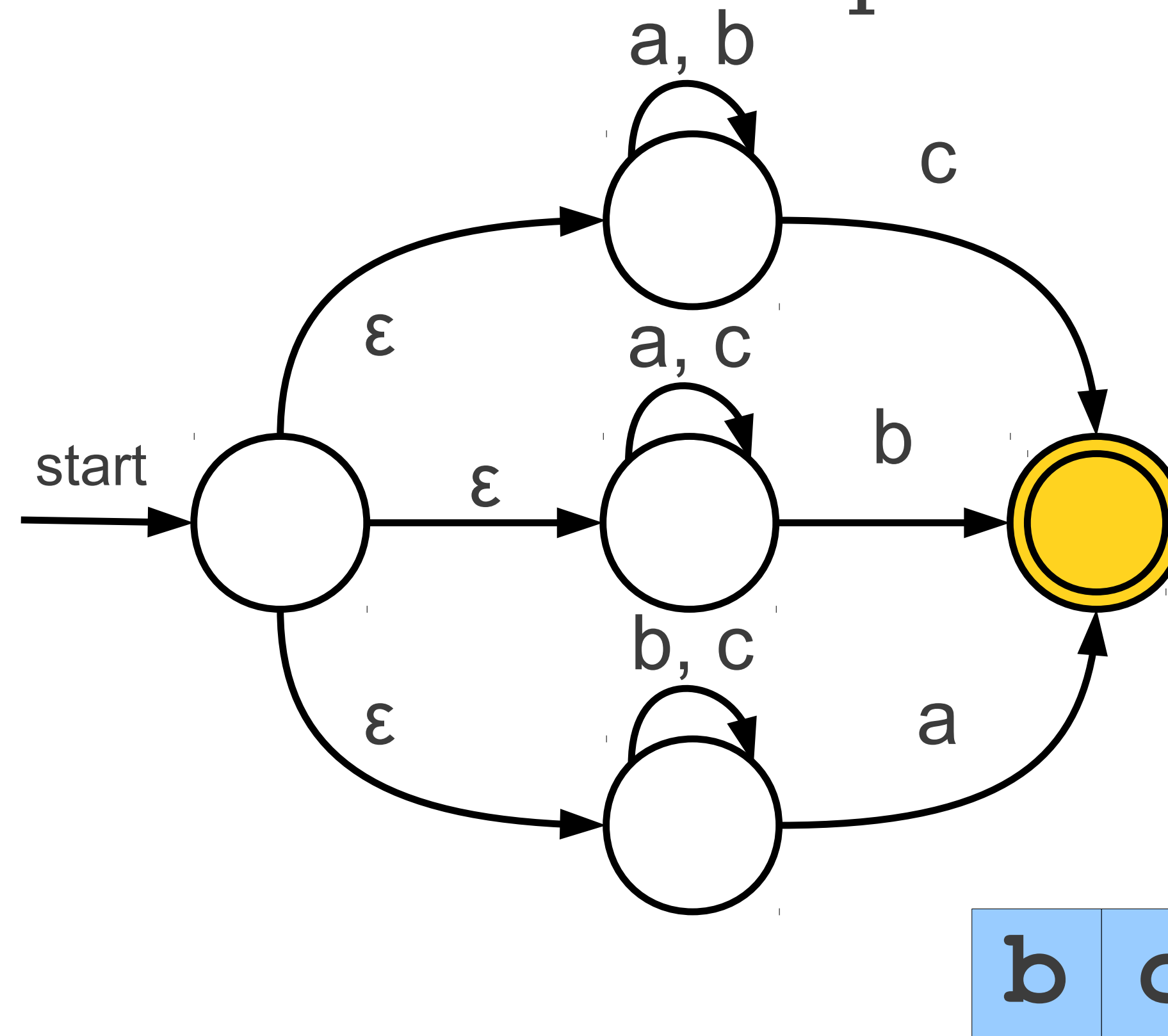
# An Even More Complex Automaton



|   |   |   |   |
|---|---|---|---|
| b | c | b | a |
|---|---|---|---|



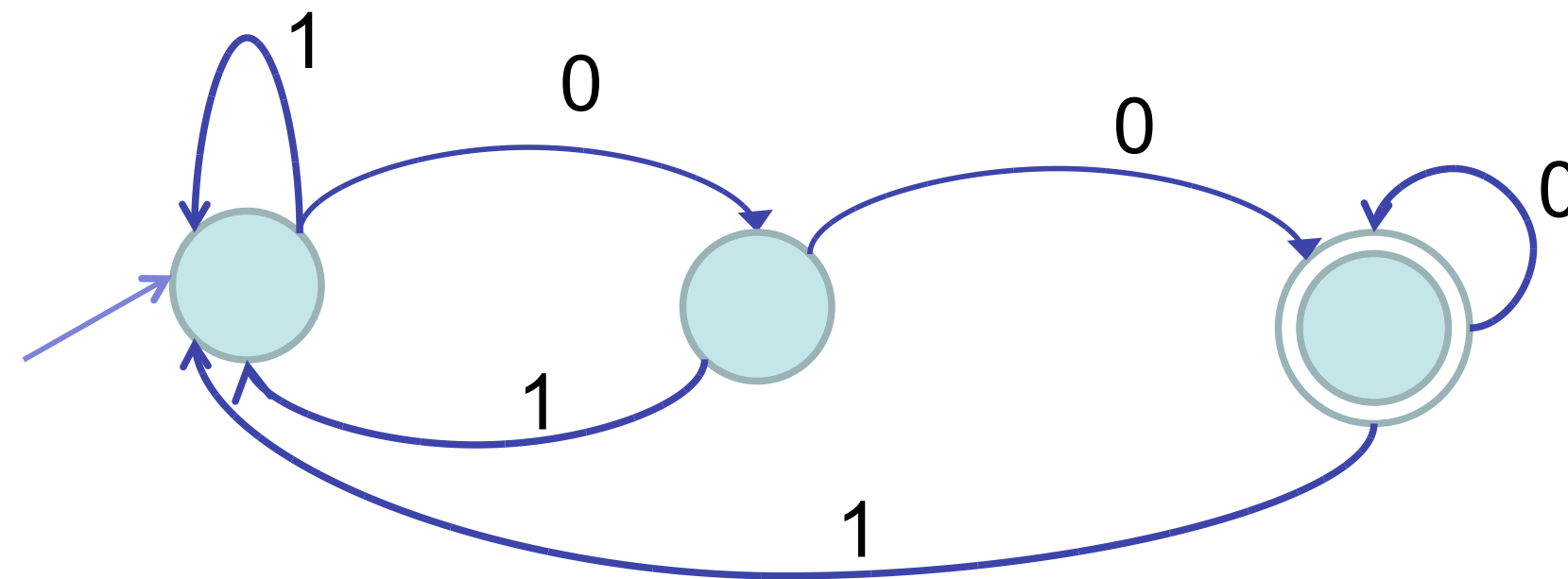
# An Even More Complex Automaton



# Finite State Automata

What language does this FA recognize?

$\Sigma = \{0,1\}$

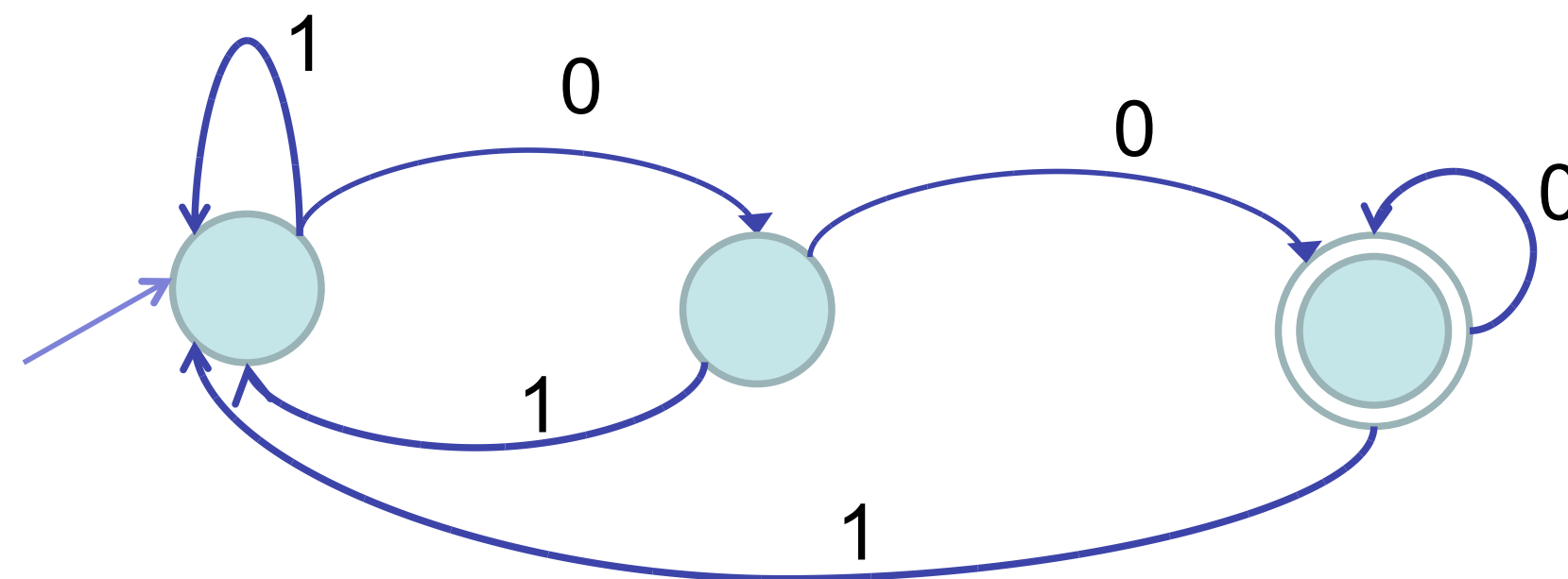




# Finite State Automata

What language does this FA recognize?

$\Sigma = \{0,1\}$



$(0|1)^*00$

I.e., bitstrings ending in at least two 0s

# Non-determinism

NFAs use "angelic" non-determinism, meaning we non-deterministically branch and if any of the branches succeeds, we succeed.

Think of it as we have an "angel" or "oracle" who will look into the future and tell us what the best choice to make is, if there is one.

Unfortunately, not yet supported by current hardware. Need to **simulate** this instead.

# DFA vs. NFA

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No  $\epsilon$ -moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have  $\epsilon$ -moves



# DFA vs. NFA

- NFAs and DFAs recognize the same set of languages (regular languages)
  - For a given NFA, there exists a DFA, and vice versa
- DFAs are faster to execute
  - There are no choices to consider
  - Tradeoff: simplicity
    - For a given language DFA can be exponentially larger than NFA.

# Automating Lexical Analyzer (scanner) Construction

To convert a specification into code:

- 1 Write down the RE for the input language
- 2 Build a big NFA
- 3 Build the DFA that simulates the NFA
- 4 Systematically shrink the DFA
- 5 Turn it into code

Scanner generators

- Lex and Flex work along these lines
- Algorithms are well-known and well-understood



# Alternative Approaches

- We'll go through the "classic" procedure above but some scanners use different approaches:
- Brzozowski: use the "derivative" operation on languages to directly produce a DFA from a regexp
- Advantage: simple to implement, extends easily to support regex conjunction, negation. Often used for regex interpreters
- Disadvantage: computationally expensive to generate minimal DFAs

# Automating Lexical Analyzer (scanner) Construction

**RE  $\rightarrow$  NFA** (*Thompson's construction*)

- Build an NFA for each term
- Combine them with  $\epsilon$ -moves

**NFA  $\rightarrow$  DFA** (*subset construction*)

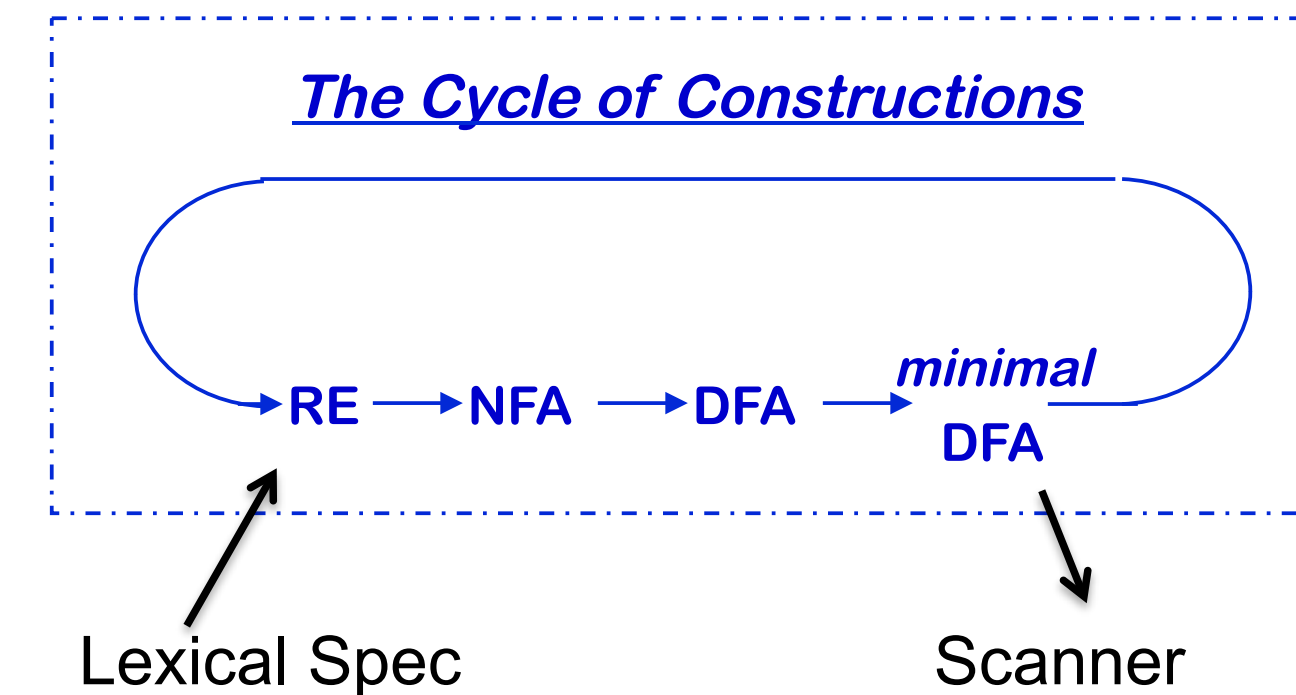
- Build the simulation

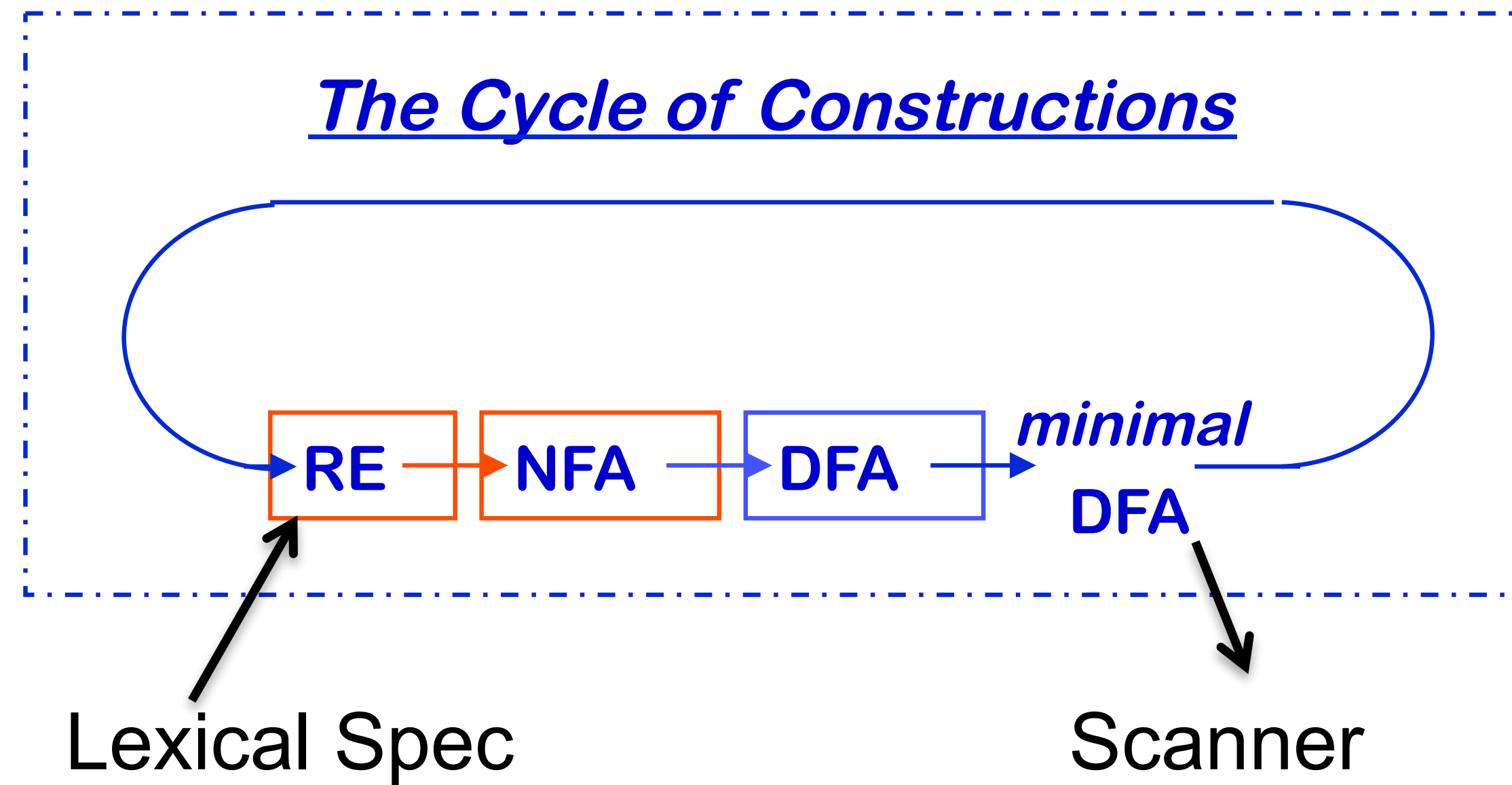
**DFA  $\rightarrow$  Minimal DFA**

- Hopcroft's algorithm

**DFA  $\rightarrow$  RE** (*Not part of the scanner construction*)

- All pairs, all paths problem
- Take the union of all paths from  $s_0$  to an accepting state







# RE $\rightarrow$ NFA using Thompson's Construction

---

Key idea

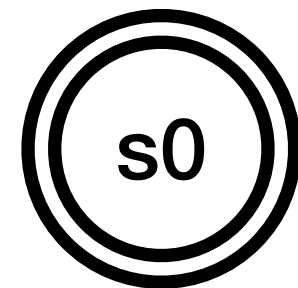
- NFA pattern for each symbol & each operator
- Join them with  $\varepsilon$  moves in precedence order

# RE $\rightarrow$ NFA using Thompson's Construction

---

Key idea

- NFA pattern for each symbol & each operator
- Join them with  $\varepsilon$  moves in precedence order



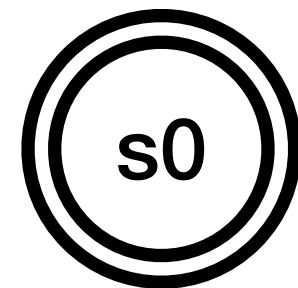
epsilon regex

# RE $\rightarrow$ NFA using Thompson's Construction

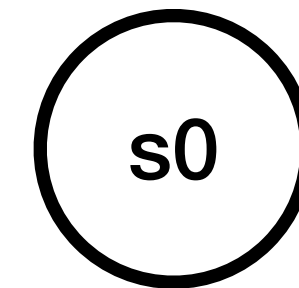
---

Key idea

- NFA pattern for each symbol & each operator
- Join them with  $\varepsilon$  moves in precedence order



epsilon regex



empty regex

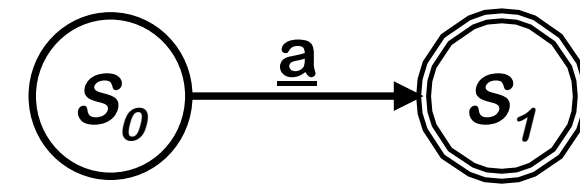


# RE $\rightarrow$ NFA using Thompson's Construction

---

Key idea

- NFA pattern for each symbol & each operator
- Join them with  $\epsilon$  moves in precedence order



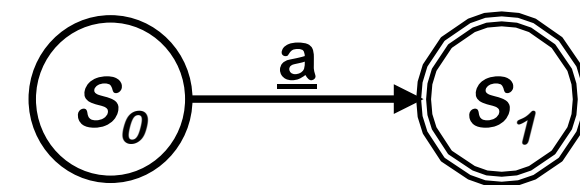
NFA for a

# RE $\rightarrow$ NFA using Thompson's Construction

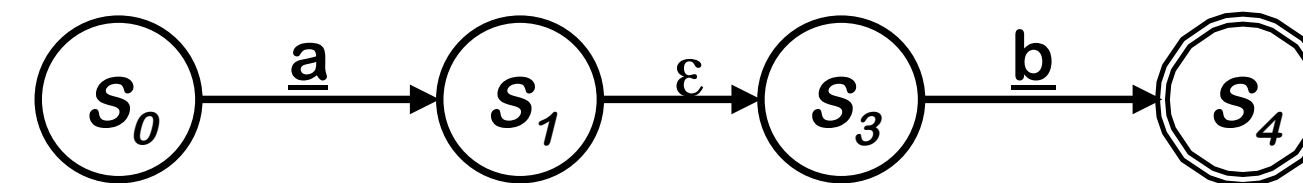
---

Key idea

- NFA pattern for each symbol & each operator
- Join them with  $\epsilon$  moves in precedence order



NFA for a



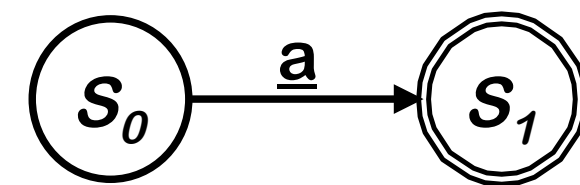
NFA for ab



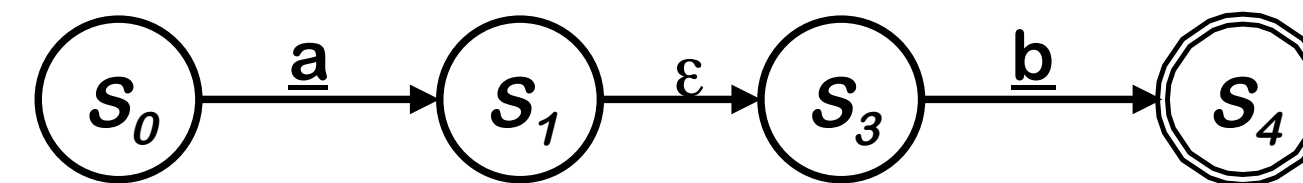
# RE $\rightarrow$ NFA using Thompson's Construction

Key idea

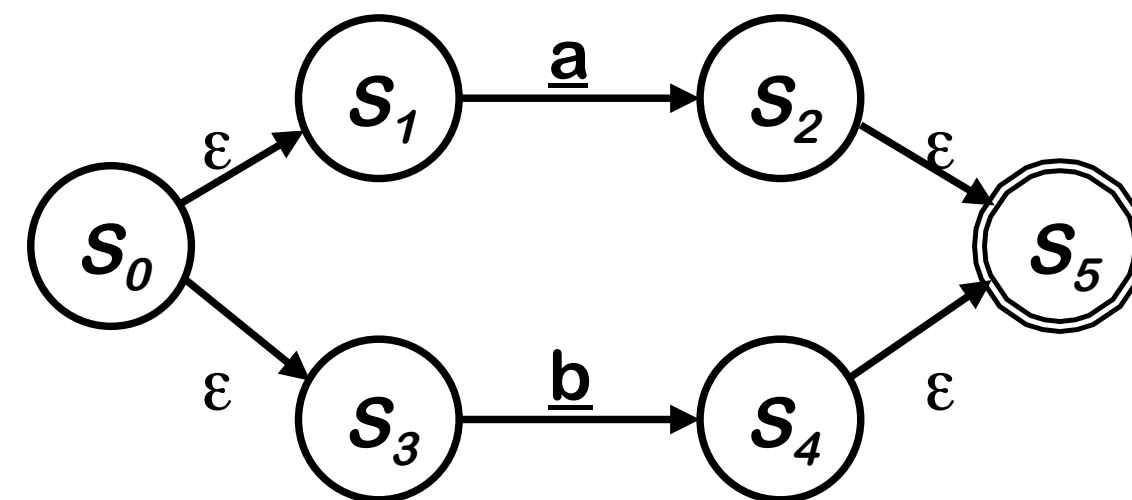
- NFA pattern for each symbol & each operator
- Join them with  $\epsilon$  moves in precedence order



NFA for a



NFA for ab

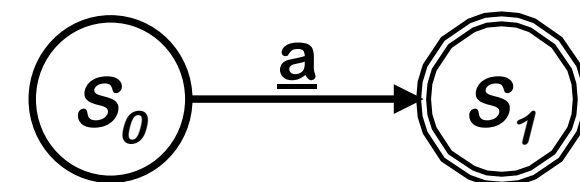


NFA for a | b

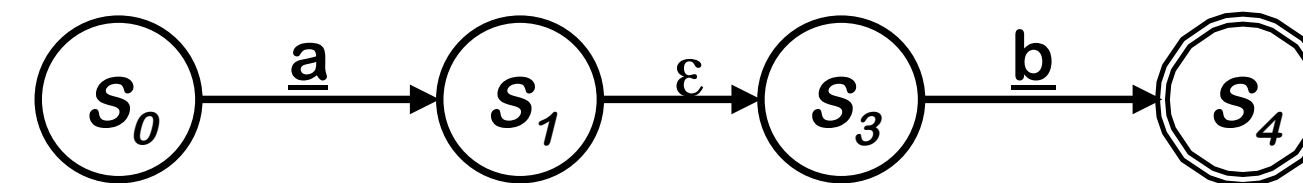
# RE $\rightarrow$ NFA using Thompson's Construction

Key idea

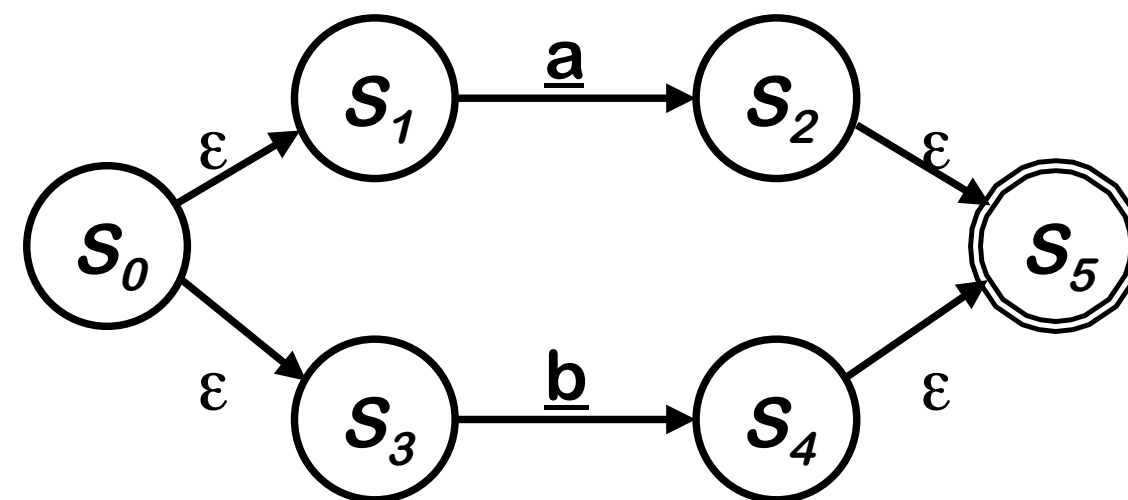
- NFA pattern for each symbol & each operator
- Join them with  $\epsilon$  moves in precedence order



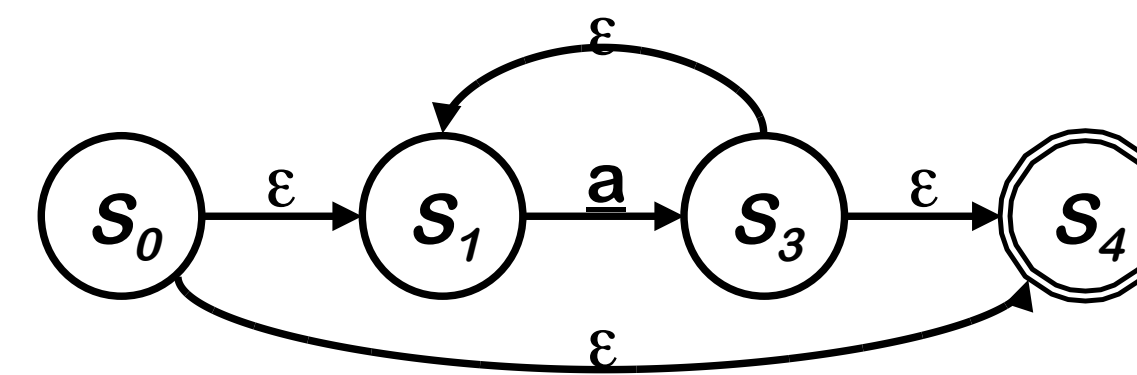
NFA for a



NFA for ab



NFA for a | b



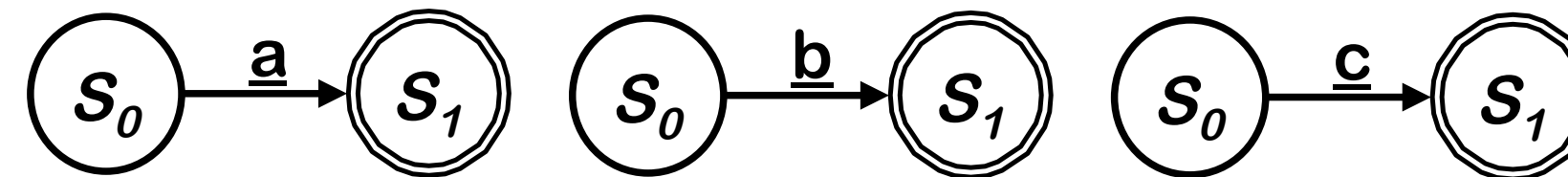
NFA for a\*

Ken Thompson, CACM, 1968

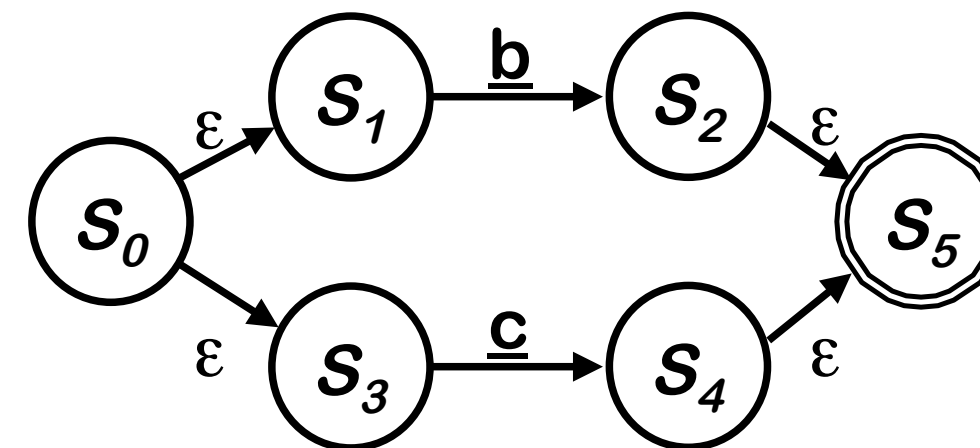
# Example of Thompson's Construction

Let's try  $a ( \underline{b} \mid \underline{c} )^*$

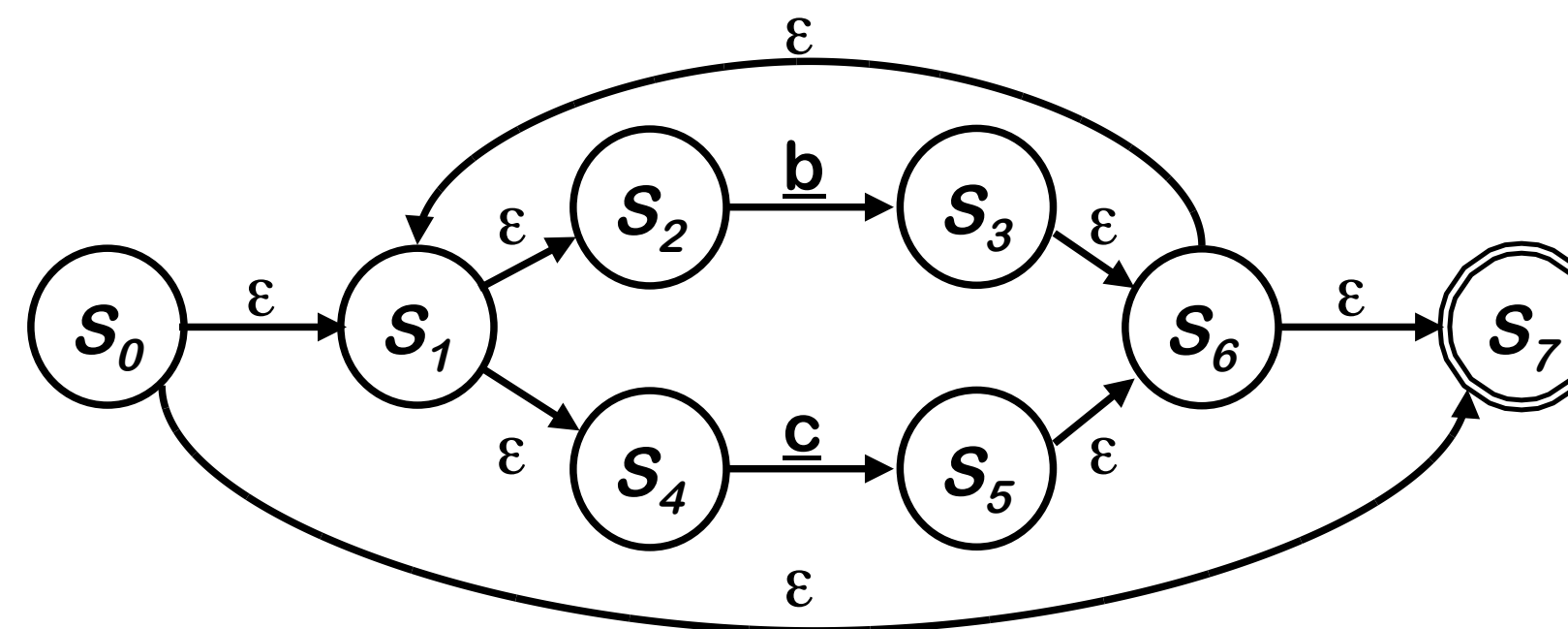
1.  $\underline{a}$ ,  $\underline{b}$ , &  $\underline{c}$



2.  $\underline{b} \mid \underline{c}$



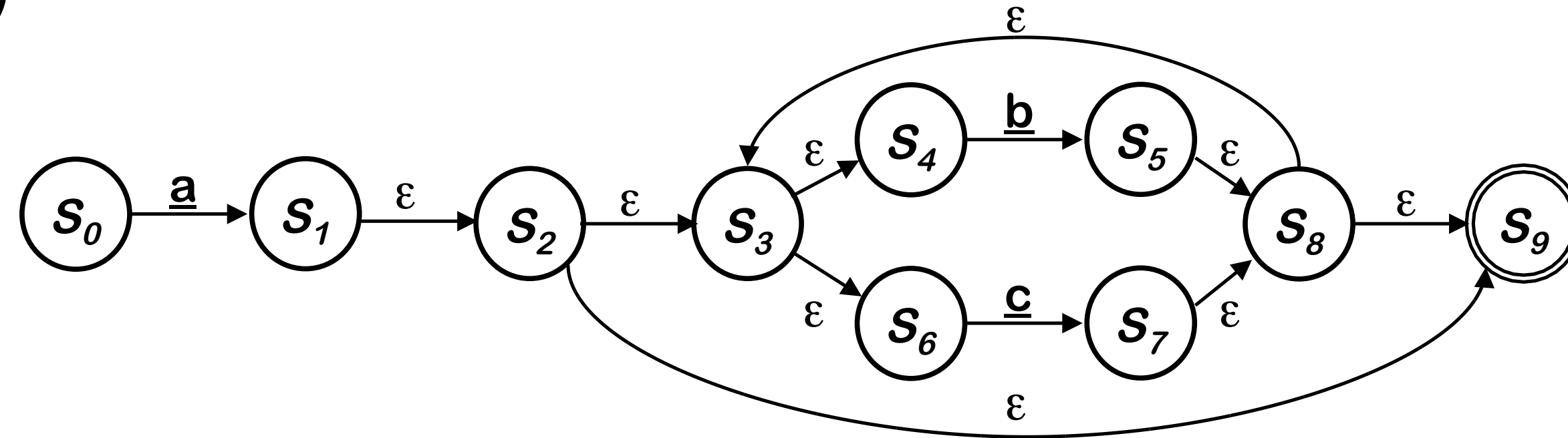
3.  $( \underline{b} \mid \underline{c} )^*$





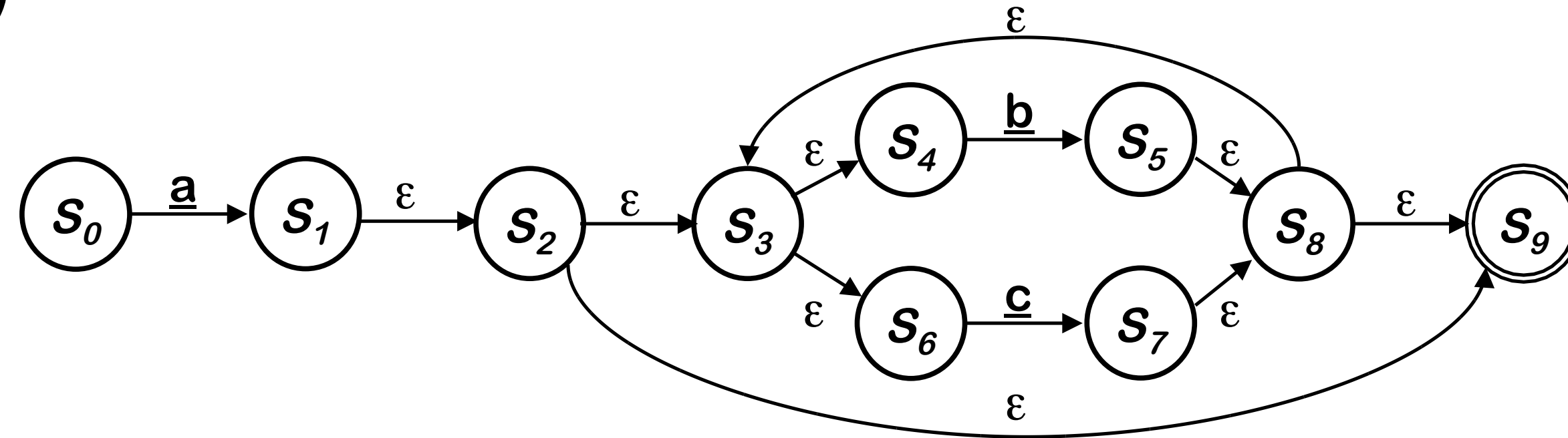
# Example of Thompson's Construction *(con't)*

4.  $\underline{a} (\underline{b} | \underline{c})^*$



# Example of Thompson's Construction *(con't)*

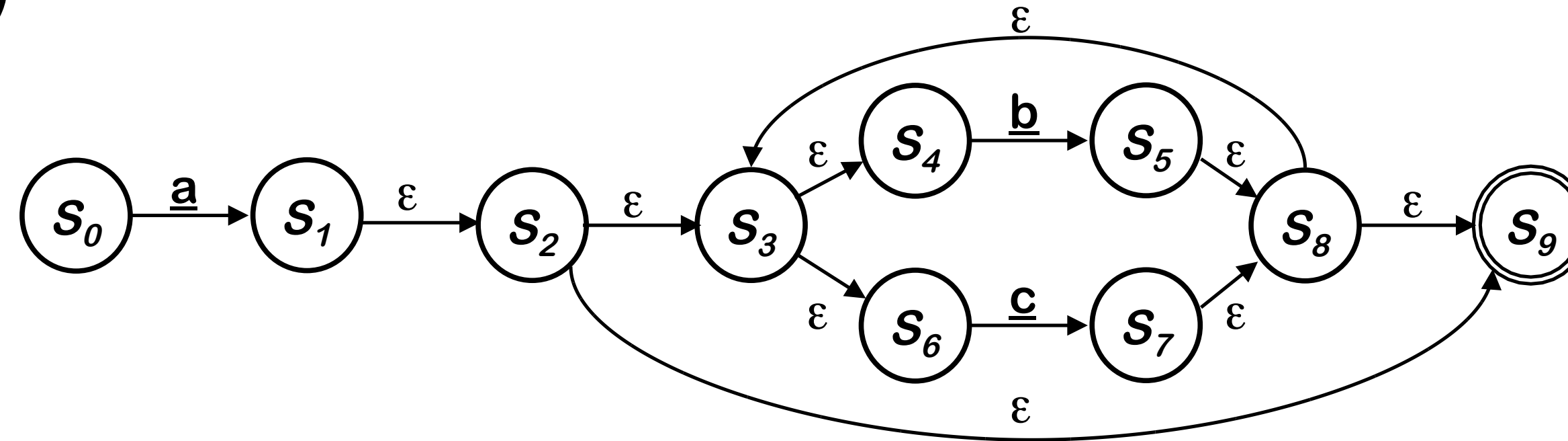
4.  $\underline{a} (\underline{b} | \underline{c})^*$



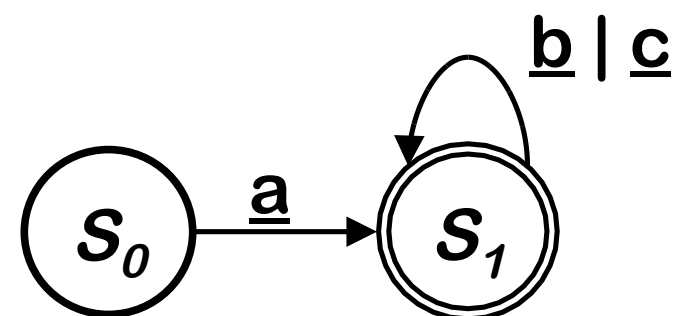
Of course, a human would design something simpler ...

# Example of Thompson's Construction *(con't)*

4.  $\underline{a} (\underline{b} | \underline{c})^*$

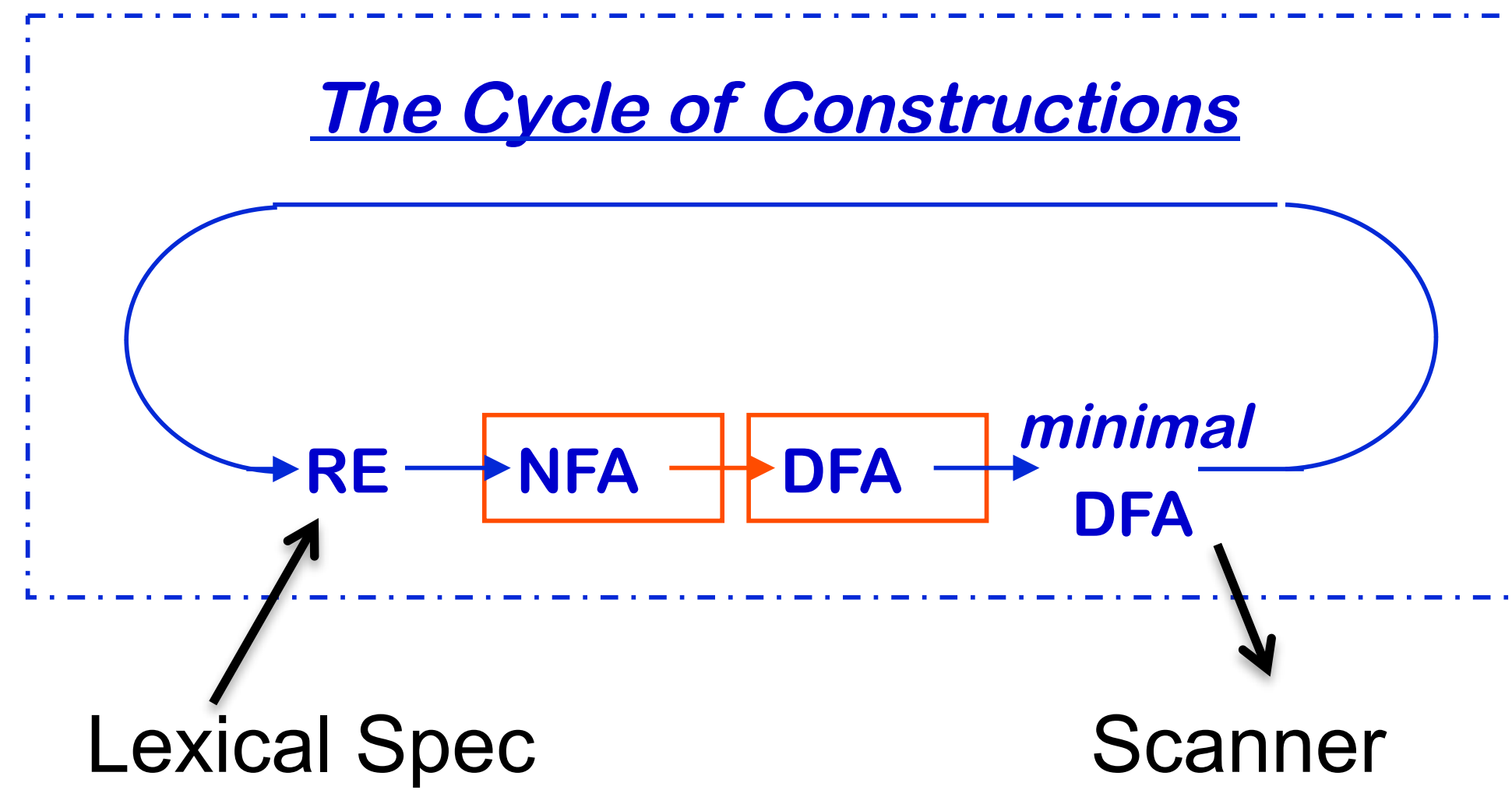


Of course, a human would design something simpler ...



But, we can automate production of the more complex one ...





# NFA to DFA : Trick

---

- Simulate the NFA
- Each state of DFA
  - = a non-empty subset of states of the NFA
- Start state
  - = the set of NFA states reachable through e-moves from NFA start state
- Add a transition  $S \xrightarrow{a} S'$  to DFA iff
  - $S'$  is the set of NFA states reachable from any state in  $S$  after seeing the input  $a$ , considering  $\epsilon$ -moves as well

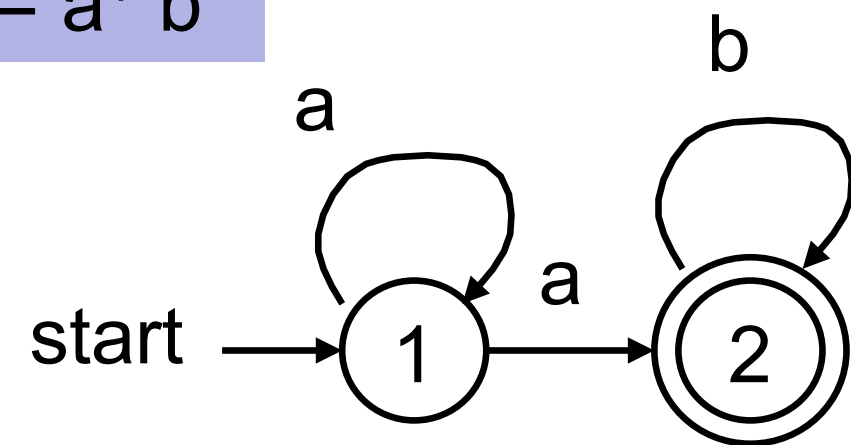


## NFA to DFA (2)

---

- Multiple transitions
  - Solve by subset construction
  - Build new DFA based upon the set of states each representing a unique subset of states in NFA

$R = a^+ b^*$



$\epsilon\text{-closure}(1) = \{1\}$  include state "1"

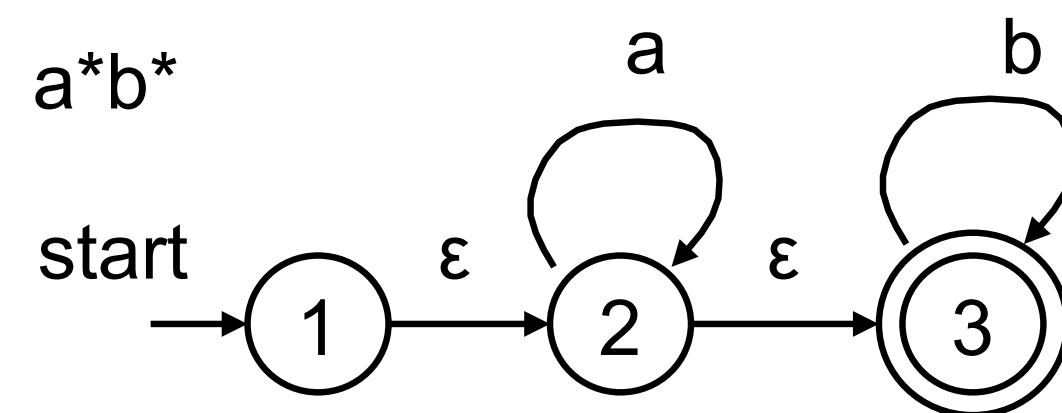
$(1, a) \rightarrow \{1, 2\}$  include state "1/2"

$(1, b) \rightarrow \text{ERROR}$

# NFA to DFA (3)

---

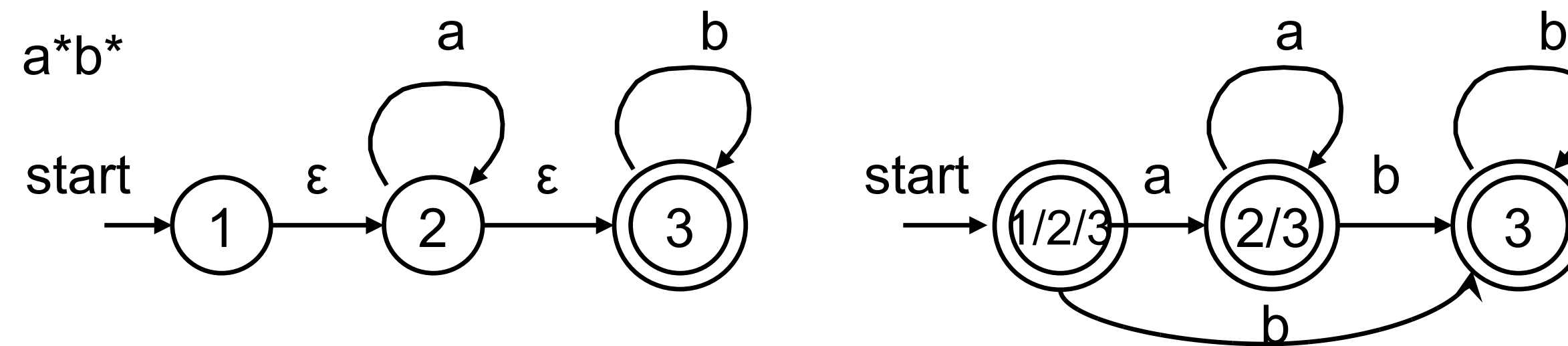
- $\epsilon$  transitions
  - Any state reachable by an  $\epsilon$  transition is “part of the state”
  - $\epsilon$ -closure - Any state reachable from S by  $\epsilon$  transitions is in the  $\epsilon$ -closure; treat  $\epsilon$ -closure as 1 big state, always include  $\epsilon$ -closure as part of the state



1.  $\epsilon$ -closure(1) = {1,2,3}; include 1/2/3
2. Move(1/2/3, a) = {2, 3} +  $\epsilon$ -closure(2,3) = {2,3} ; include 2/3
3. Move(1/2/3, b) = {3} +  $\epsilon$ -closure(3) = {3} ; include state 3
4. Move(2/3, a) = {2} +  $\epsilon$ -closure(2) = {2,3}
5. Move(2/3, b) = {3} +  $\epsilon$ -closure(3) = {3}
6. Move(3, b) = {3} +  $\epsilon$ -closure(3) = {3}

# NFA to DFA (3)

- $\epsilon$  transitions
  - Any state reachable by an  $\epsilon$  transition is “part of the state”
  - $\epsilon$ -closure - Any state reachable from S by  $\epsilon$  transitions is in the  $\epsilon$ -closure; treat  $\epsilon$ -closure as 1 big state, always include  $\epsilon$ -closure as part of the state

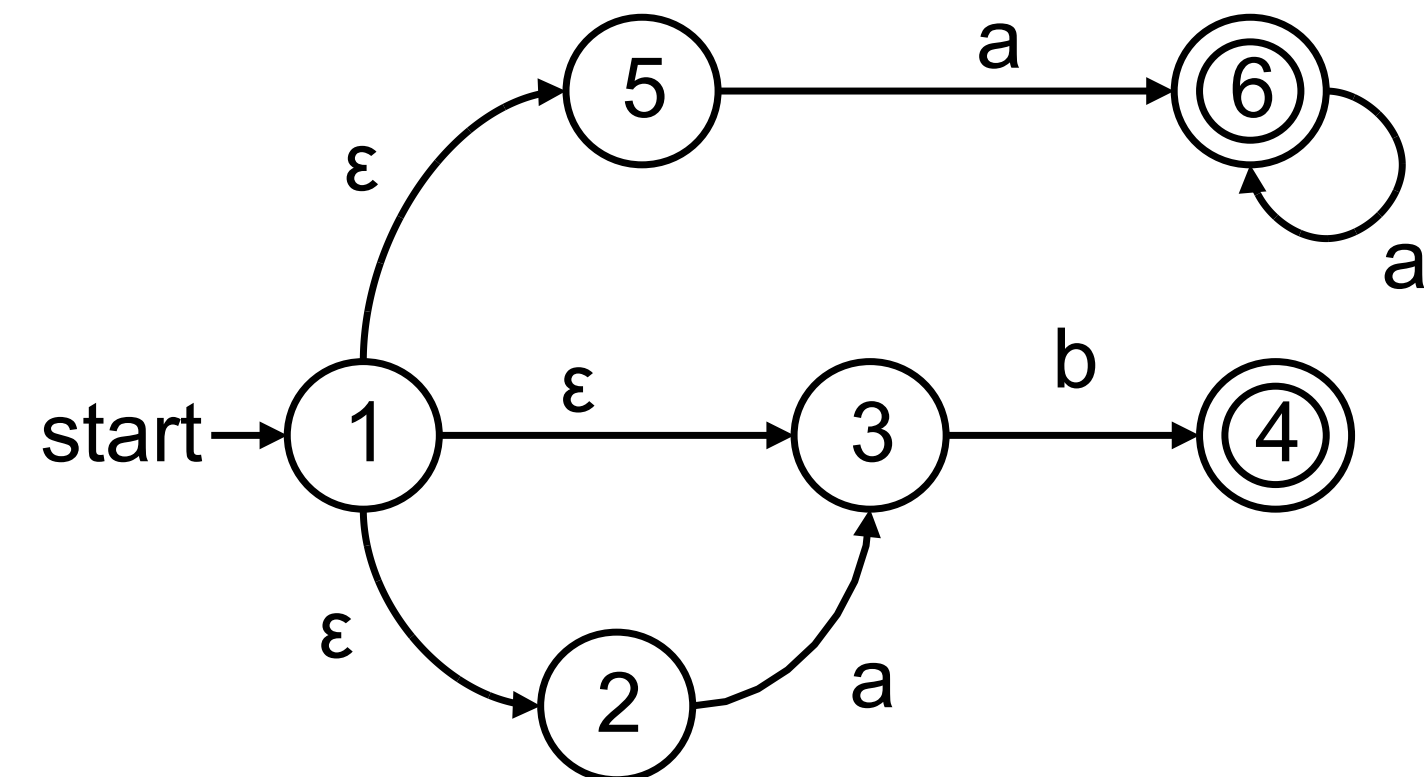


1.  $\epsilon$ -closure(1) = {1,2,3};
2. Move(1/2/3, a) = {2, 3} +  $\epsilon$ -closure(2,3) = {2,3} ; include 2/3
3. Move(1/2/3, b) = {3} +  $\epsilon$ -closure(3) = {3} ; include state 3
4. Move(2/3, a) = {2} +  $\epsilon$ -closure(2) = {2,3}
5. Move(2/3, b) = {3} +  $\epsilon$ -closure(3) = {3}
6. Move(3, b) = {3} +  $\epsilon$ -closure(3) = {3}

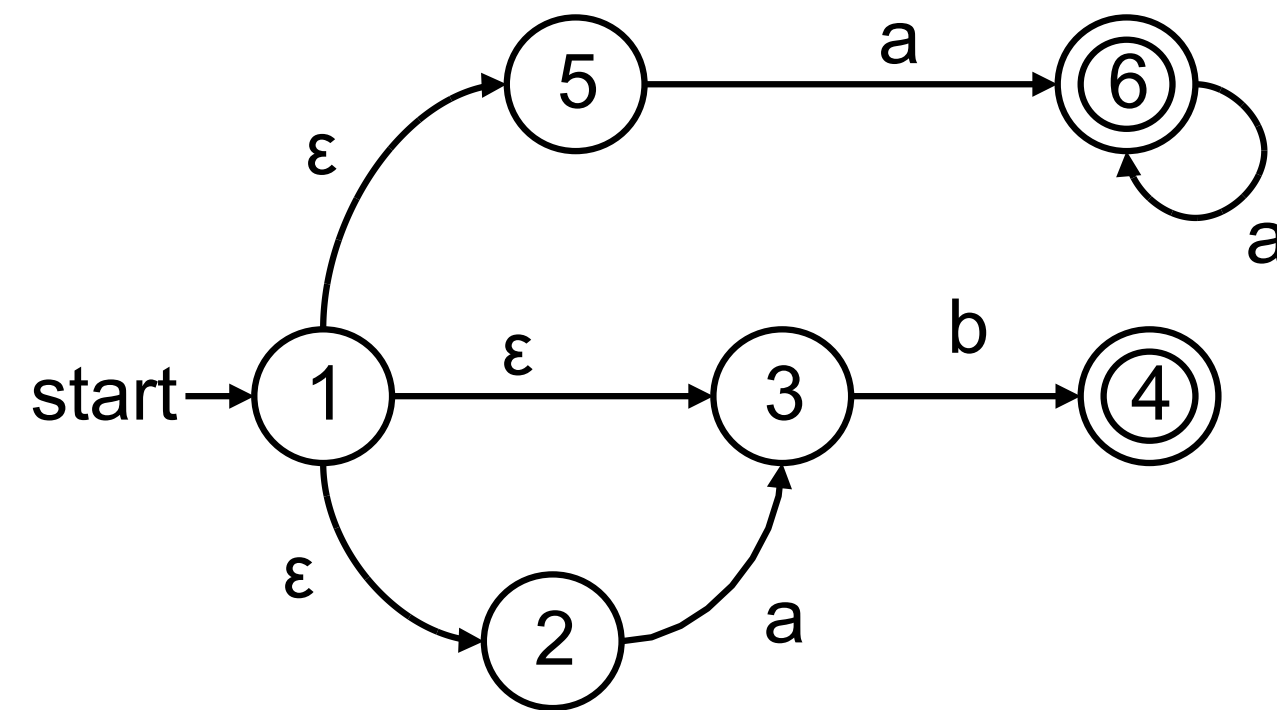


# NFA to DFA - Example

---



# NFA to DFA - Example



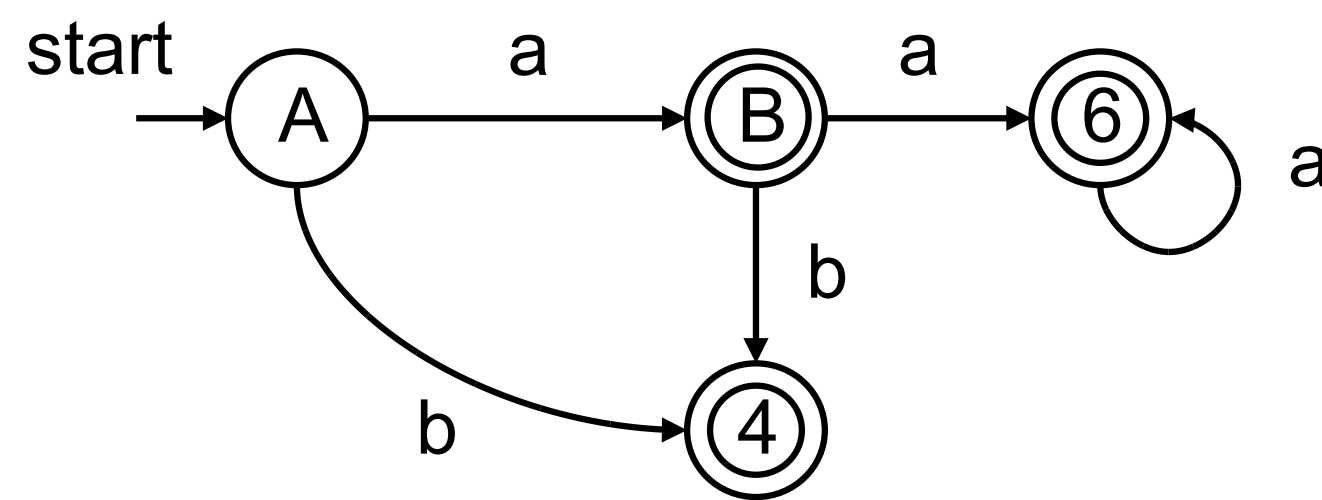
$\epsilon\text{-closure}(1) = \{1, 2, 3, 5\}$

Create a new state  $A = \{1, 2, 3, 5\}$

$\text{move}(A, a) = \{3, 6\} + \epsilon\text{-closure}(3, 6) = \{3, 6\}$

Create  $B = \{3, 6\}$

$\text{move}(A, b) = \{4\} + \epsilon\text{-closure}(4) = \{4\}$



$\text{move}(B, a) = \{6\} + \epsilon\text{-closure}(6) = \{6\}$

$\text{move}(B, b) = \{4\} + \epsilon\text{-closure}(4) = \{4\}$

$\text{move}(6, a) = \{6\} + \epsilon\text{-closure}(6) = \{6\}$

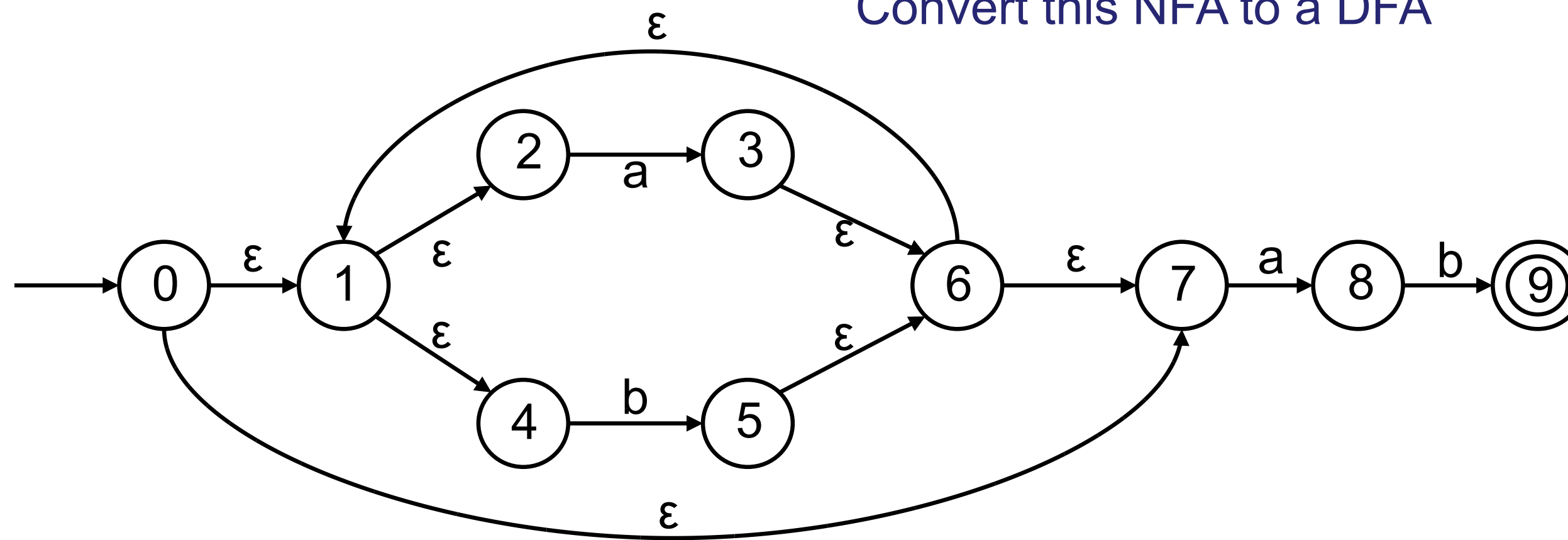
$\text{move}(6, b) \rightarrow \text{ERROR}$

$\text{move}(4, a|b) \rightarrow \text{ERROR}$

# Class Problem

---

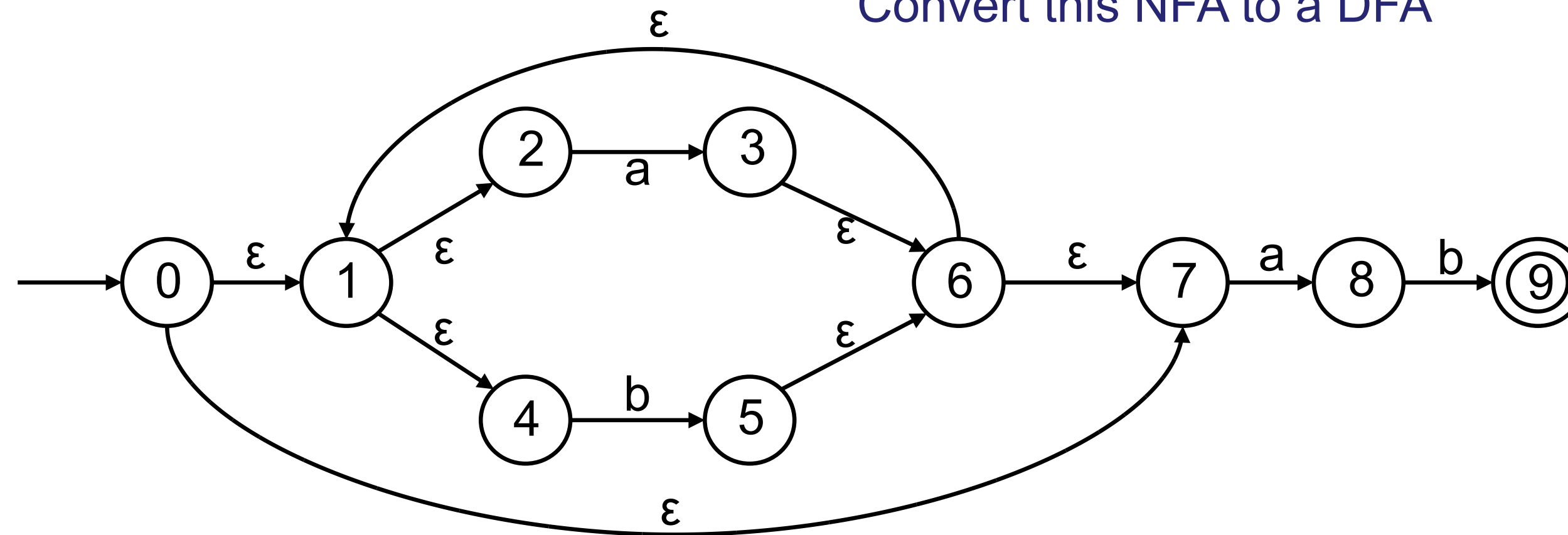
Convert this NFA to a DFA





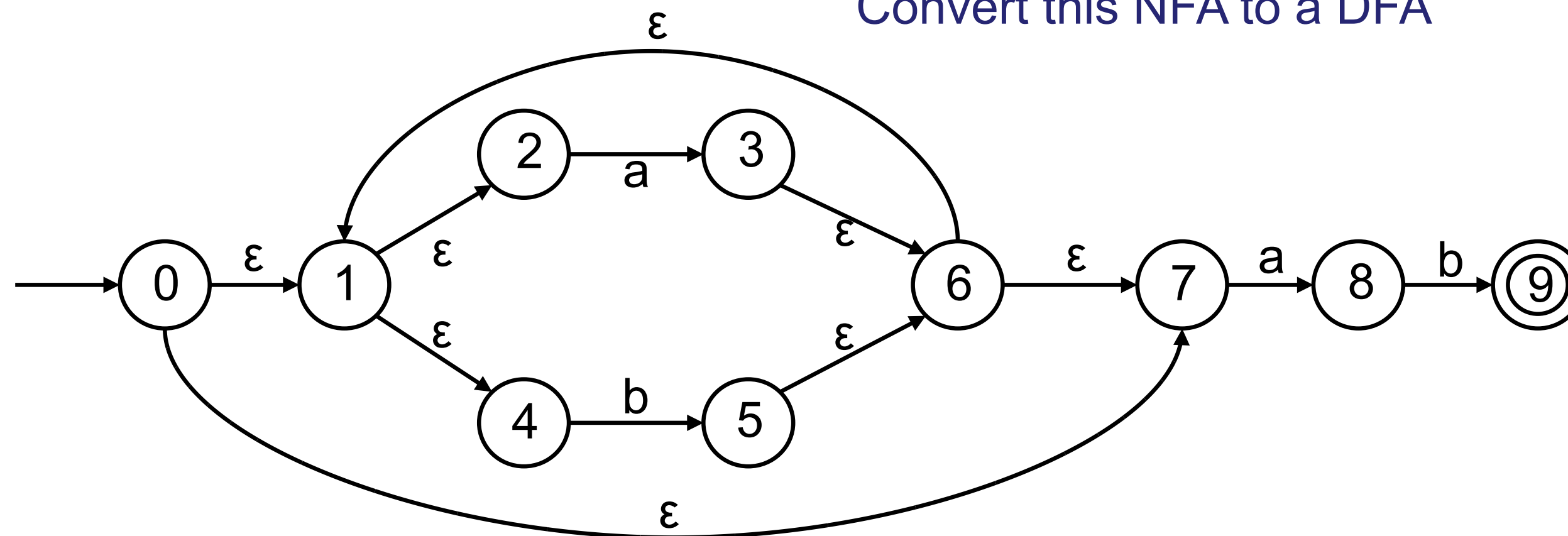
# Class Problem

Convert this NFA to a DFA



# Class Problem

Convert this NFA to a DFA

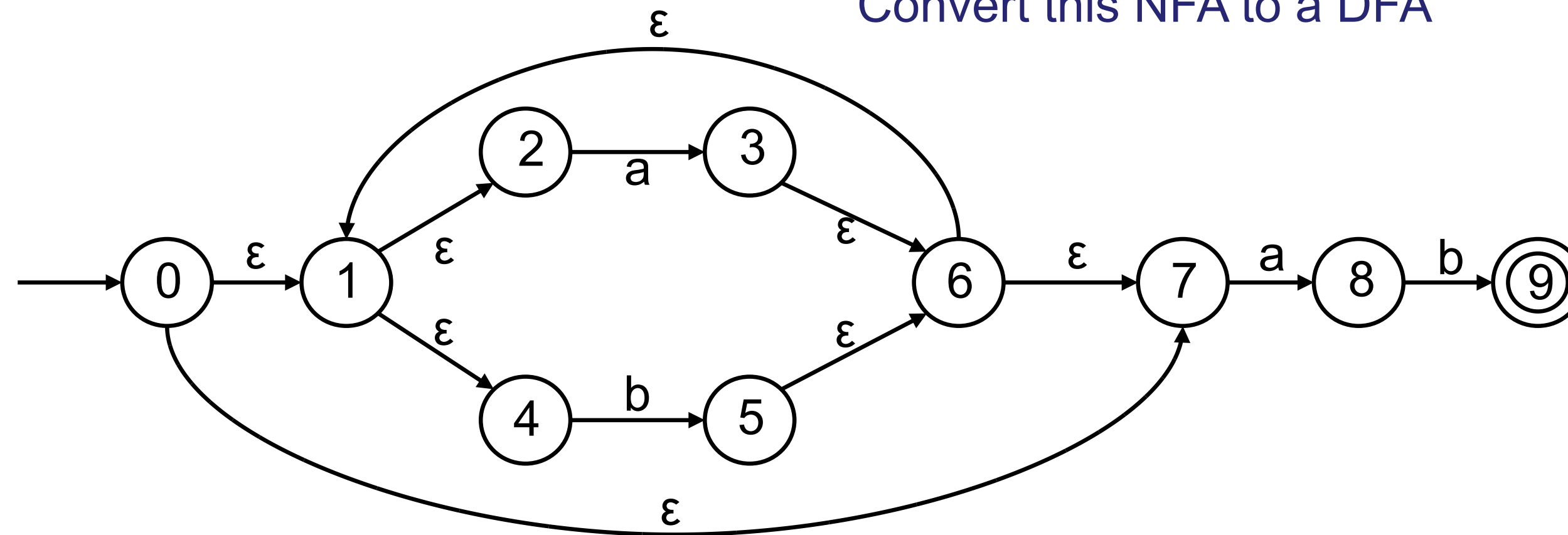


$$A = \{ 0, 1, 2, 4, 7 \}$$



# Class Problem

Convert this NFA to a DFA

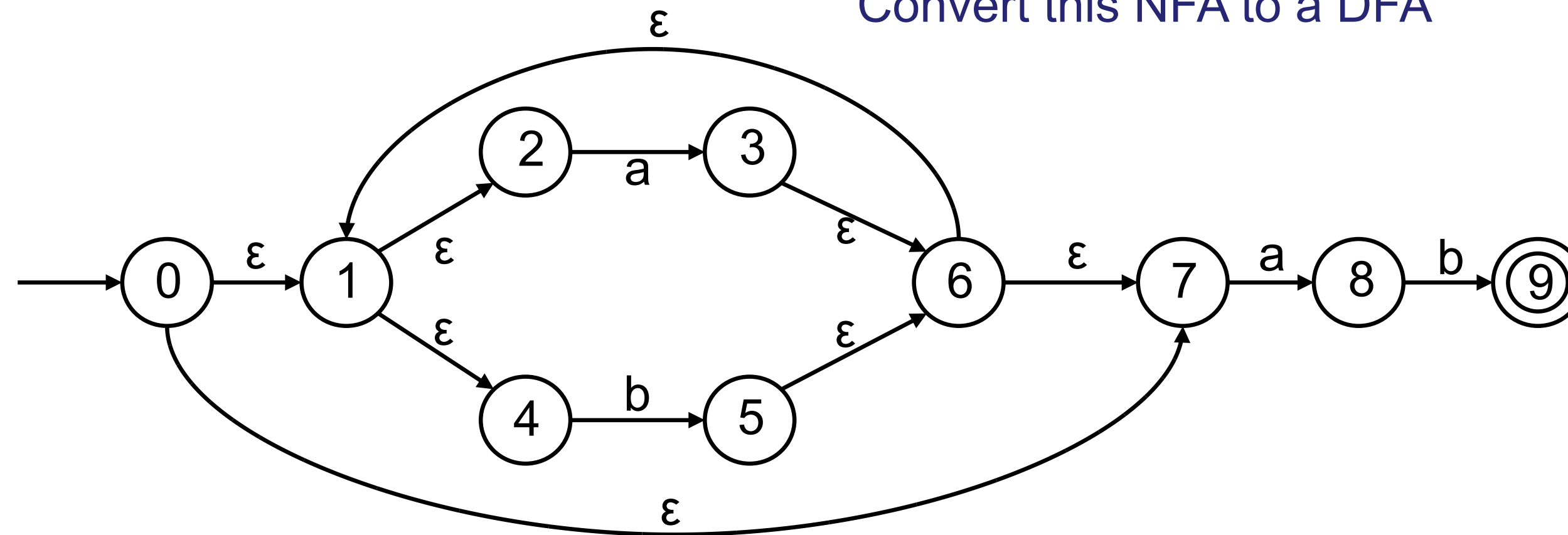


$$A = \{ 0, 1, 2, 4, 7 \}$$

$$B = \{ 1, 2, 3, 4, 6, 7, 8 \}$$

# Class Problem

Convert this NFA to a DFA



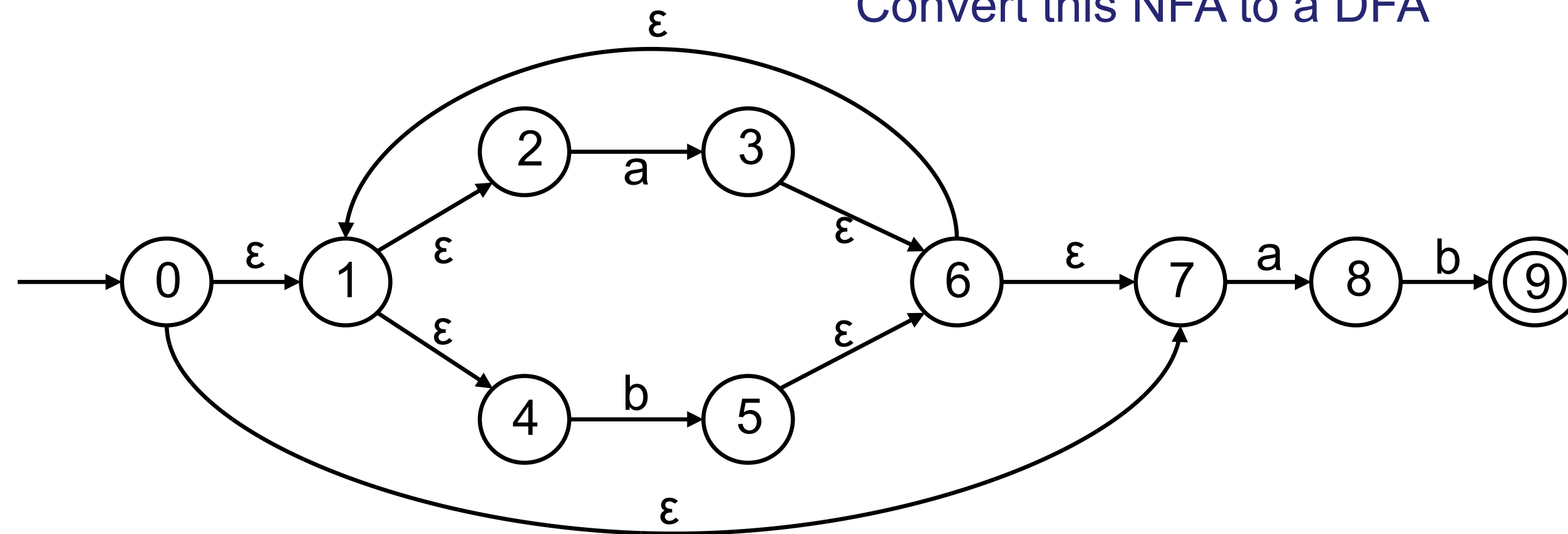
$$A = \{ 0, 1, 2, 4, 7 \}$$

$$B = \{ 1, 2, 3, 4, 6, 7, 8 \}$$

$$C = \{ 1, 2, 4, 5, 6, 7 \}$$

# Class Problem

Convert this NFA to a DFA



$$A = \{ 0, 1, 2, 4, 7 \}$$

$$B = \{ 1, 2, 3, 4, 6, 7, 8 \}$$

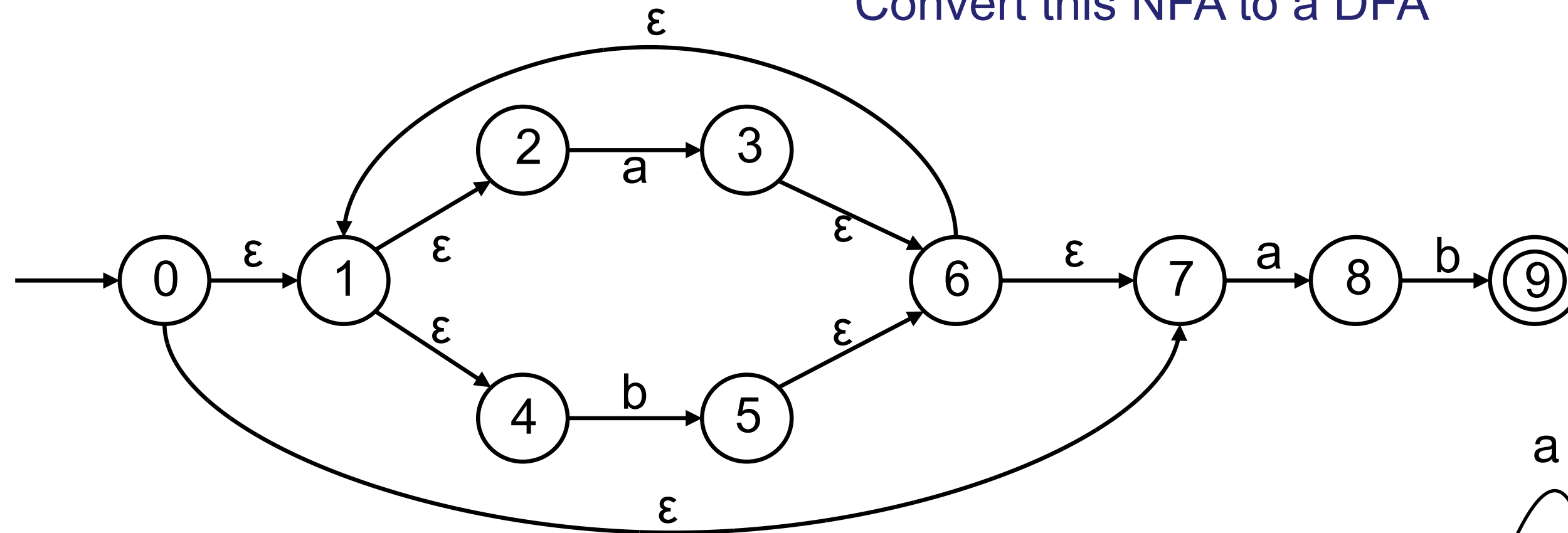
$$C = \{ 1, 2, 4, 5, 6, 7 \}$$

$$D = \{ 1, 2, 4, 5, 6, 7, 9 \}$$



# Class Problem

Convert this NFA to a DFA

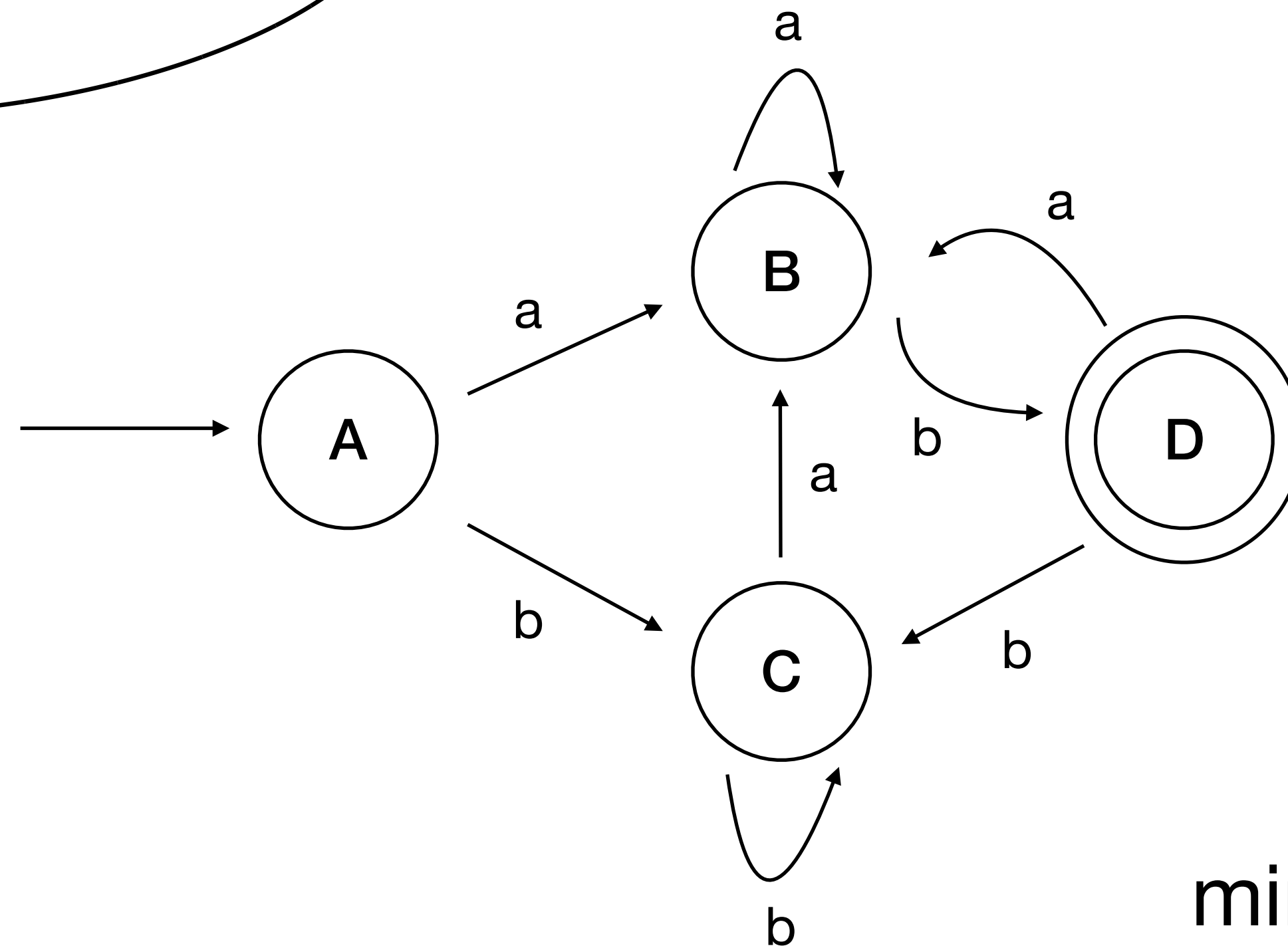


$A = \{ 0, 1, 2, 4, 7 \}$

$B = \{ 1, 2, 3, 4, 6, 7, 8 \}$

$C = \{ 1, 2, 4, 5, 6, 7 \}$

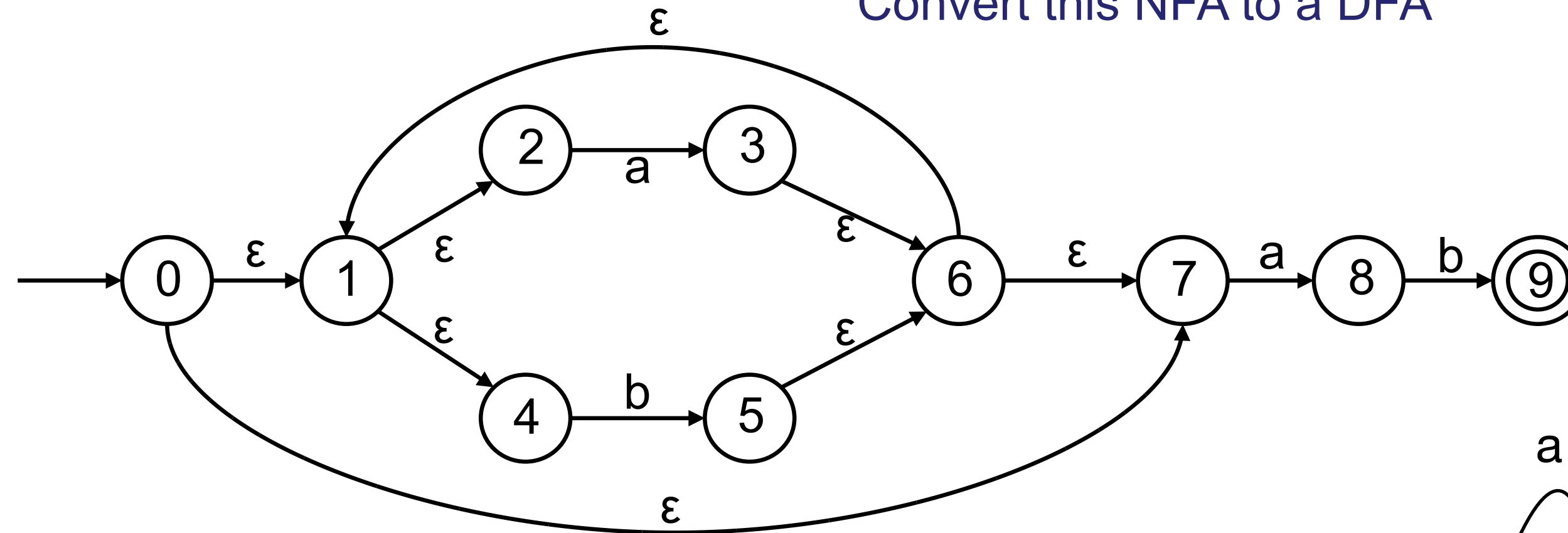
$D = \{ 1, 2, 4, 5, 6, 7, 9 \}$



minimal?

# Class Problem

Convert this NFA to a DFA

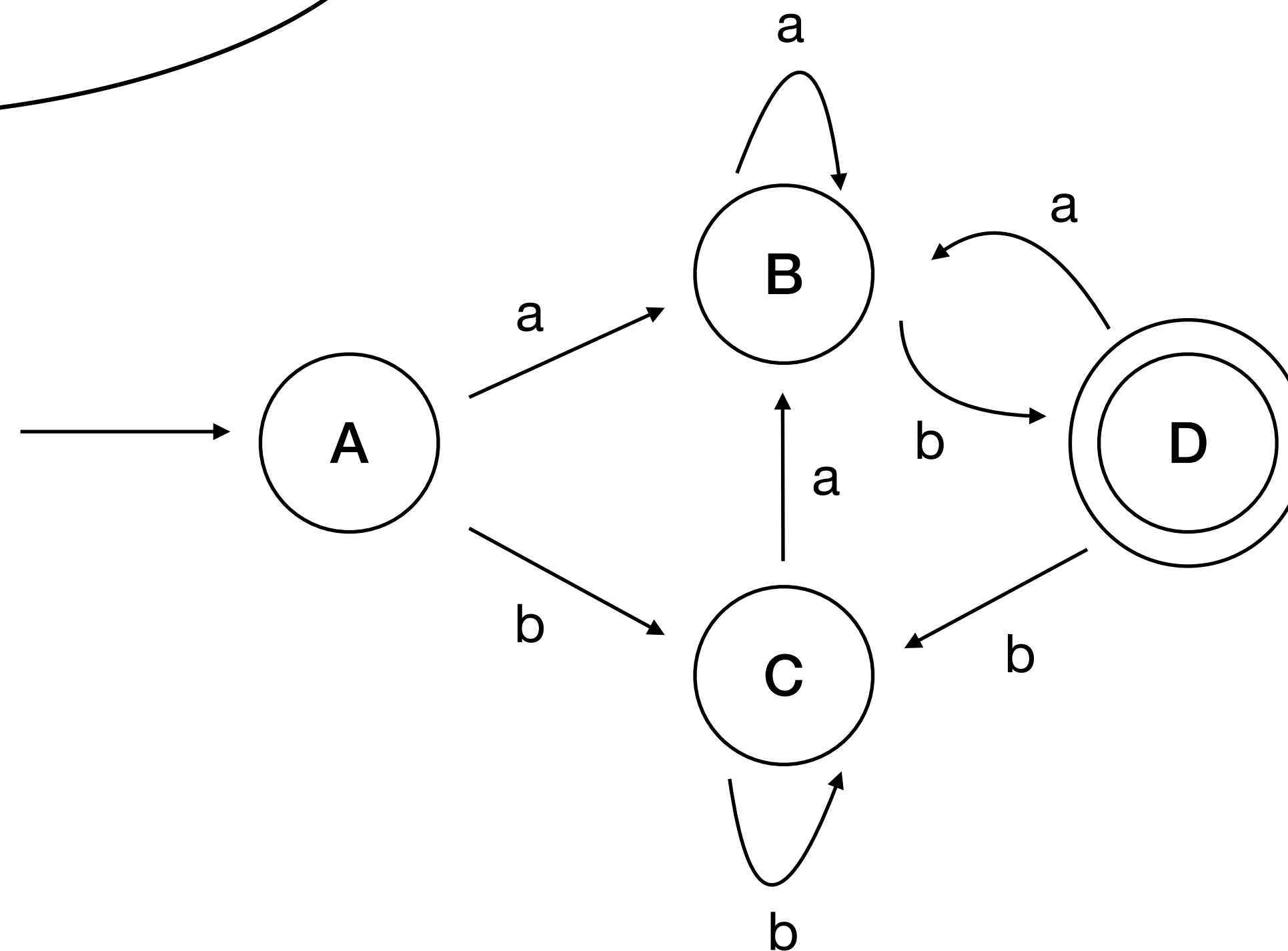


$A = \{ 0, 1, 2, 4, 7 \}$

$B = \{ 1, 2, 3, 4, 6, 7, 8 \}$

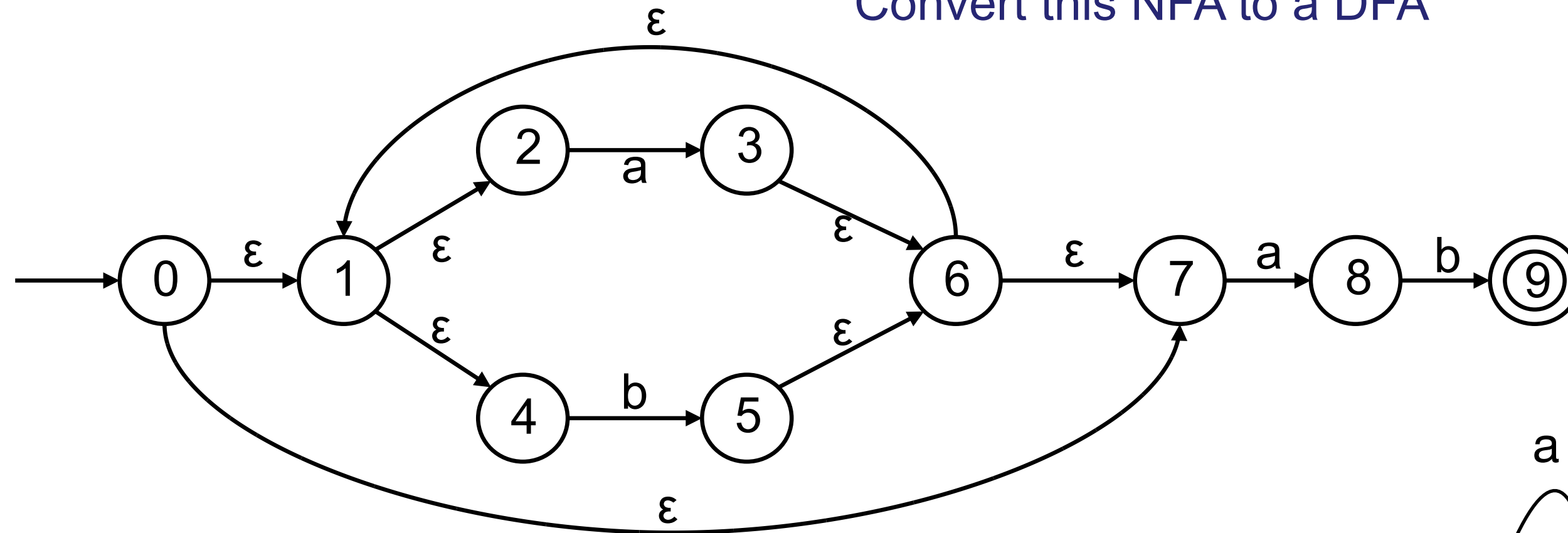
$C = \{ 1, 2, 4, 5, 6, 7 \}$

$D = \{ 1, 2, 4, 5, 6, 7, 9 \}$



# Class Problem

Convert this NFA to a DFA

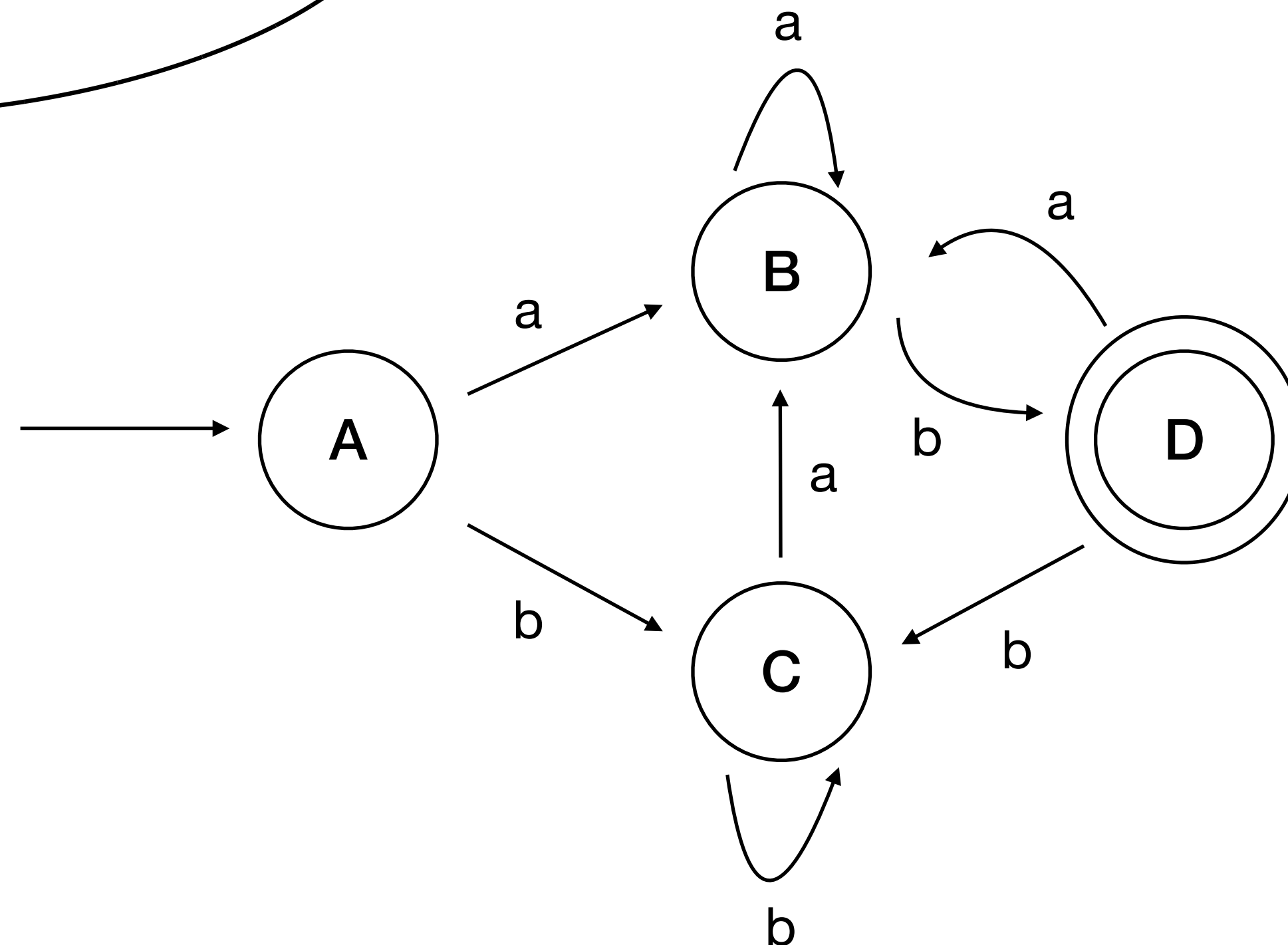


$$A = \{ 0, 1, 2, 4, 7 \} == (a|b)^*ab$$

$$B = \{ 1, 2, 3, 4, 6, 7, 8 \} == b|A$$

$$C = \{ 1, 2, 4, 5, 6, 7 \} == A$$

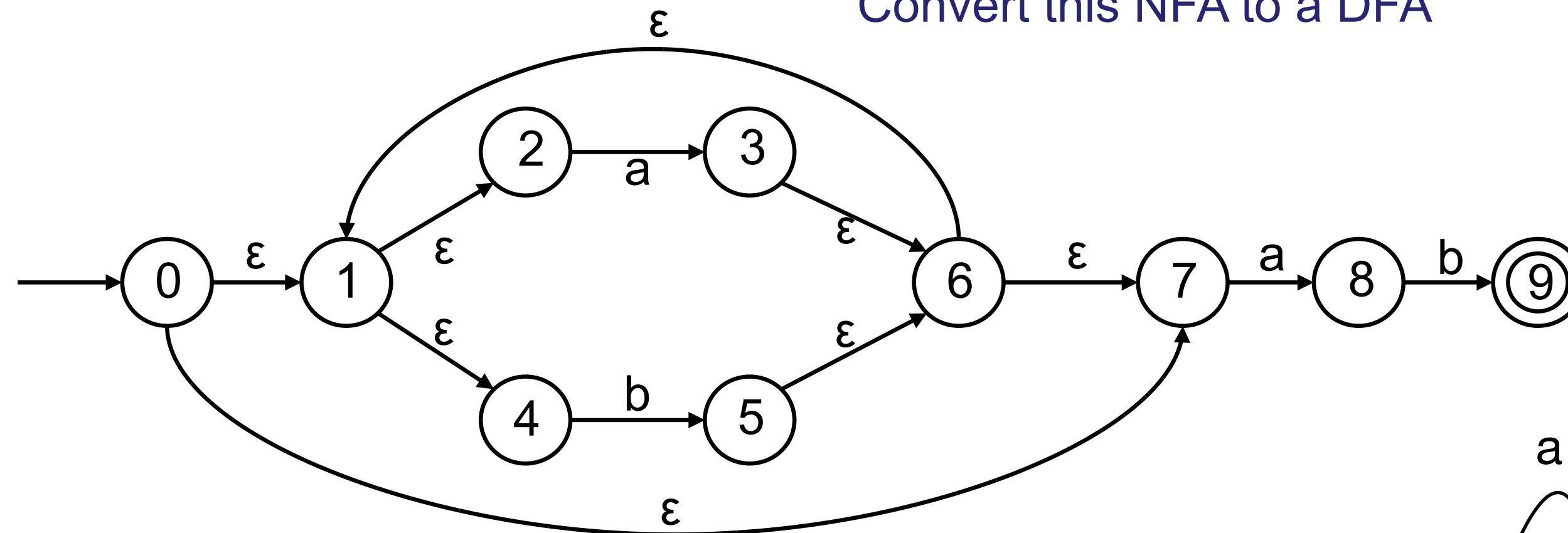
$$D = \{ 1, 2, 4, 5, 6, 7, 9 \} == \epsilon|A$$





# Class Problem

Convert this NFA to a DFA

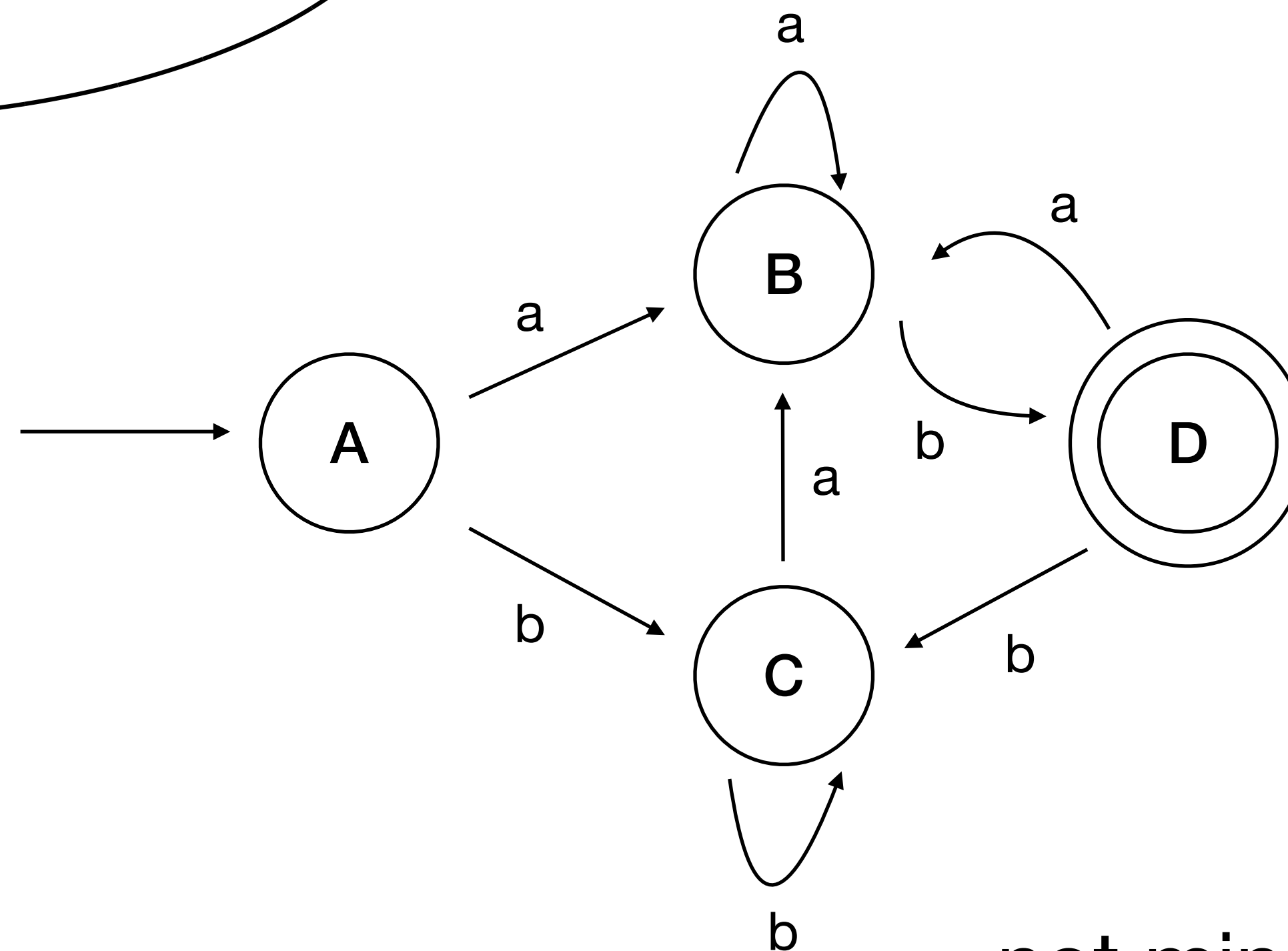


$$A = \{ 0, 1, 2, 4, 7 \} == (a|b)^*ab$$

$$B = \{ 1, 2, 3, 4, 6, 7, 8 \} == b|A$$

$$C = \{ 1, 2, 4, 5, 6, 7 \} == A$$

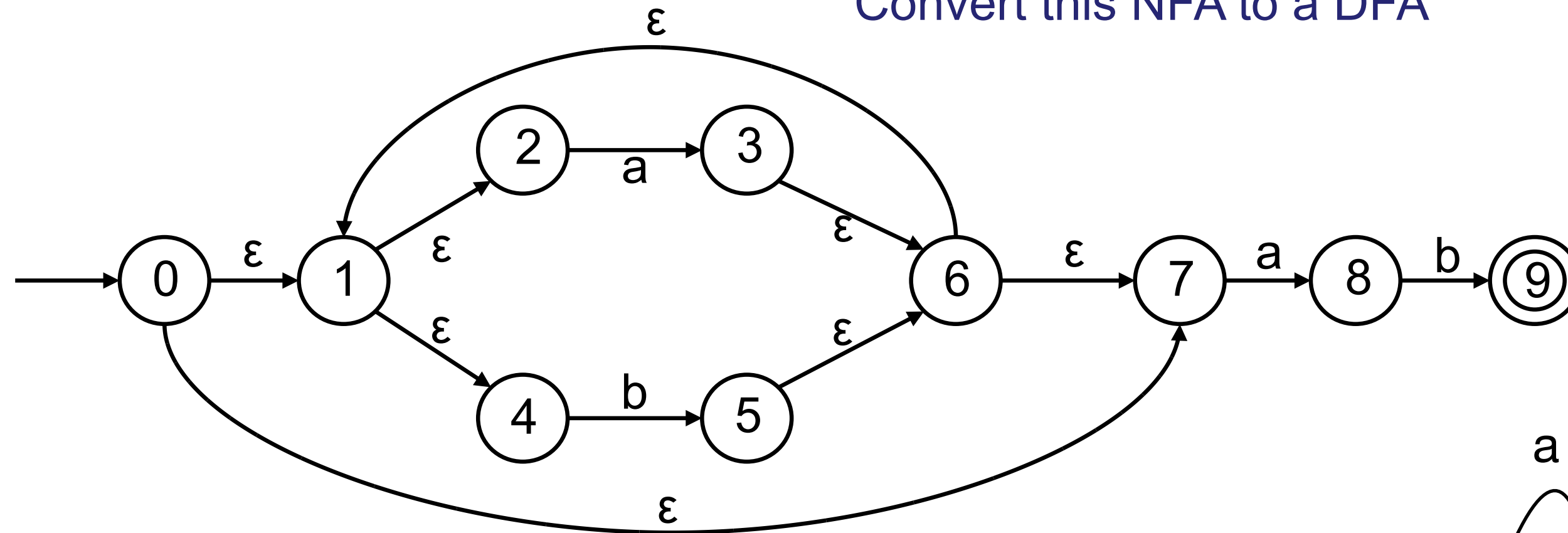
$$D = \{ 1, 2, 4, 5, 6, 7, 9 \} == \epsilon|A$$



not minimal,  $A == C$

# Class Problem

Convert this NFA to a DFA

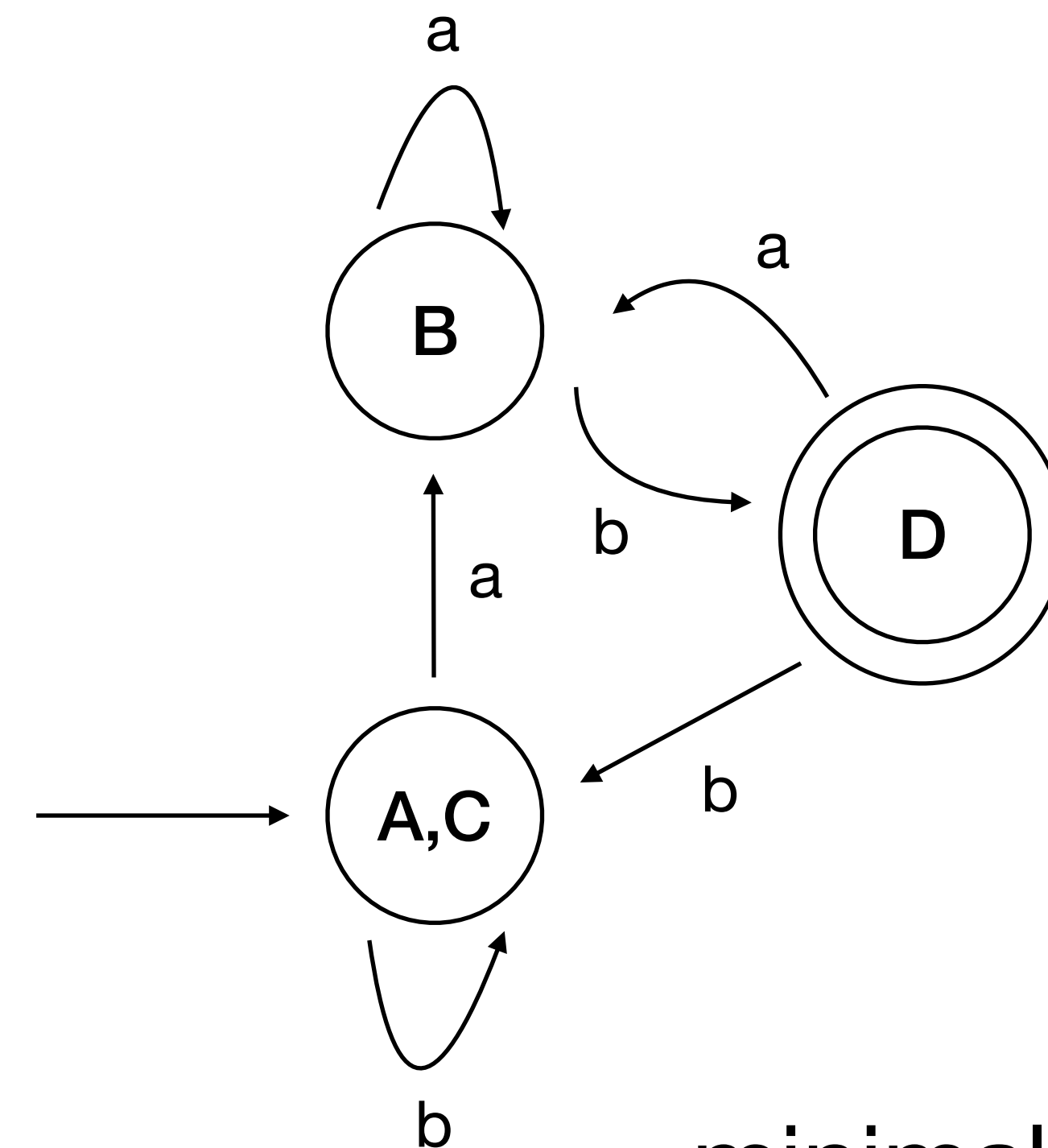


$$A = \{ 0, 1, 2, 4, 7 \} == (a|b)^*ab$$

$$B = \{ 1, 2, 3, 4, 6, 7, 8 \} == b|A$$

$$C = \{ 1, 2, 4, 5, 6, 7 \} == A$$

$$D = \{ 1, 2, 4, 5, 6, 7, 9 \} == \epsilon|A$$



minimal



## NFA to DFA : cont..

---

- An NFA may be in many states at any time
- How many different states ?
- If there are  $N$  states, the NFA must be in some subset of those  $N$  states
- How many subsets are there?

$$2^N - 1 = \text{finitely many}$$

# NFA Determinization: Correctness

The powerset construction takes an NFA **N** and constructs an equivalent DFA  $\text{Pow}(\text{N})$ .

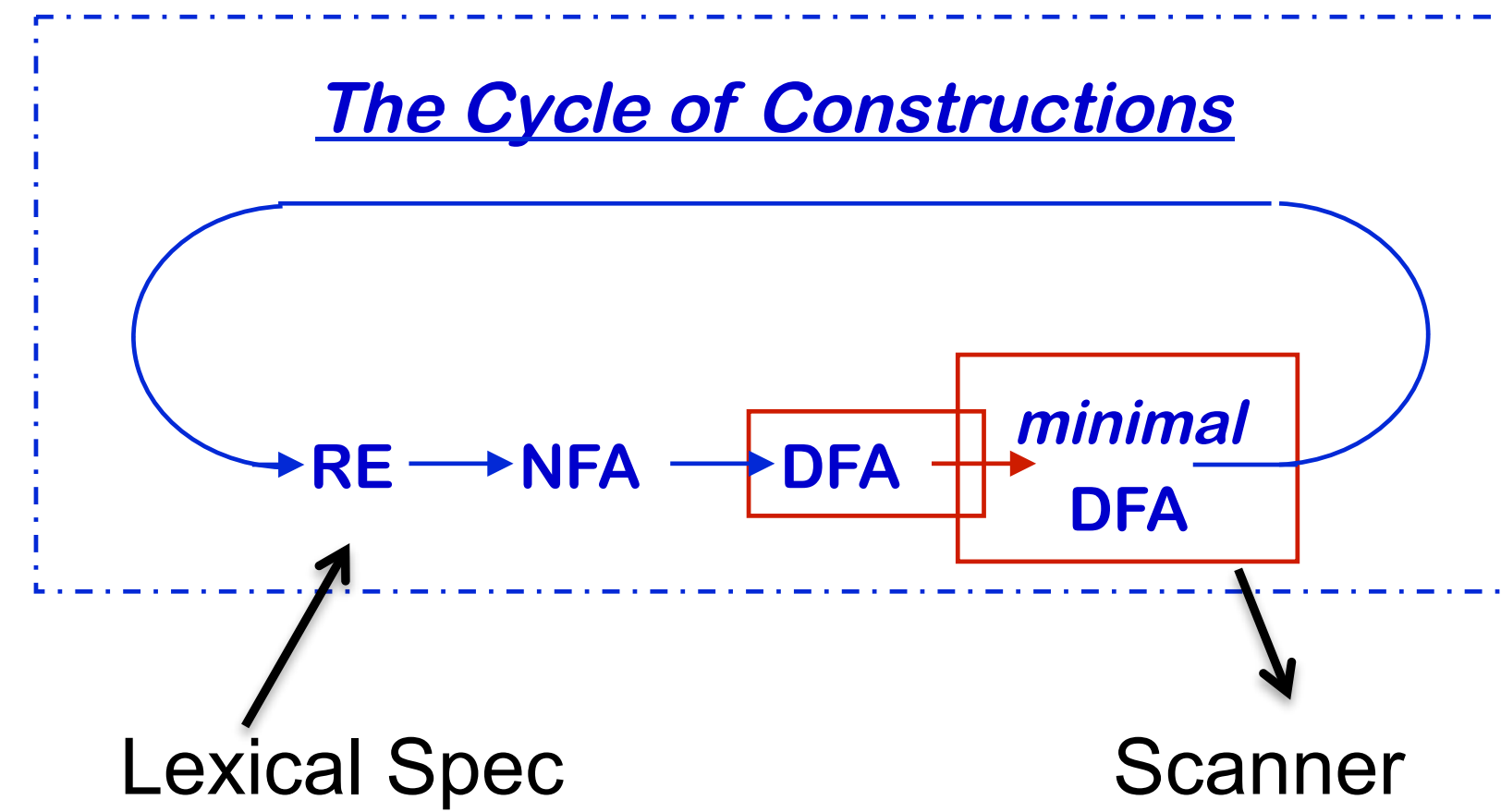
Equivalent means that a string has an accepting trace in **N** (starting at the start state) if and only if it does in  $\text{Pow}(\text{N})$  (starting at the start state).

Idea of proof is to generalize to a statement about traces starting at **any** state:

For any state  $s$  in  $N$  and accepting trace starting at  $s$ , show that **for all states**  $S$  in  $\text{Pow}(\text{N})$  if  $s$  in  $S$ , then there is an accepting trace starting at  $S$ .

For any state  $S$  in  $\text{Pow}(\text{N})$ , show that there **exists** a state  $s$  in  $S$  that has an accepting trace in  $N$  starting at  $s$ .

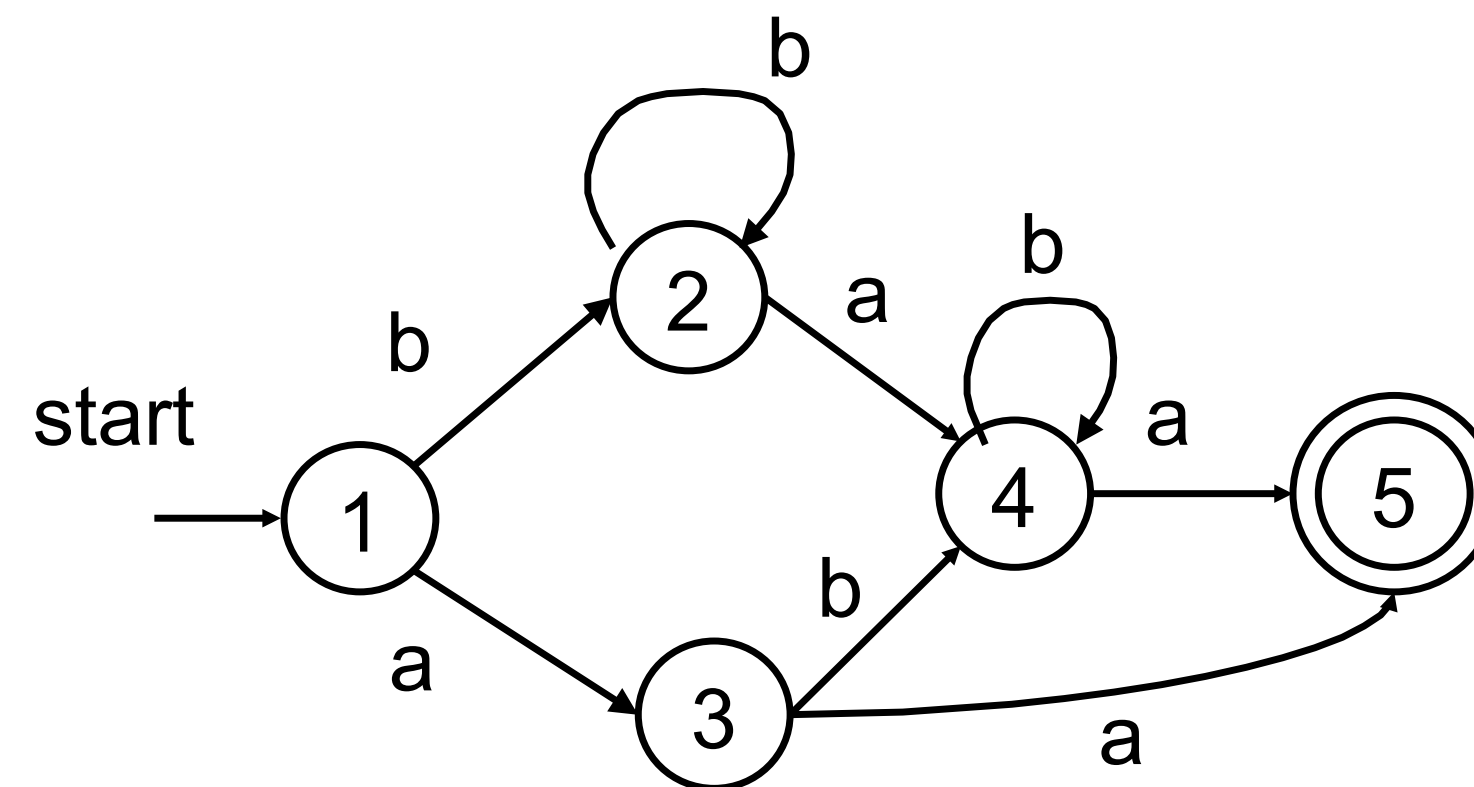
Proof each of these holds by induction on the length of the trace.



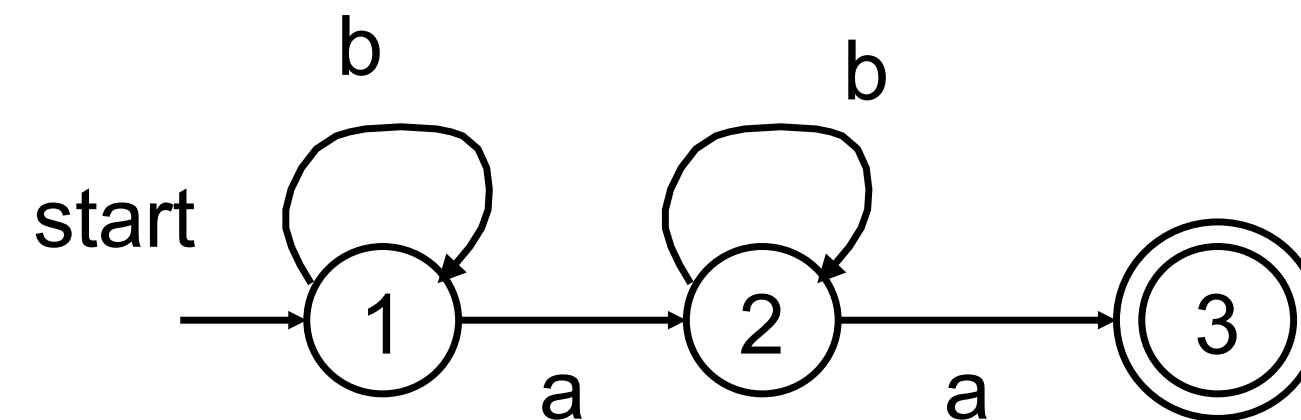


# State Minimization

- Resulting DFA can be quite large
  - Contains redundant or equivalent states



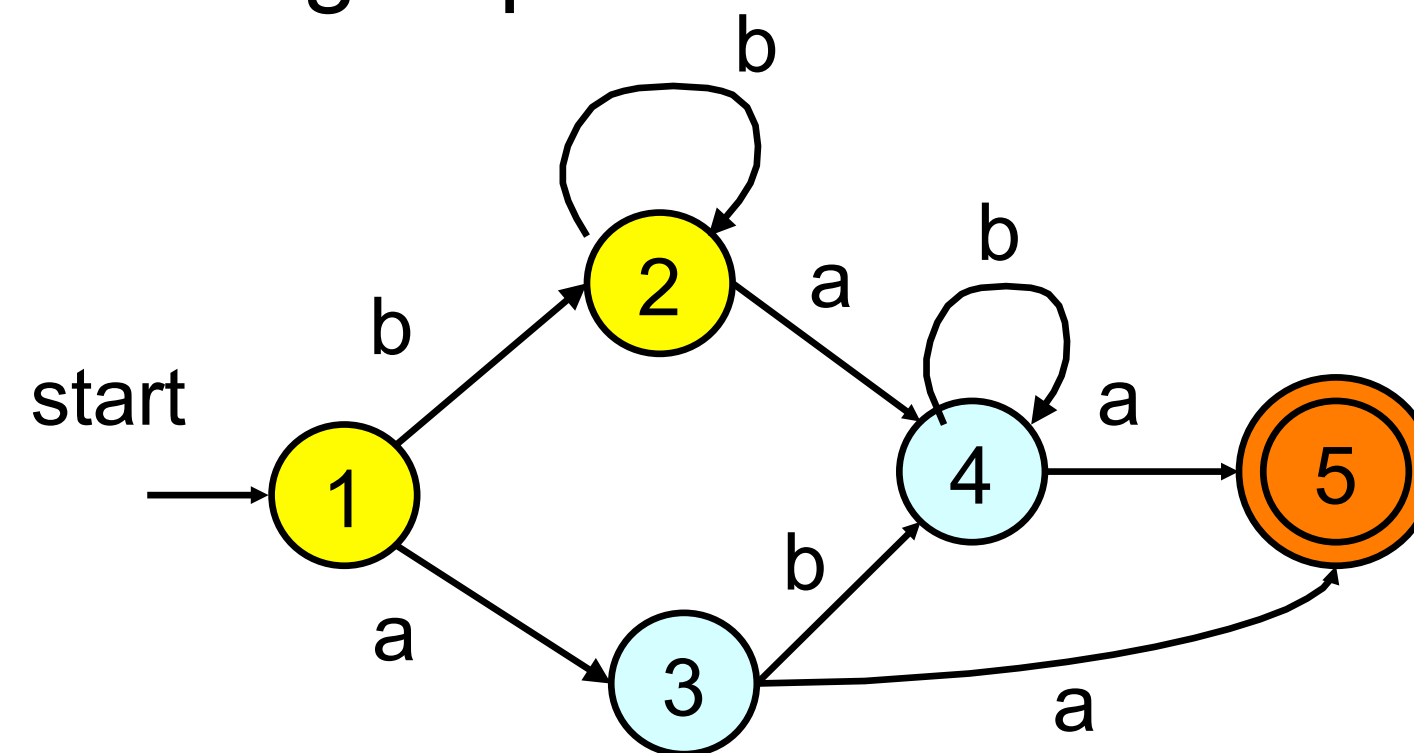
Both DFAs accept  
 $b^*ab^*a$



## State Minimization (2)

---

- Idea – find groups of equivalent states and merge them
  - All transitions from states in group G1 go to states in another group G2
  - Construct minimized DFA such that there is 1 state for each group of states



Basic strategy: identify distinguishing transitions

# DFA Minimization

Overview of algorithm:

Produce a **partition** of the states, so that states are in the same partition if they are equivalent.

Initialize: two sets, accepting and rejecting

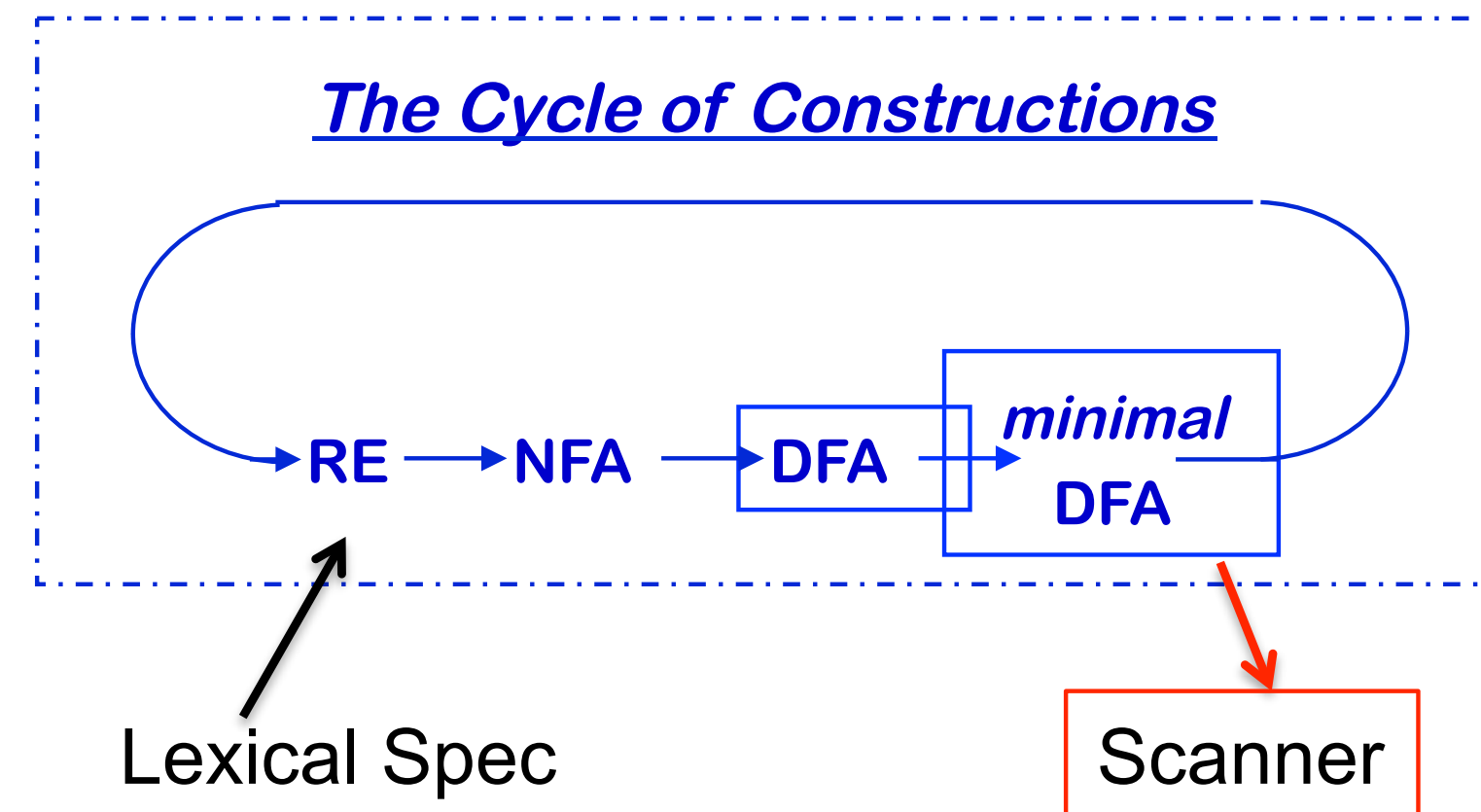
Update: If any states in the same partition make transitions to different partitions, split them.

Repeat until no new partitions are created

Minimized DFA has the partitions as states.

Similar to iterative fixpoint algorithms for dataflow analysis!





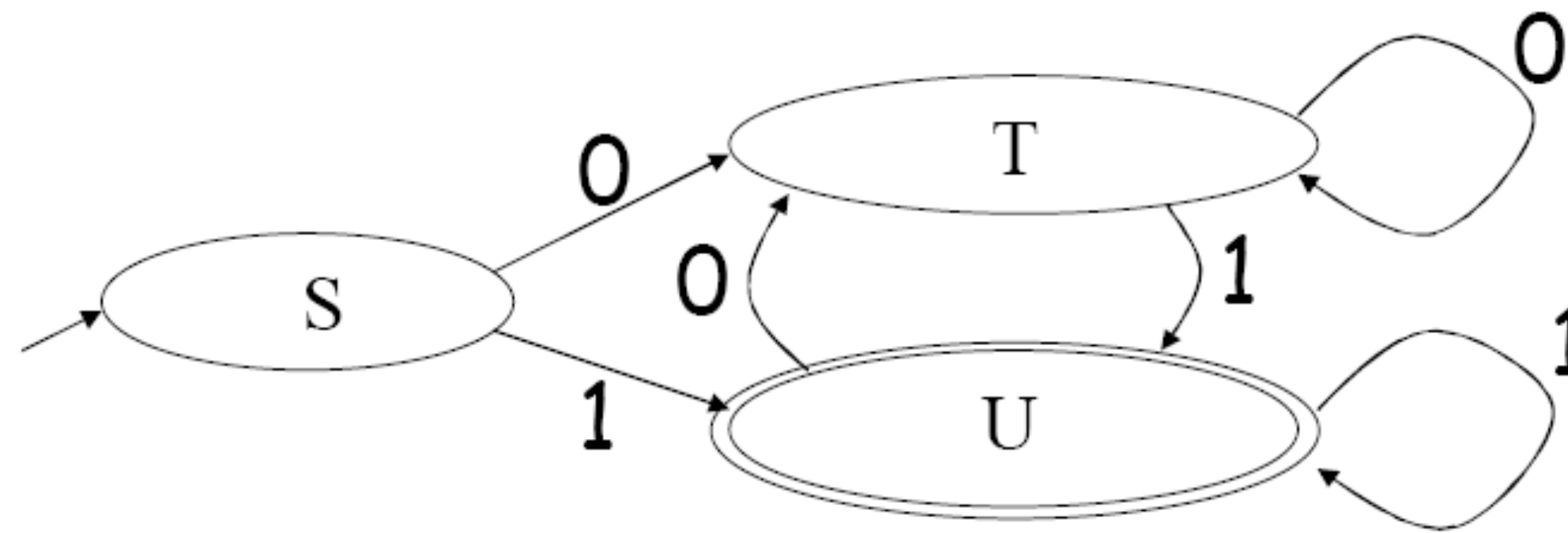
# DFA Implementation

---

- A DFA can be implemented by a 2D table T
  - One dimension is “states”
  - Other dimension is “input symbol”
  - For every transition  $S_i \xrightarrow{a} S_k$  define  $T[i,a] = k$
- DFA “execution”
  - If in state  $S_i$  and input  $a$ , read  $T[i,a] = k$  and skip to state  $S_k$
  - Very efficient



## DFA Table Implementation : Example



|   | 0 | 1 |
|---|---|---|
| S | T | U |
| T | T | U |
| U | T | U |

## Implementation Cont ..

---

- NFA -> DFA conversion is at the heart of tools such as flex
- But, DFAs can be huge
- In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations

# Lexer Generator

- Given regular expressions to describe the language (token types),
  - Step 1: Generates NFA that can recognize the regular language defined
  - Step 2: Transforms NFA to DFA
- Implemented in various lexer generators tools: lex/flex (C), ocamllex (OCaml), logos/lalrpop (Rust)



# Challenges for Lexical Analyzer

- How do we determine which lexemes are associated with each token?
  - Regular expression to describe token type
- When there are multiple ways we could scan the input, how do we know which one to pick?
- How do we address these concerns efficiently?

# Lexing Ambiguities

|              |                         |
|--------------|-------------------------|
| T_For        | for                     |
| T_Identifier | [A-Za-z_] [A-Za-z0-9_]* |

# Lexing Ambiguities

T\_For            for  
T\_Identifier    [A-Za-z\_] [A-Za-z0-9\_]\*

|   |   |   |   |
|---|---|---|---|
| f | o | r | t |
|---|---|---|---|



# Lexing Ambiguities

T\_For

for

T\_Identifier

[A-Za-z\_][A-Za-z0-9\_]\*

|   |   |   |   |
|---|---|---|---|
| f | o | r | t |
|---|---|---|---|

|   |   |   |   |   |
|---|---|---|---|---|
| f | o | r | t |   |
| f | o | r |   | t |
| f | o | r |   | t |
| f | o |   | r | t |
| f | o |   | r | t |

|   |   |   |   |   |
|---|---|---|---|---|
| f | o | r | t |   |
| f | o | r | t |   |
| f | o |   | r | t |
| f | o |   | r | t |

# Conflict Resolution

- Assume all tokens are specified as regular expressions.
- Algorithm: **Left-to-right scan**.
- Tiebreaking rule one: **Maximal munch**.
  - Always match the longest possible prefix of the remaining text.



# Implementing Maximal Munch

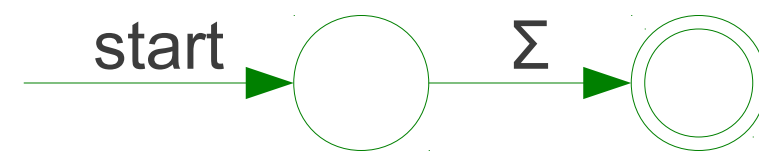
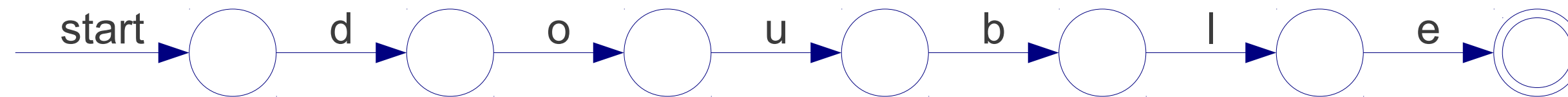
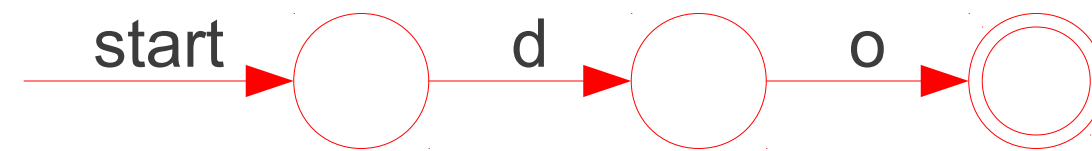
- Given a set of regular expressions, how can we use them to implement maximum munch?

# Implementing Maximal Munch

|           |          |
|-----------|----------|
| T_Do      | do       |
| T_Double  | double   |
| T_Mystery | [A-Za-z] |

# Implementing Maximal Munch

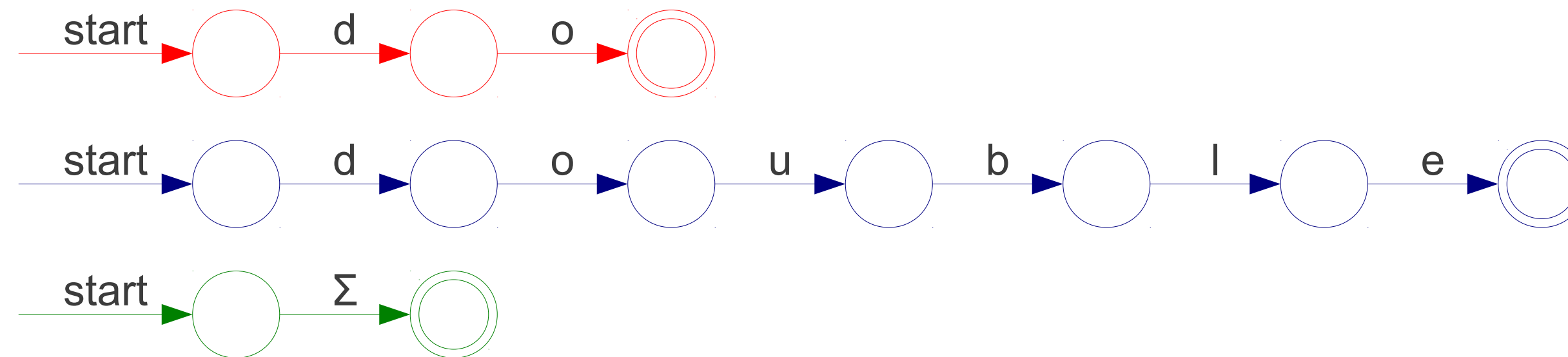
|           |          |
|-----------|----------|
| T_Do      | do       |
| T_Double  | double   |
| T_Mystery | [A-Za-z] |





# Implementing Maximal Munch

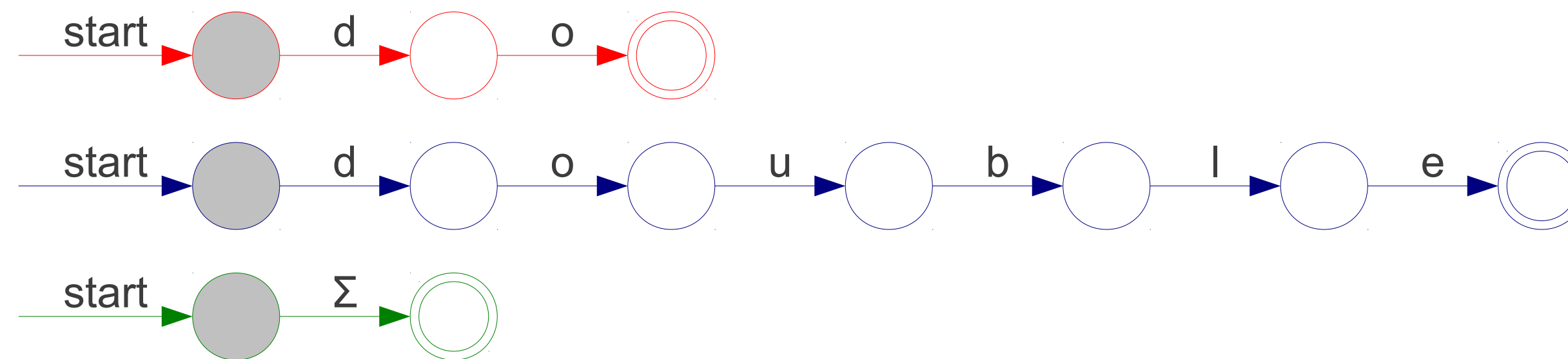
T\_Do            do  
T\_Double       double  
T\_Mystery       [A-Za-z]



|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| D | O | U | B | D | O | U | B | L | E |
|---|---|---|---|---|---|---|---|---|---|

# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery       [A-Za-z]



|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| D | O | U | B | D | O | U | B | L | E |
|---|---|---|---|---|---|---|---|---|---|

# Implementing Maximal Munch

T\_Do

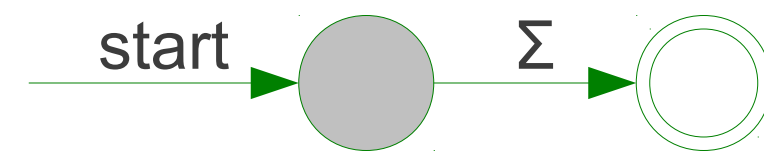
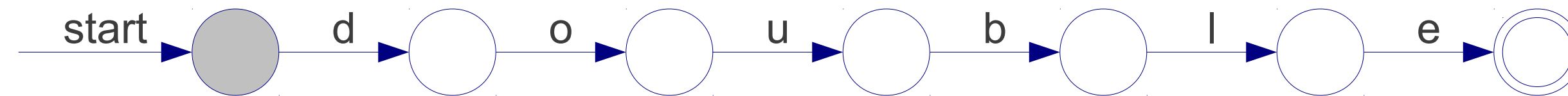
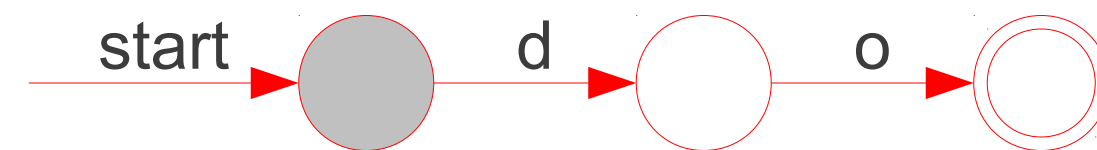
do

T\_Double

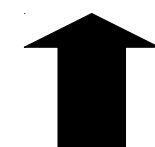
double

T\_Mystery

[A-Za-z]



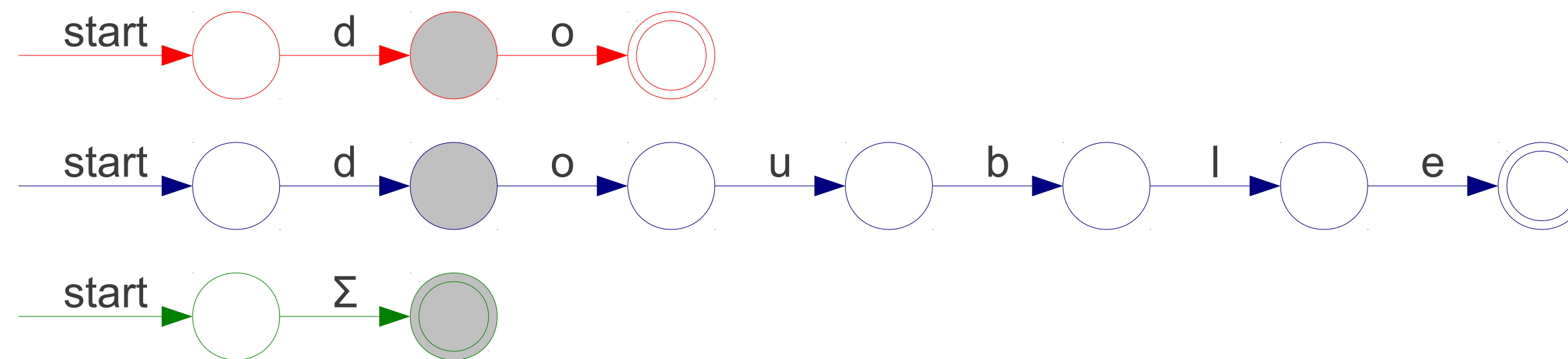
|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| D | O | U | B | D | O | U | B | L | E |
|---|---|---|---|---|---|---|---|---|---|



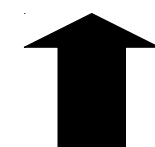


# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery       [A-Za-z]



|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| D | O | U | B | D | O | U | B | L | E |
|---|---|---|---|---|---|---|---|---|---|



# Implementing Maximal Munch

T\_Do

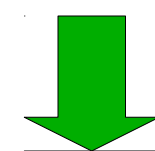
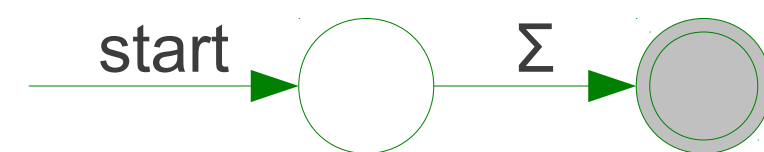
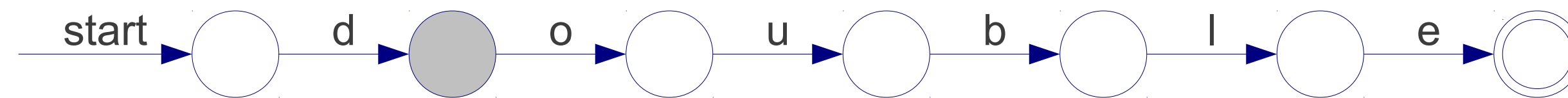
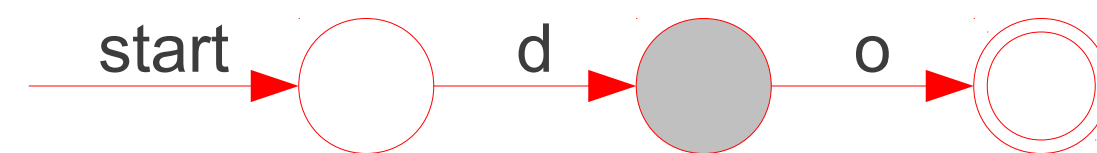
do

T\_Double

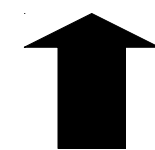
double

T\_Mystery

[A-Za-z]



|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| D | O | U | B | D | O | U | B | L | E |
|---|---|---|---|---|---|---|---|---|---|





# Implementing Maximal Munch

T\_Do

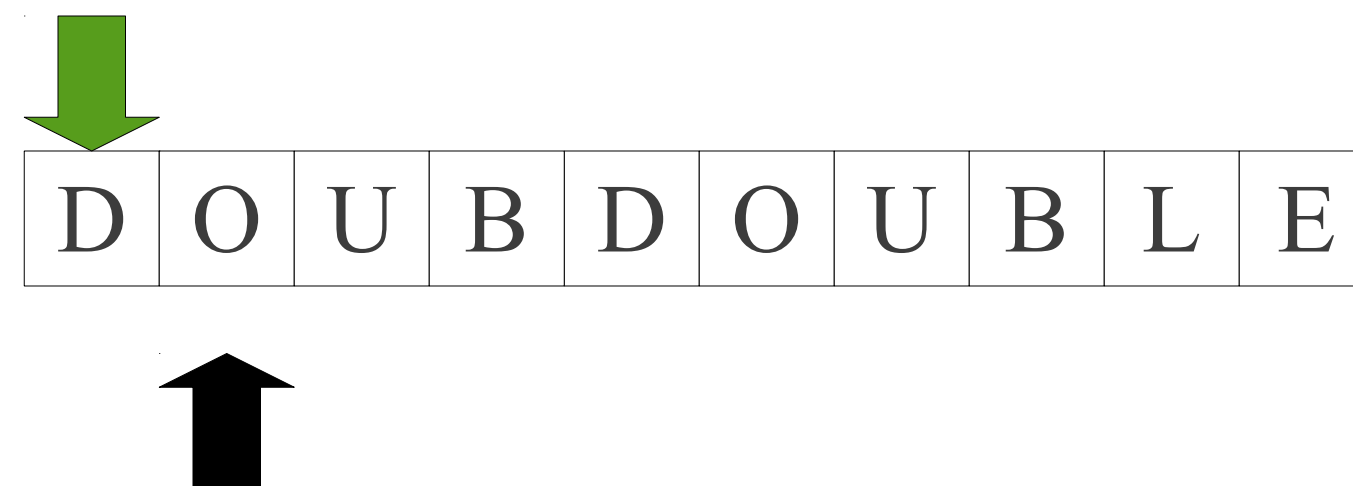
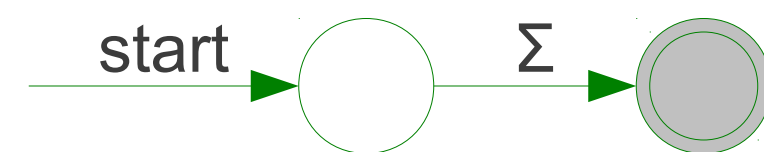
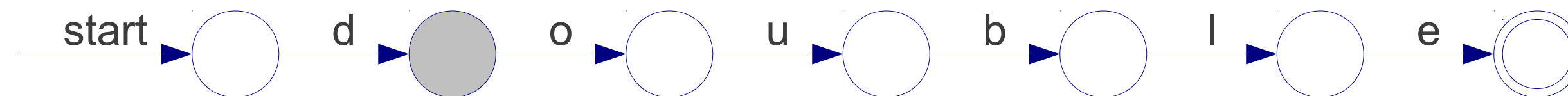
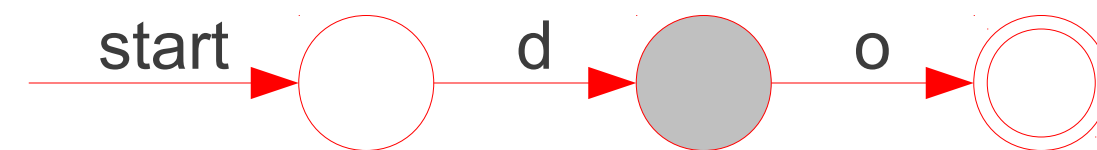
do

T\_Double

double

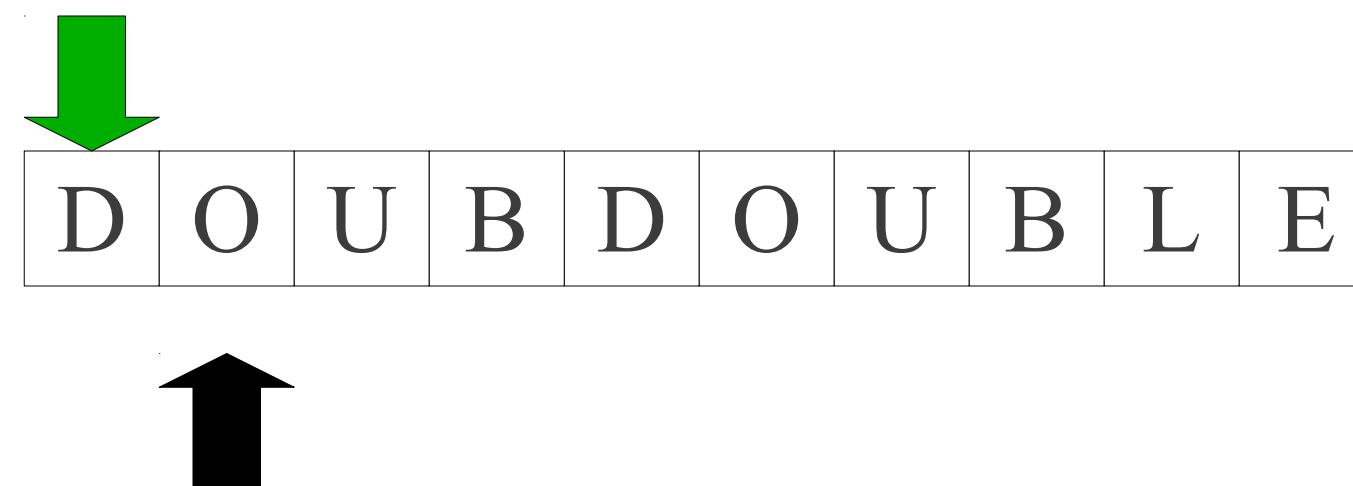
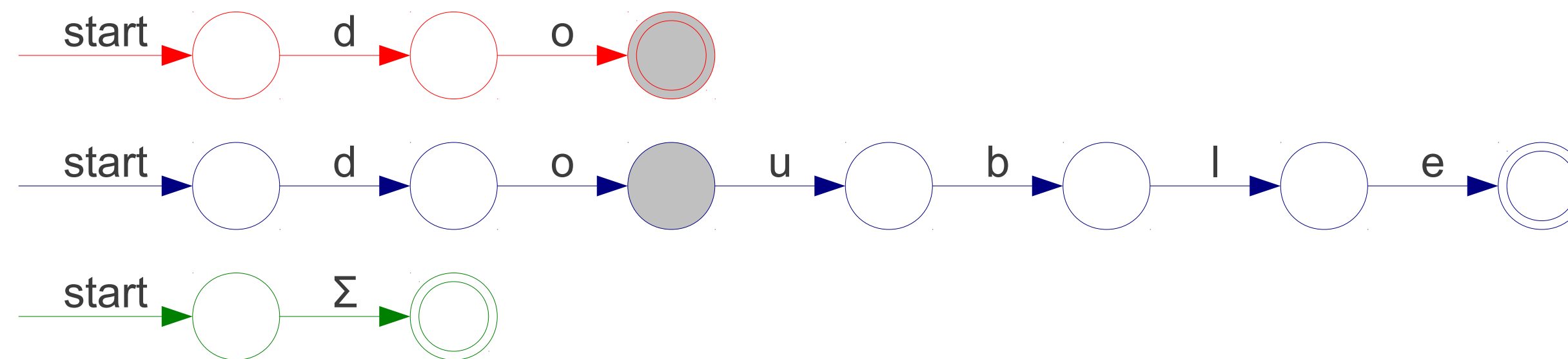
T\_Mystery

[A-Za-z]



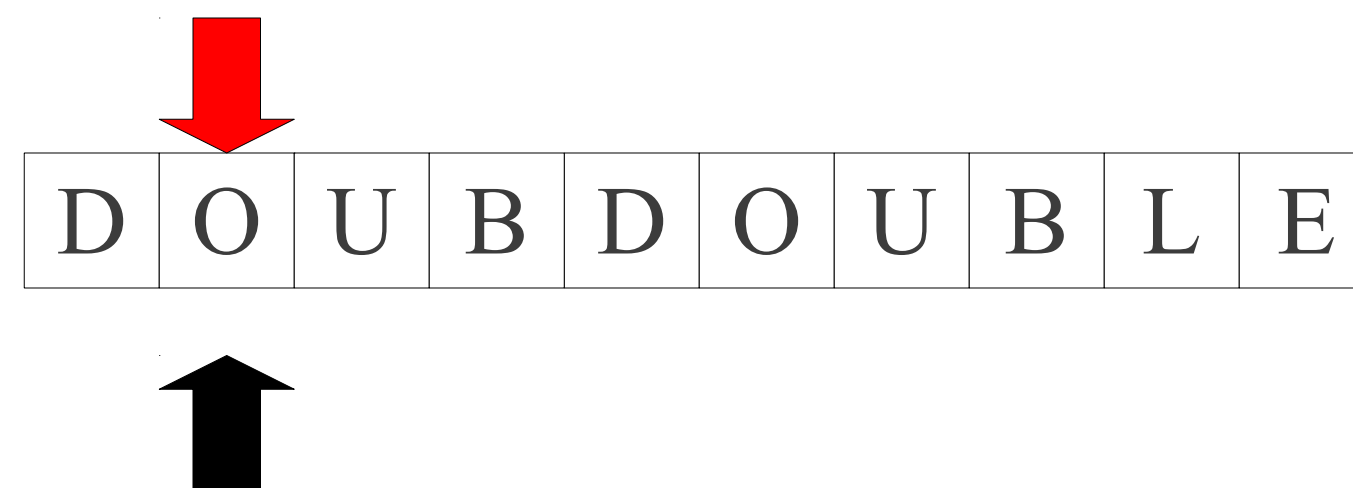
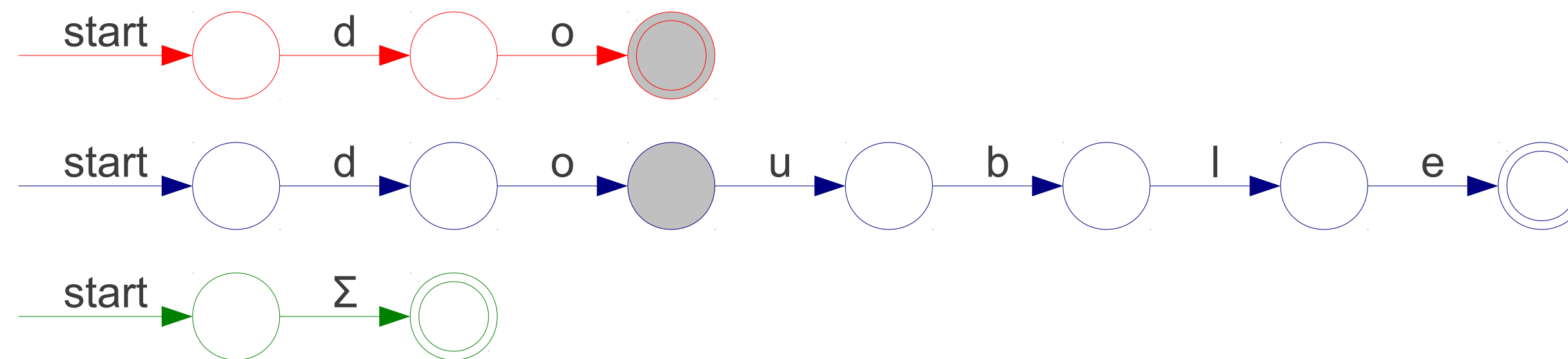
# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery       [A-Za-z]



# Implementing Maximal Munch

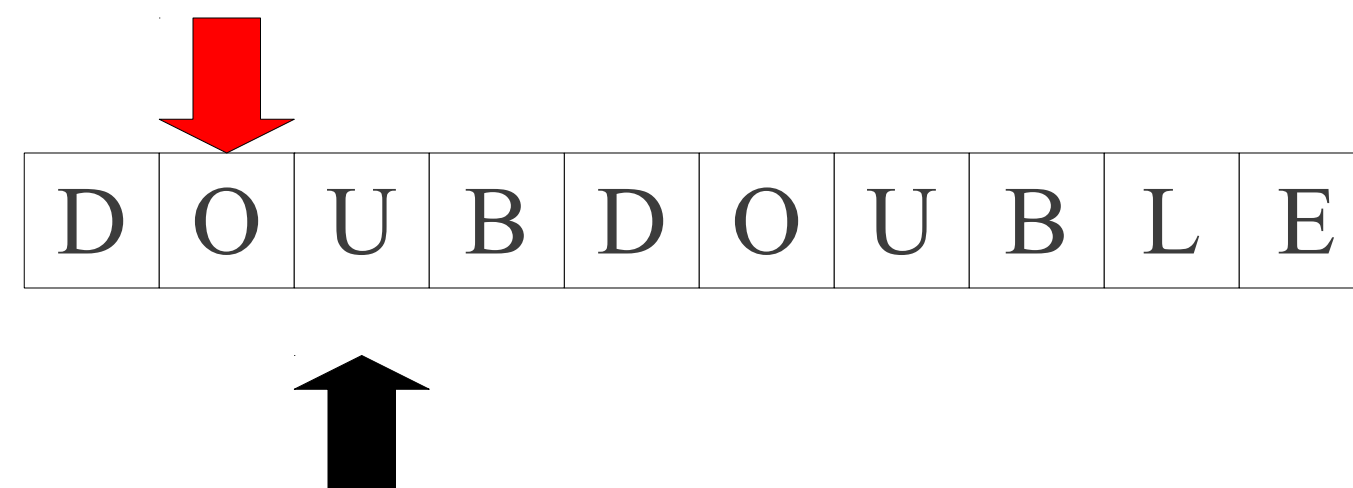
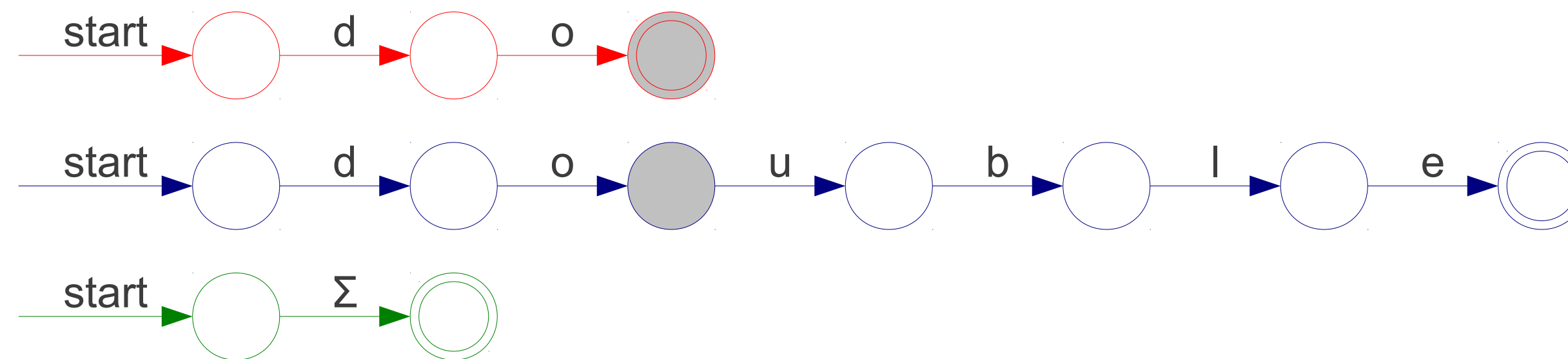
T\_Do            do  
T\_Double       double  
T\_Mystery      [A-Za-z]





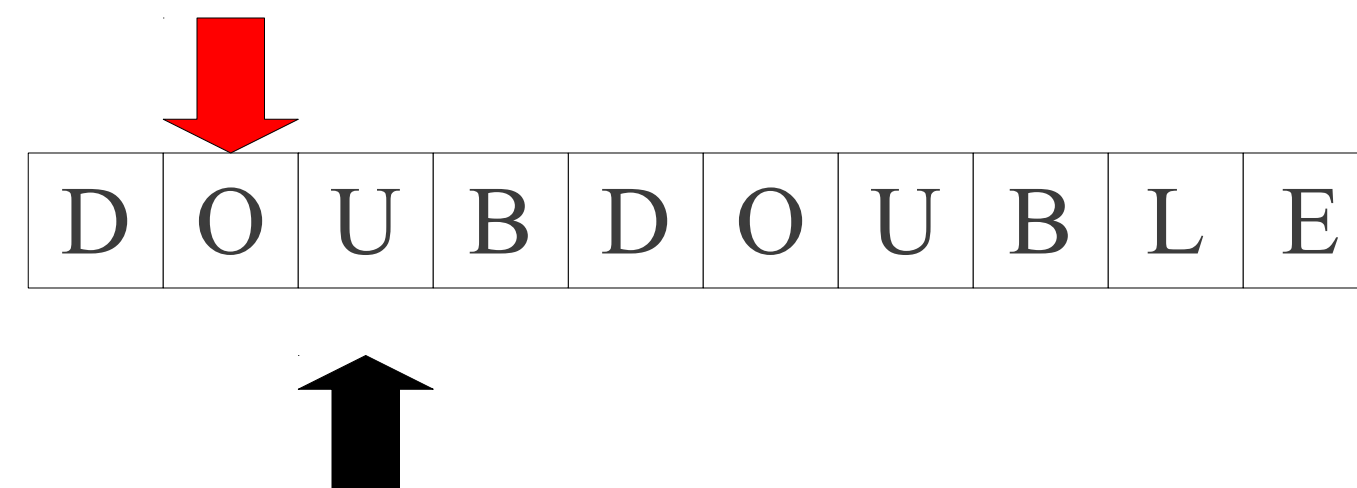
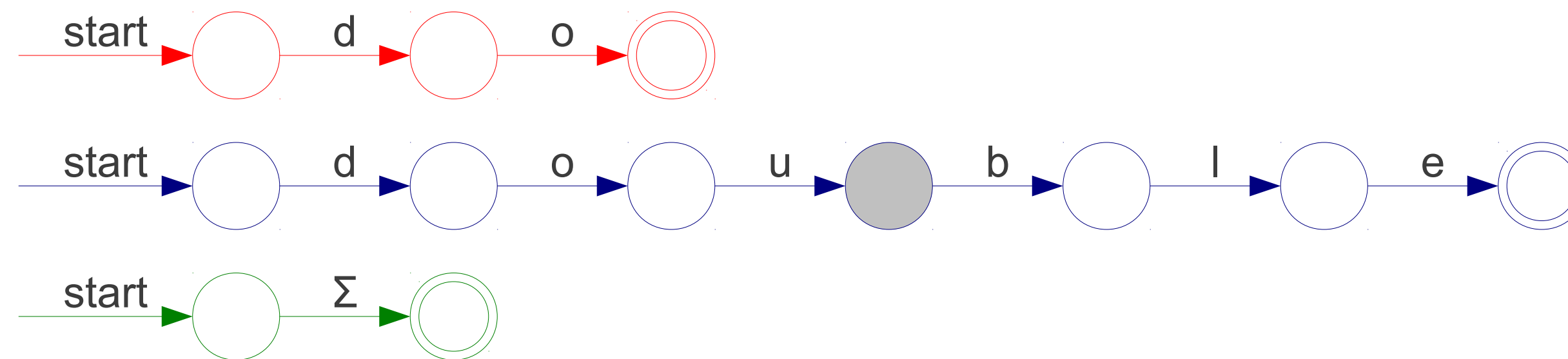
# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery       [A-Za-z]



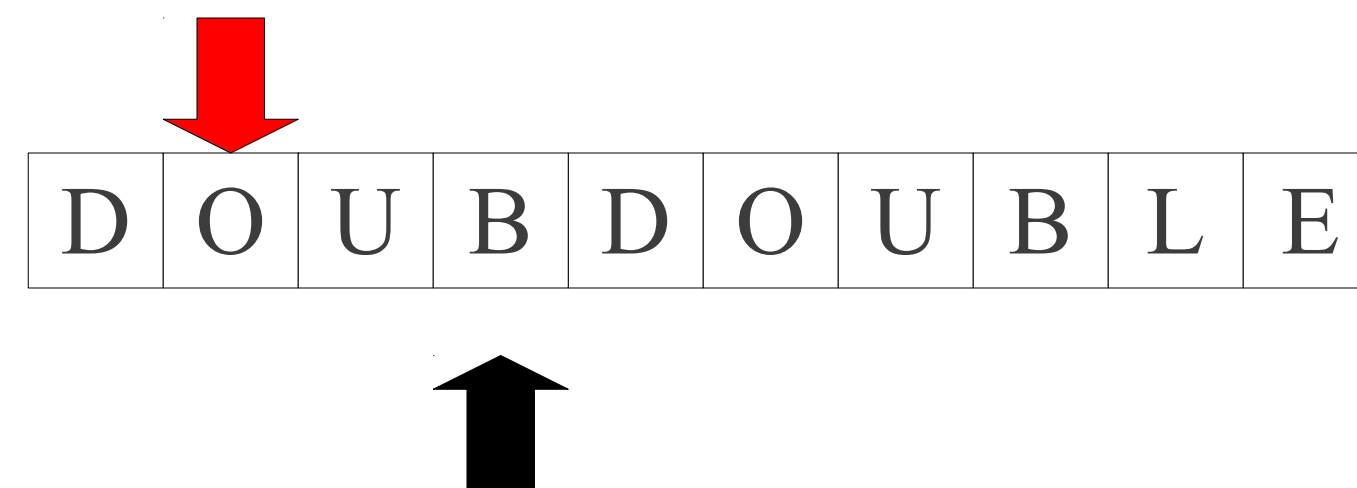
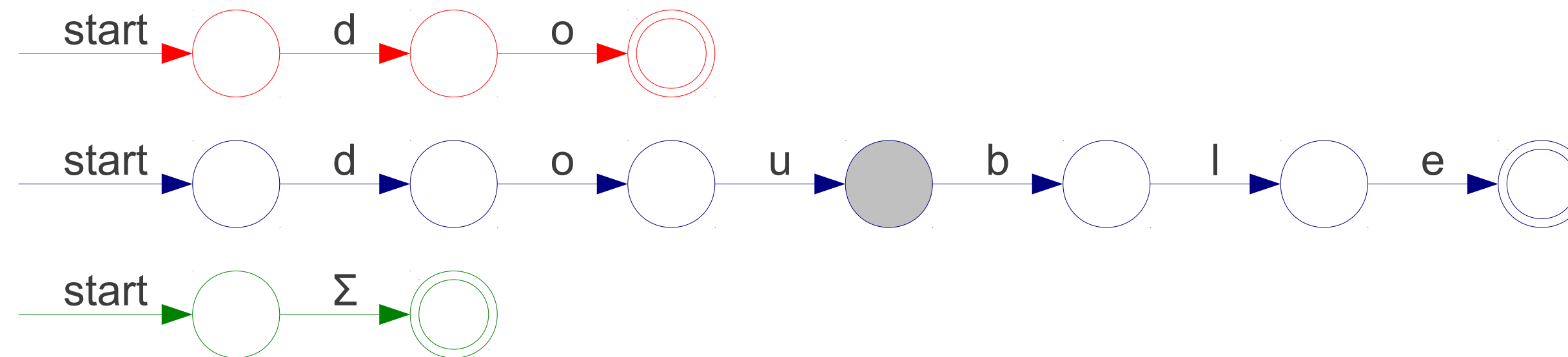
# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery      [A-Za-z]



# Implementing Maximal Munch

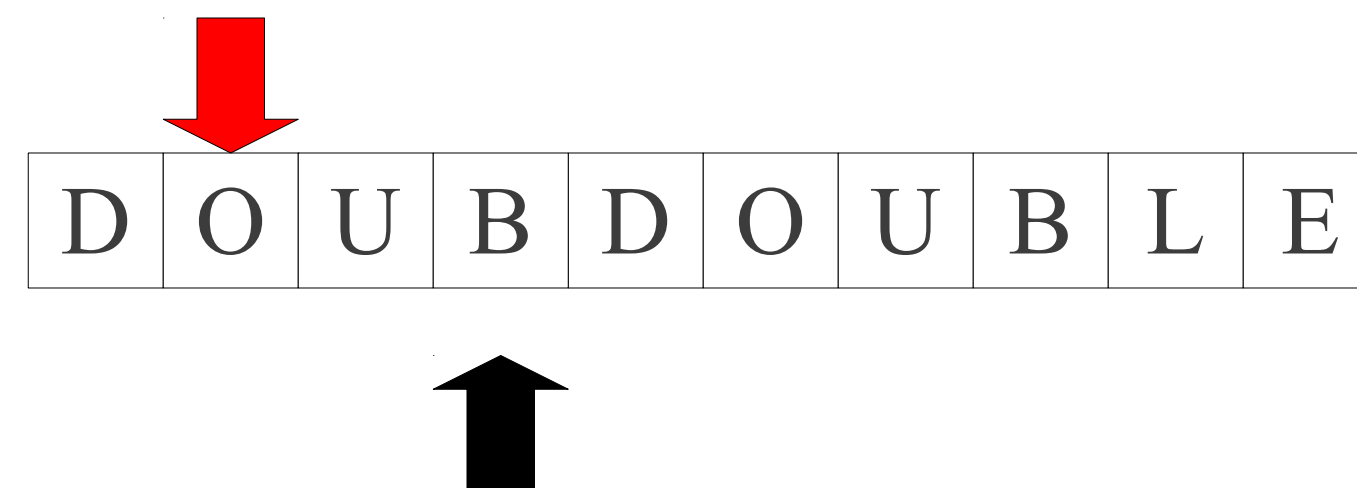
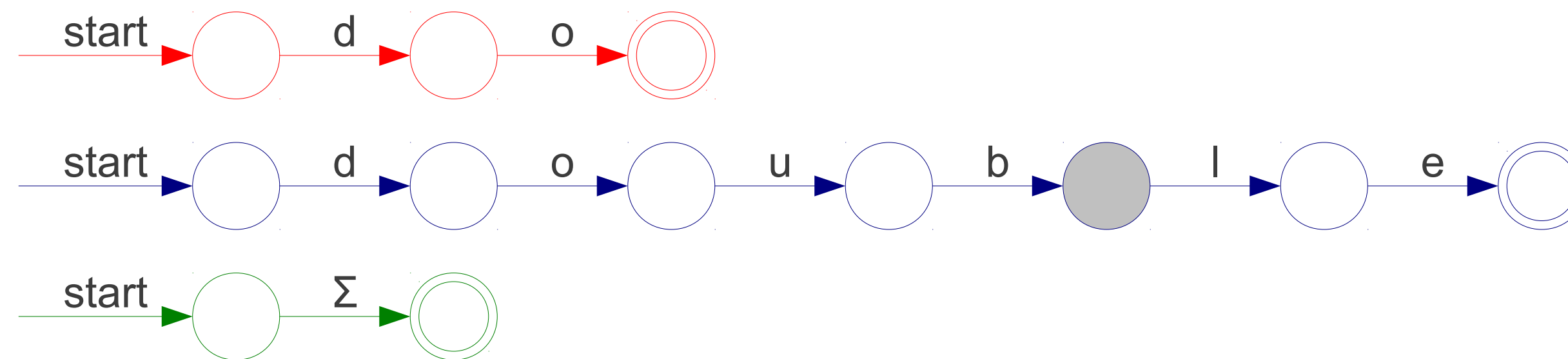
T\_Do            do  
T\_Double       double  
T\_Mystery       [A-Za-z]





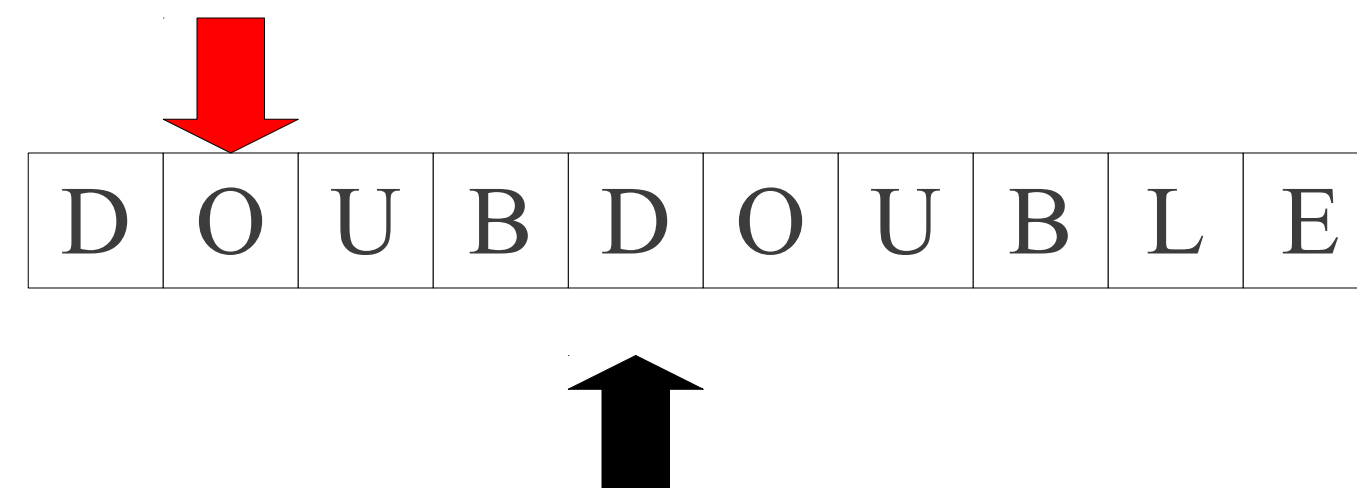
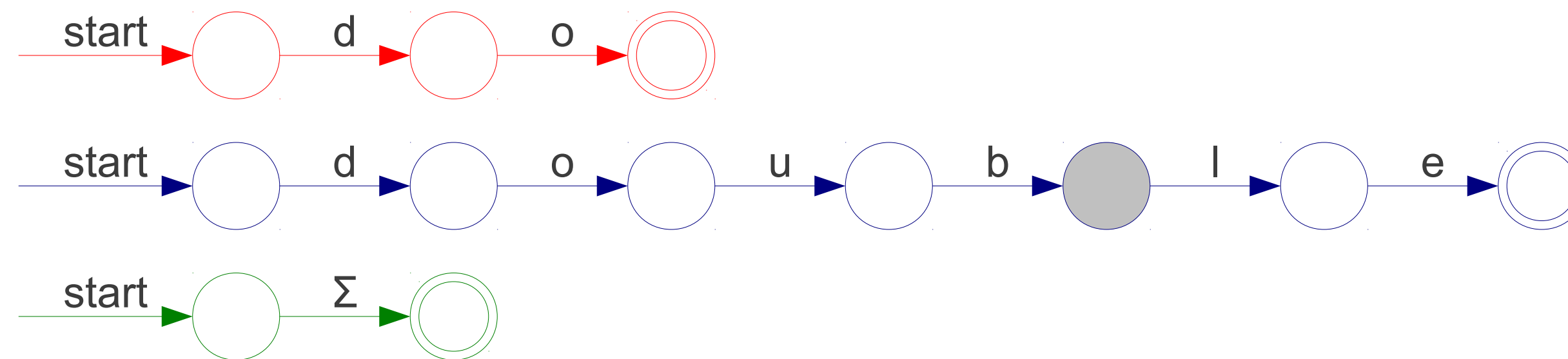
# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery       [A-Za-z]



# Implementing Maximal Munch

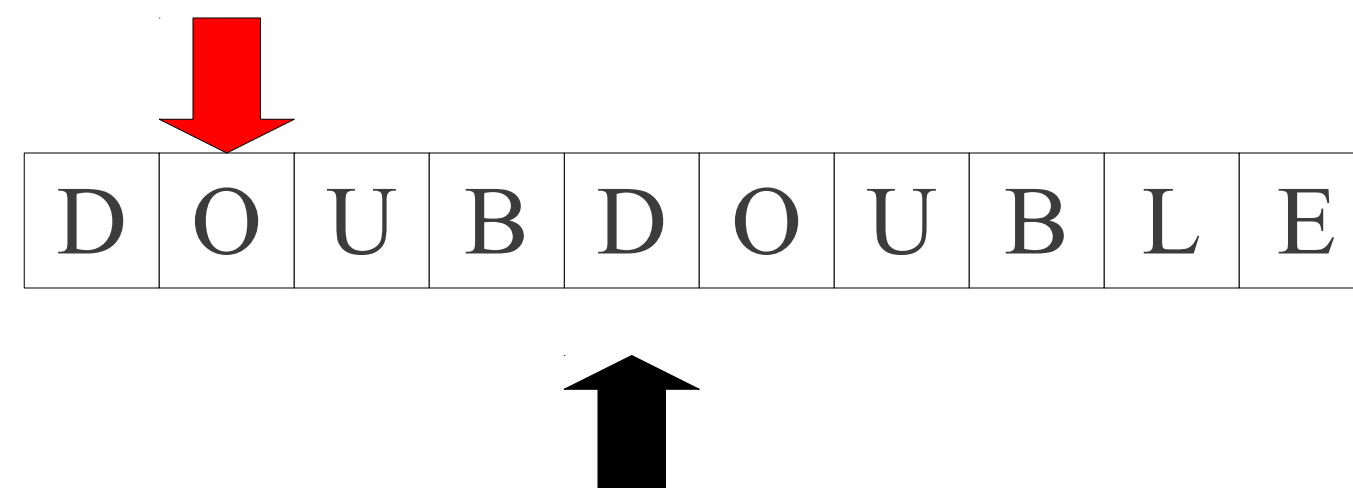
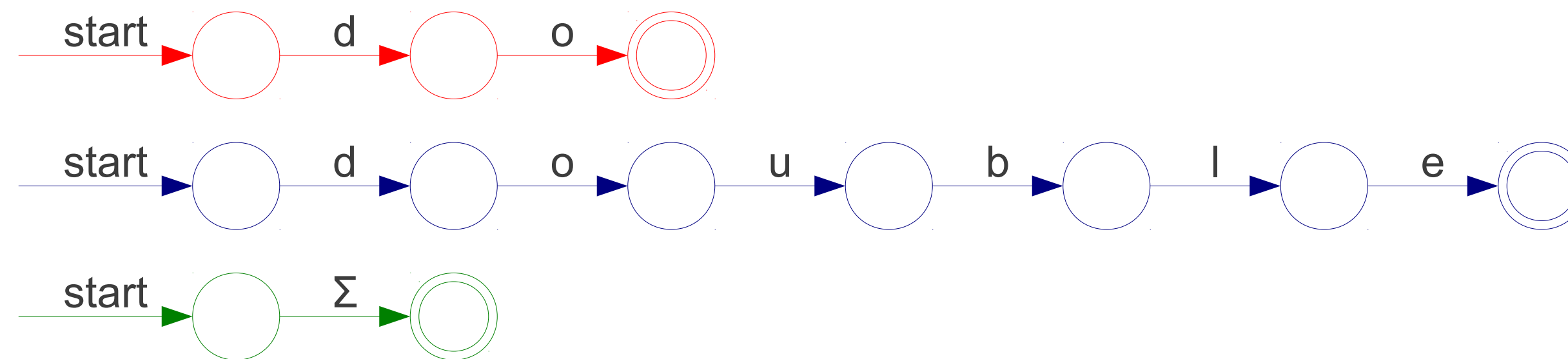
T\_Do            do  
T\_Double       double  
T\_Mystery      [A-Za-z]





# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery       [A-Za-z]



# Implementing Maximal Munch

T\_Do

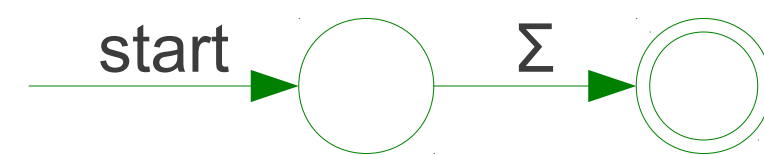
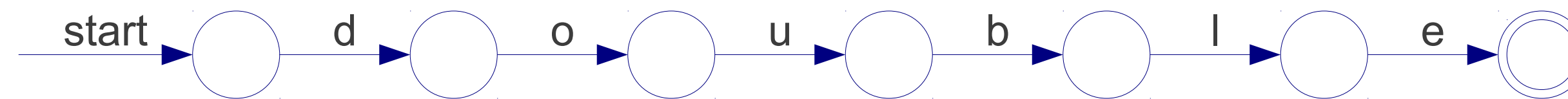
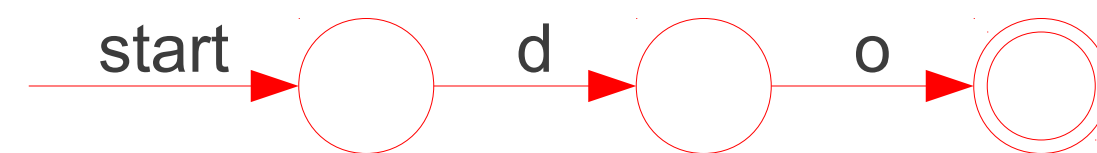
do

T\_Double

double

T\_Mystery

[A-Za-z]



**D O**

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| U | B | D | O | U | B | L | E |
|---|---|---|---|---|---|---|---|



# Implementing Maximal Munch

T\_Do

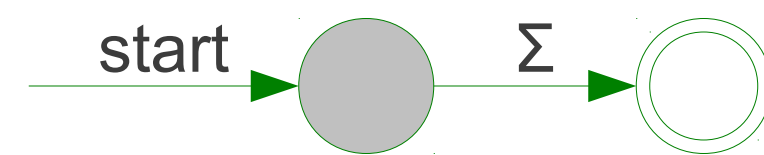
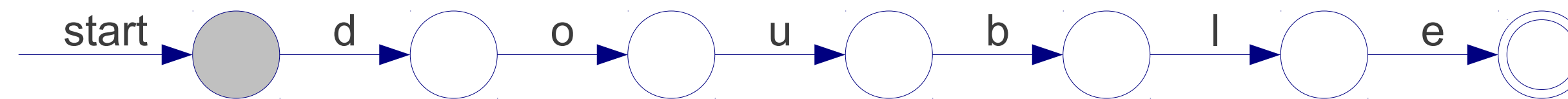
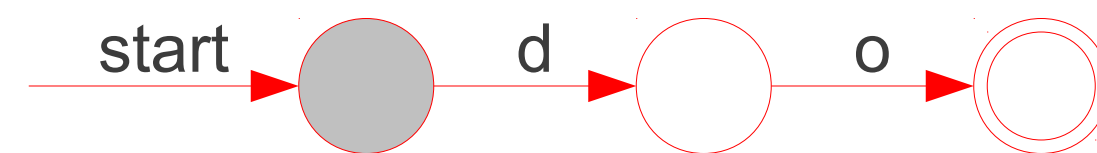
do

T\_Double

double

T\_Mystery

[A-Za-z]



**D O**

U B D O U B L E





# Implementing Maximal Munch

T\_Do

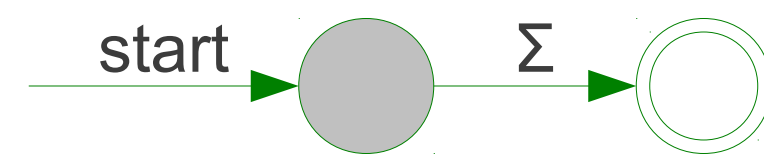
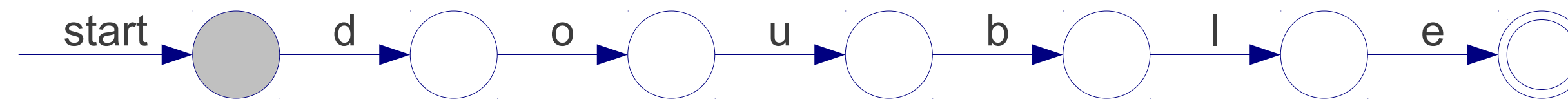
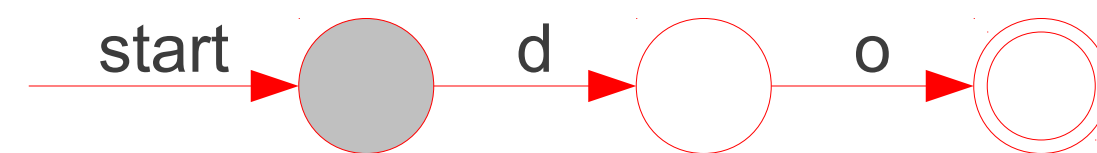
do

T\_Double

double

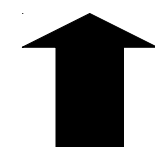
T\_Mystery

[A-Za-z]



**D O**

U B D O U B L E



# Implementing Maximal Munch

T\_Do

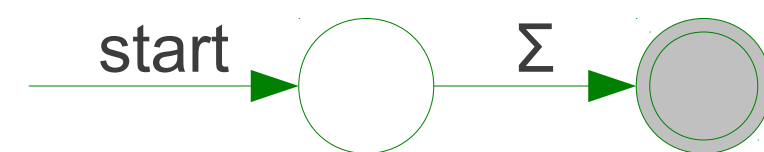
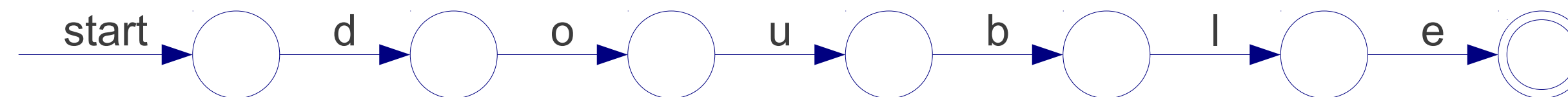
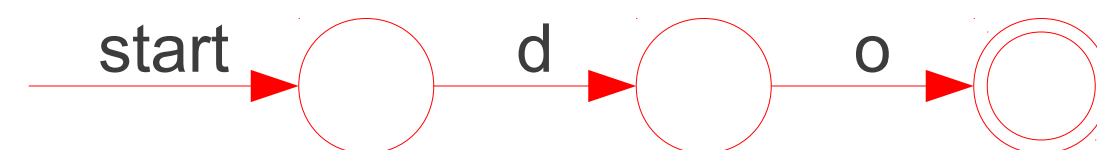
do

T\_Double

double

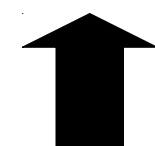
T\_Mystery

[A-Za-z]



**D O**

U B D O U B L E



# Implementing Maximal Munch

T\_Do

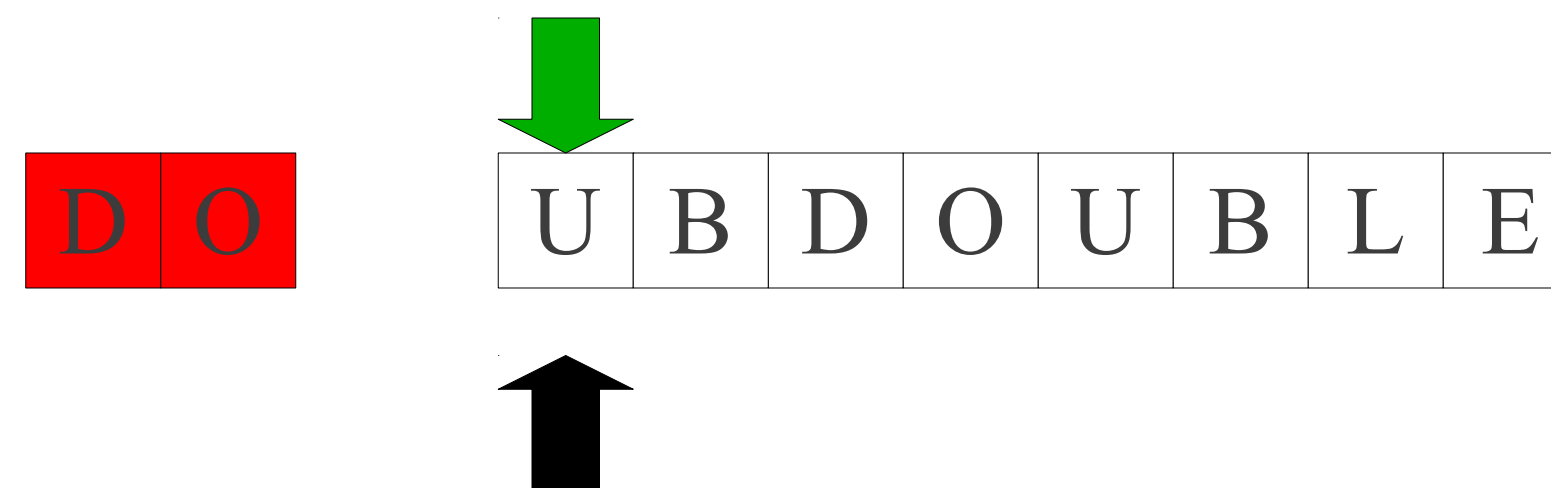
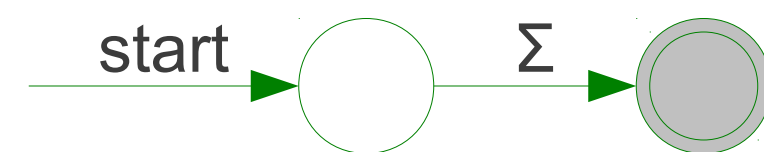
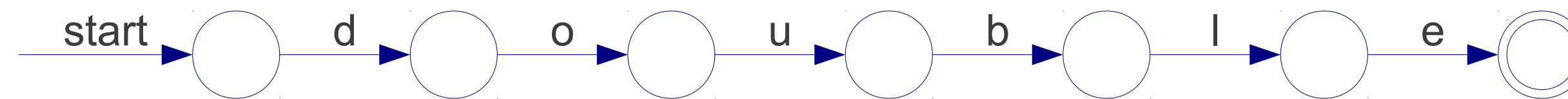
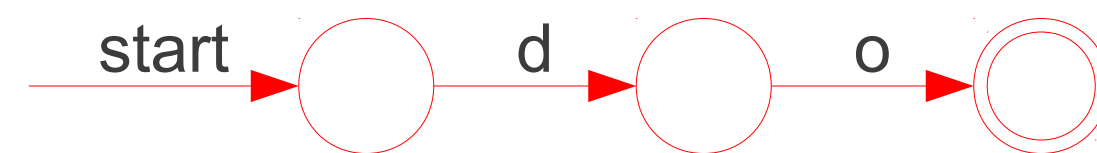
do

T\_Double

double

T\_Mystery

[A-Za-z]





# Implementing Maximal Munch

T\_Do

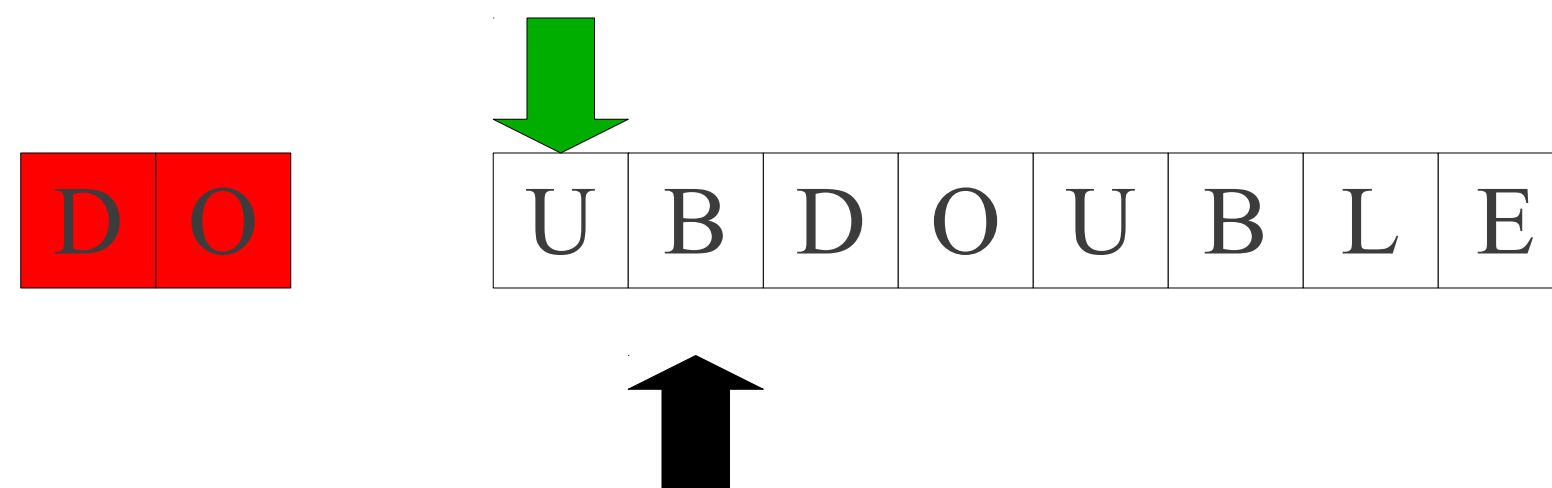
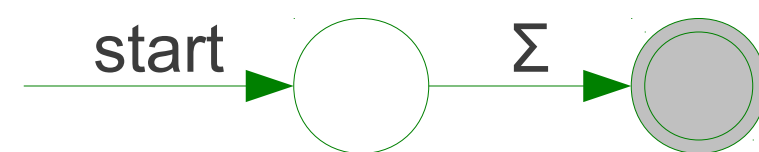
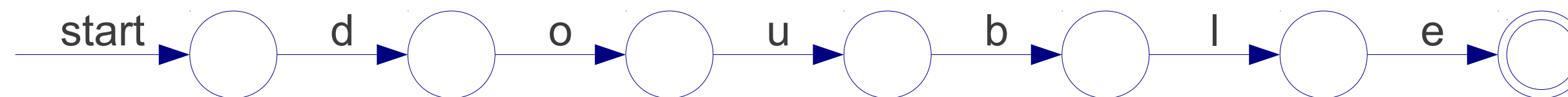
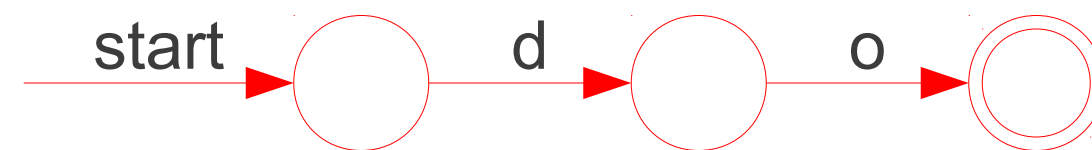
do

T\_Double

double

T\_Mystery

[A-Za-z]



# Implementing Maximal Munch

T\_Do

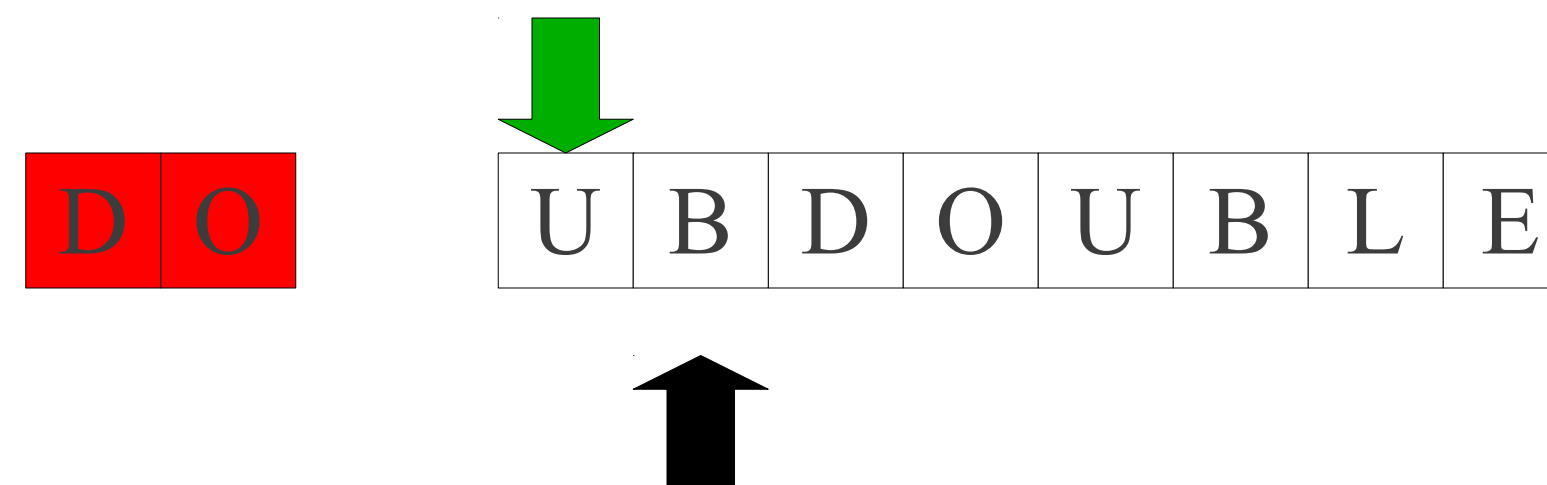
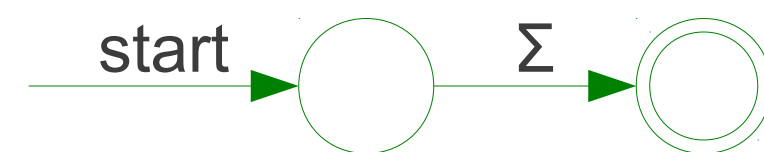
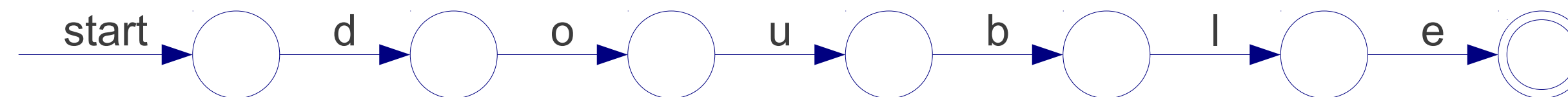
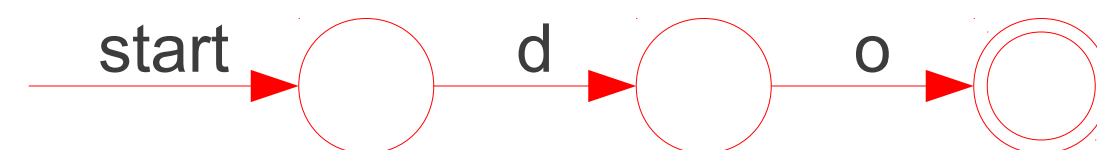
do

T\_Double

double

T\_Mystery

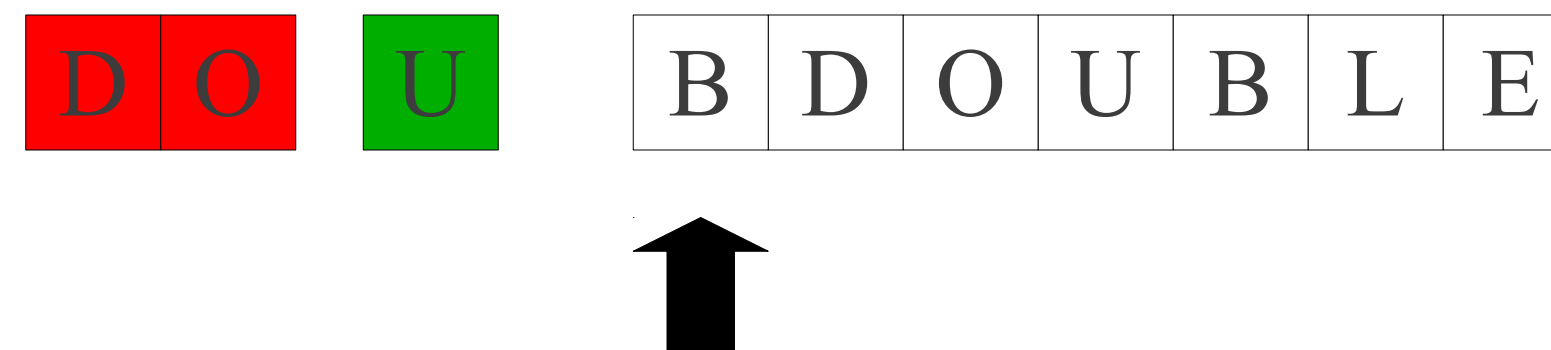
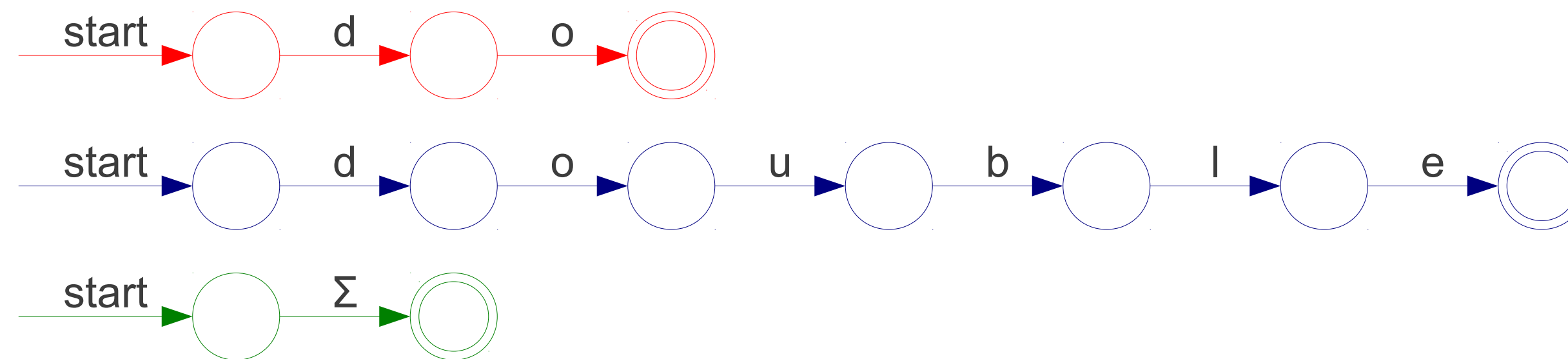
[A-Za-z]





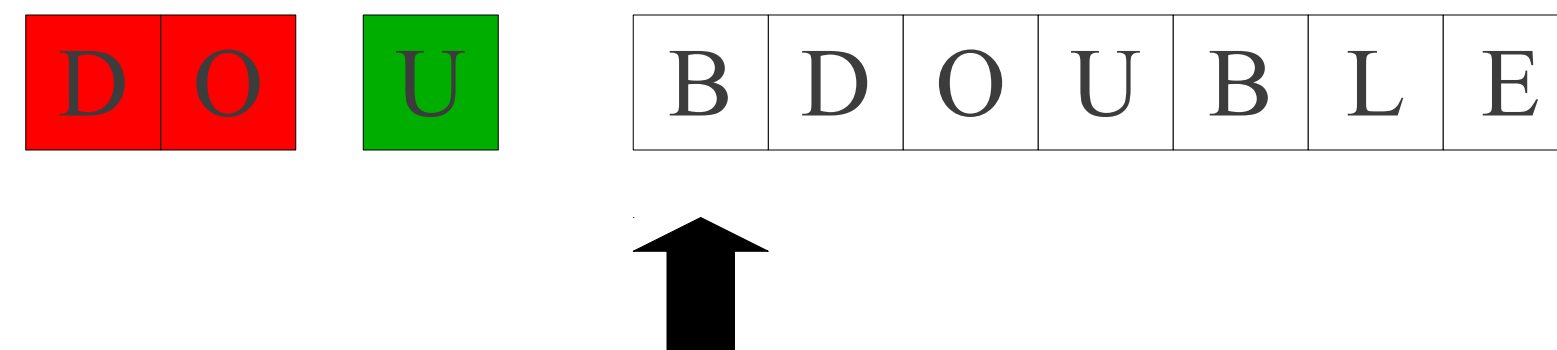
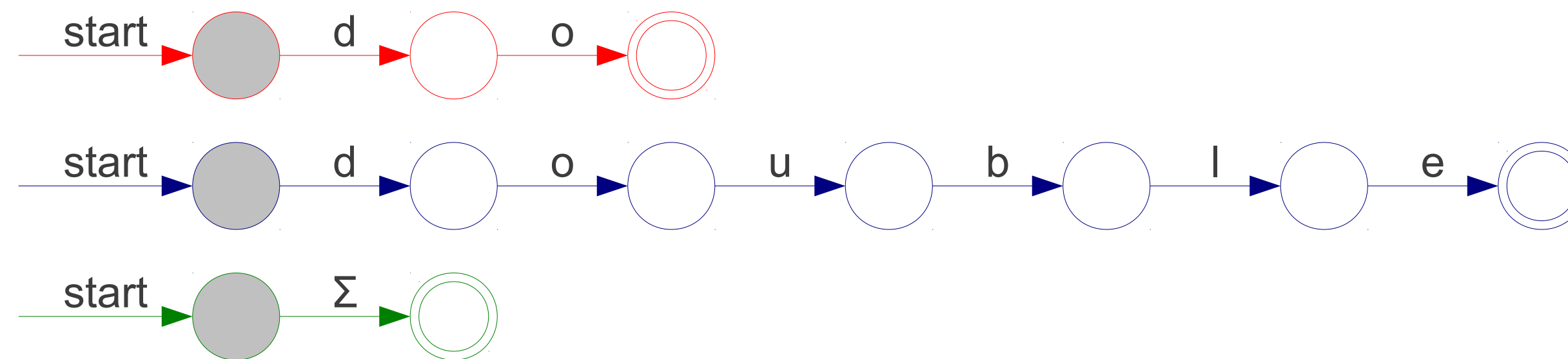
# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery      [A-Za-z]



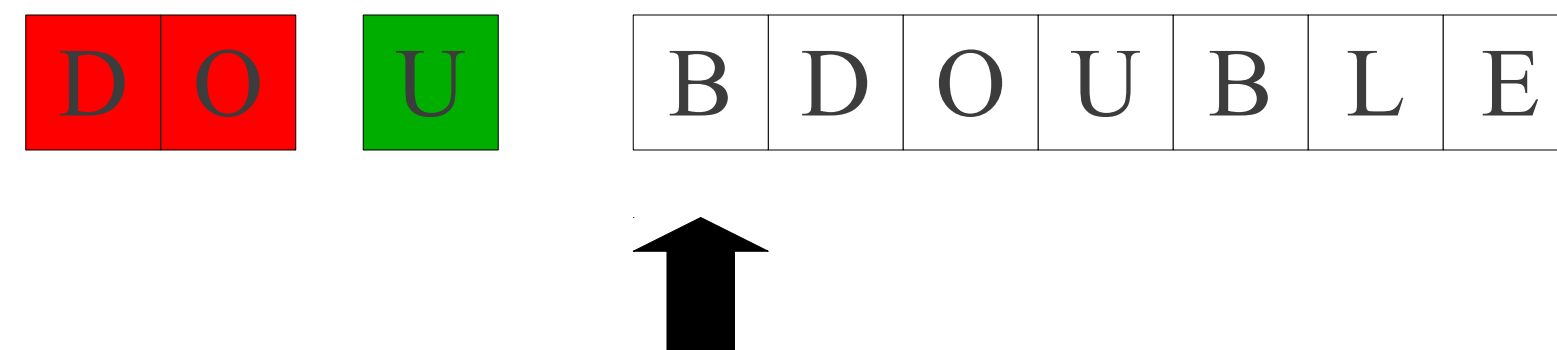
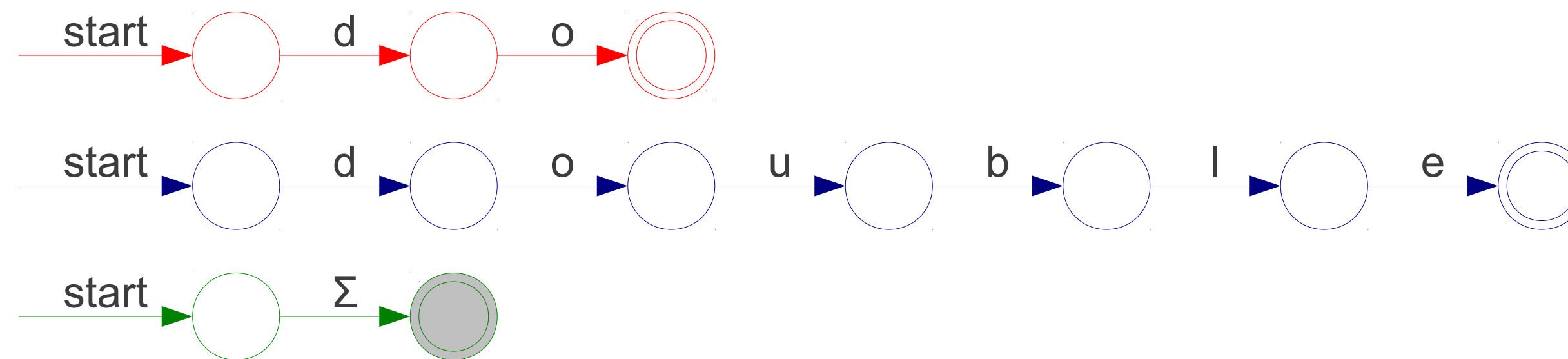
# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery       [A-Za-z]



# Implementing Maximal Munch

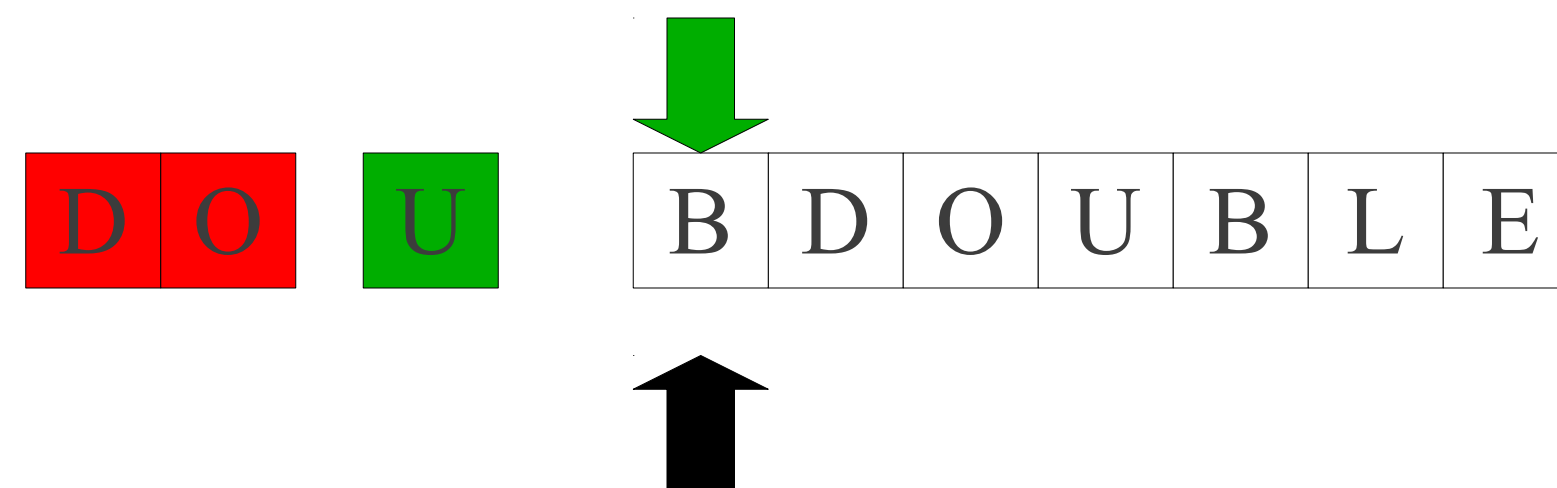
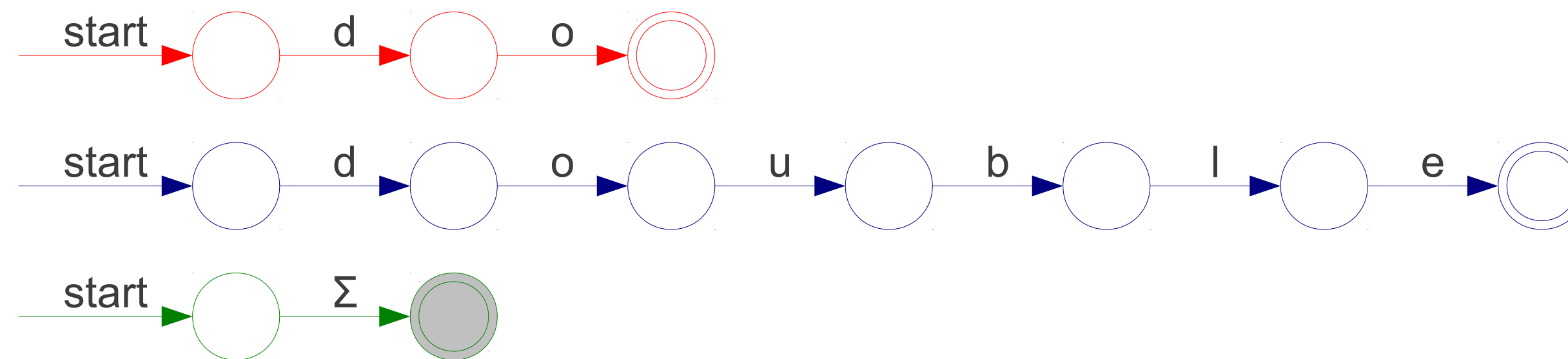
T\_Do            do  
T\_Double       double  
T\_Mystery       [A-Za-z]





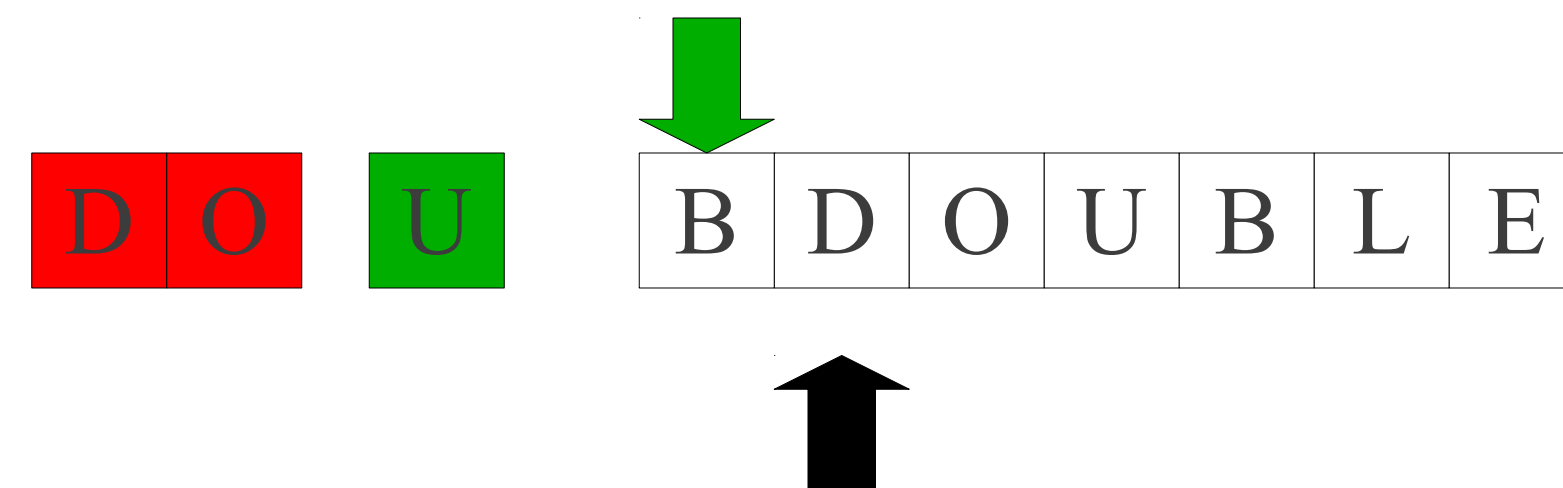
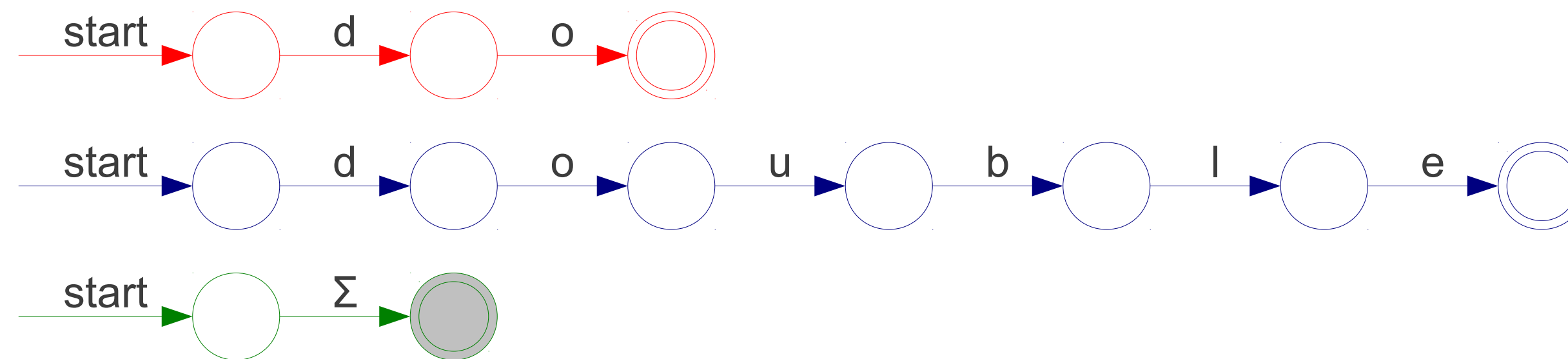
# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery       [A-Za-z]



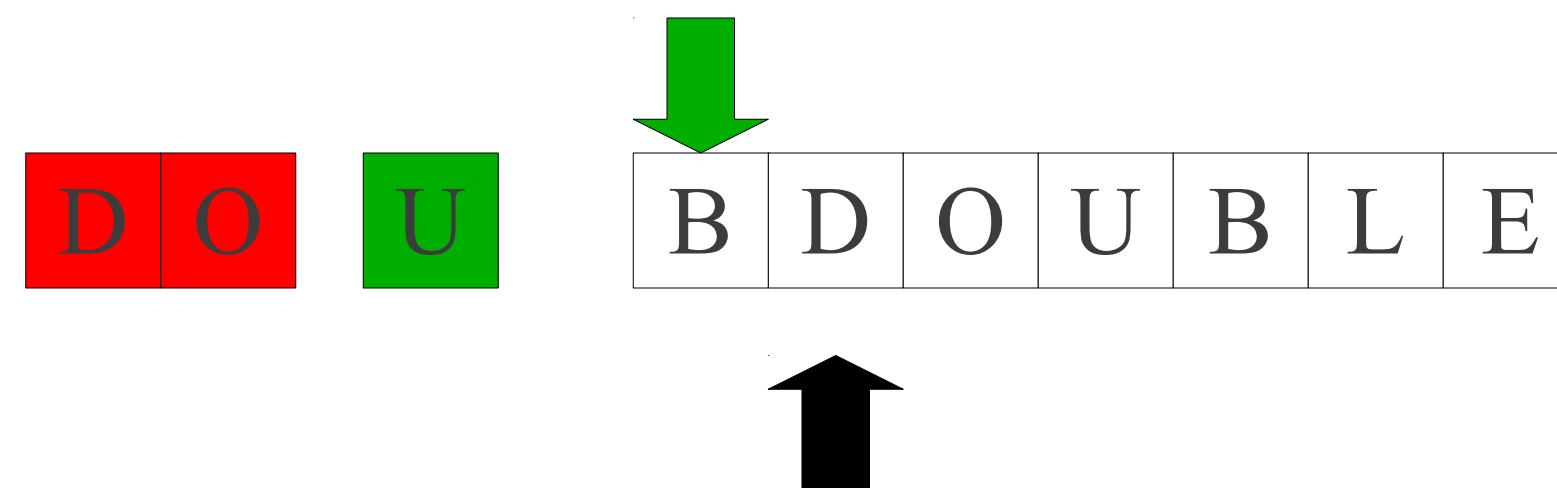
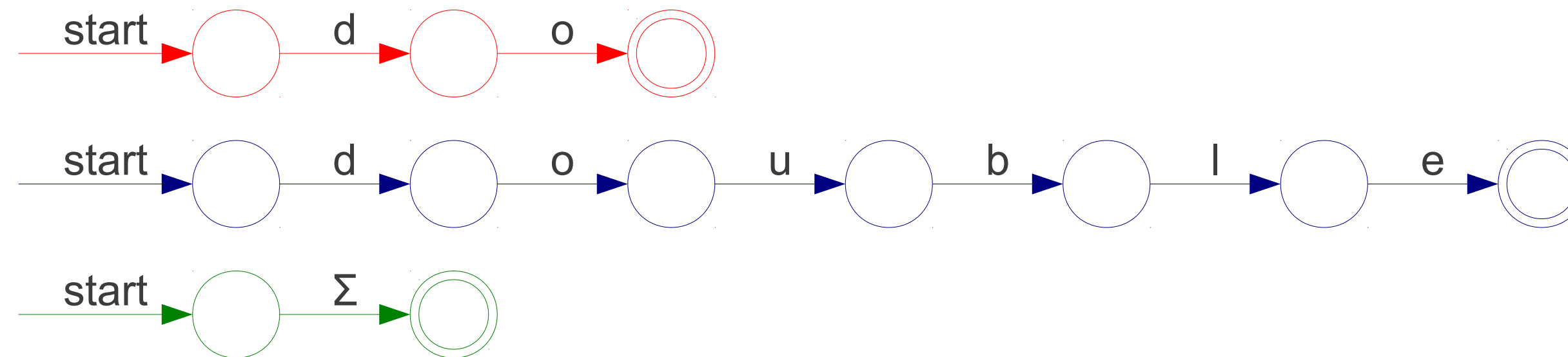
# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery      [A-Za-z]



# Implementing Maximal Munch

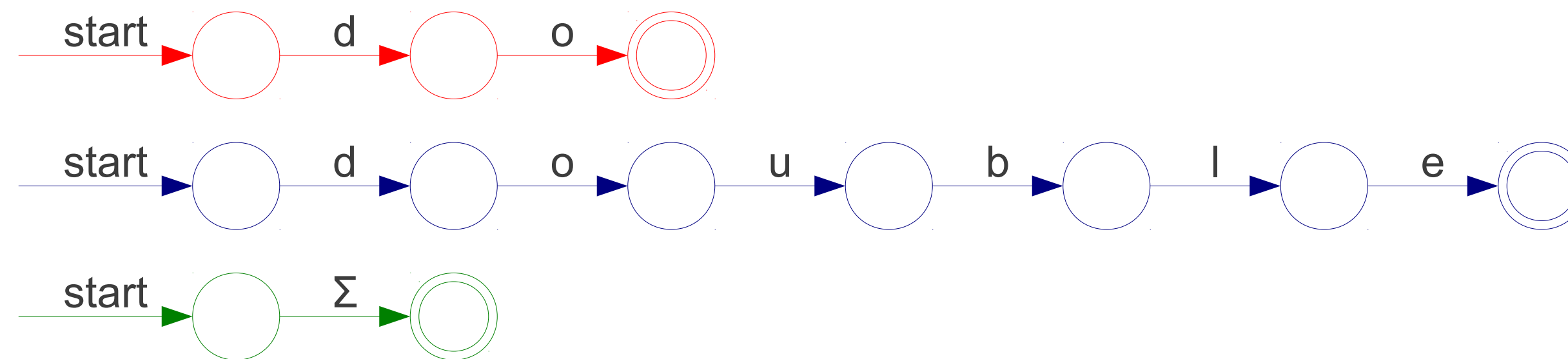
T\_Do            do  
T\_Double       double  
T\_Mystery      [A-Za-z]



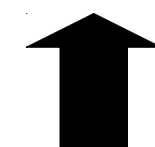


# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery      [A-Za-z]

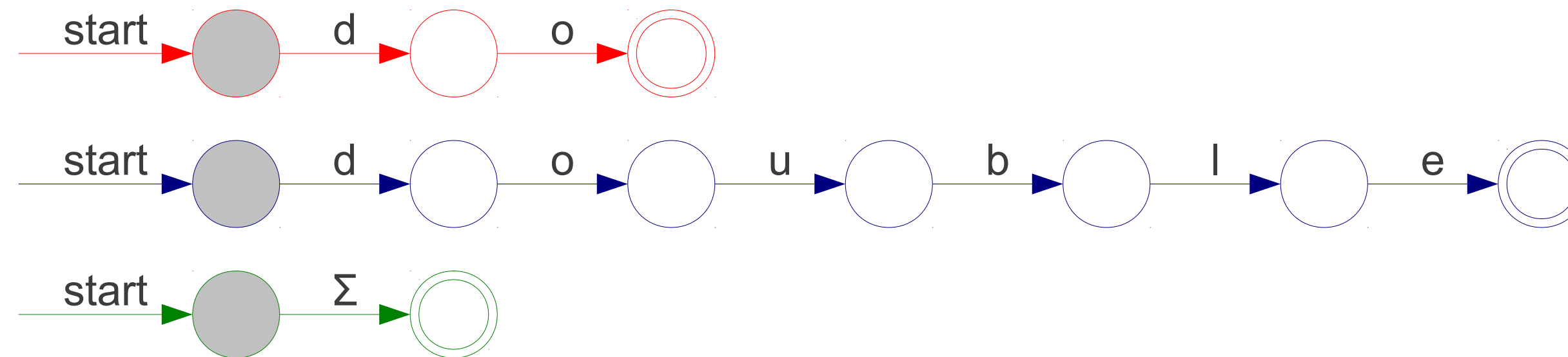


|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| D | O | U | B | D | O | U | B | L | E |
|---|---|---|---|---|---|---|---|---|---|

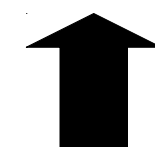


# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery       [A-Za-z]



|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| D | O | U | B | D | O | U | B | L | E |
|---|---|---|---|---|---|---|---|---|---|





# Implementing Maximal Munch

T\_Do

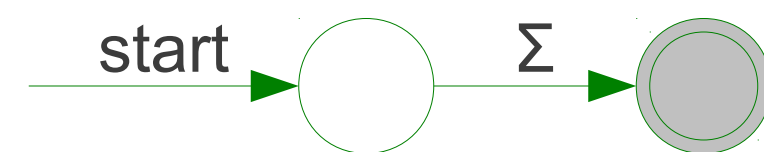
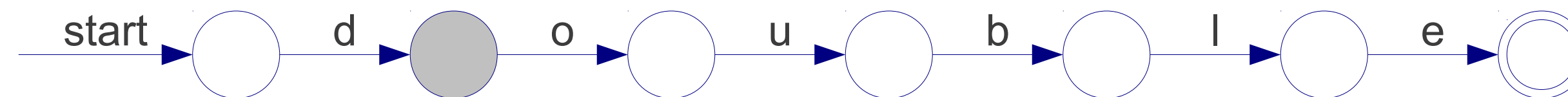
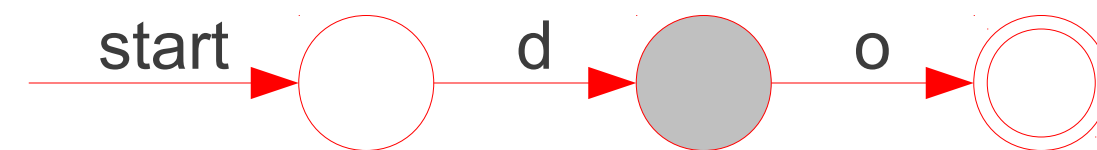
do

T\_Double

double

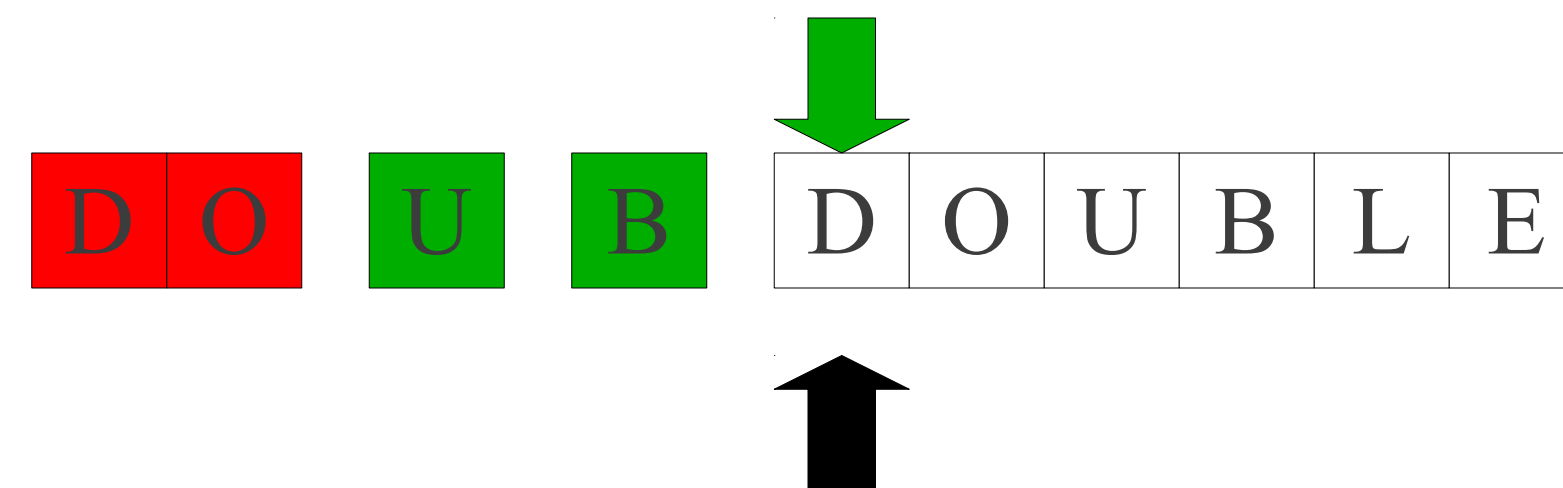
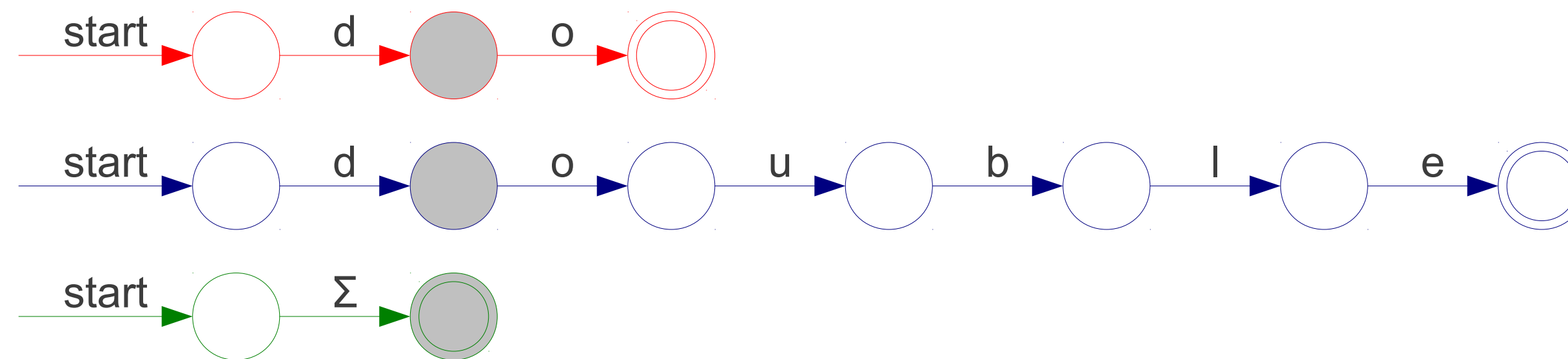
T\_Mystery

[A-Za-z]



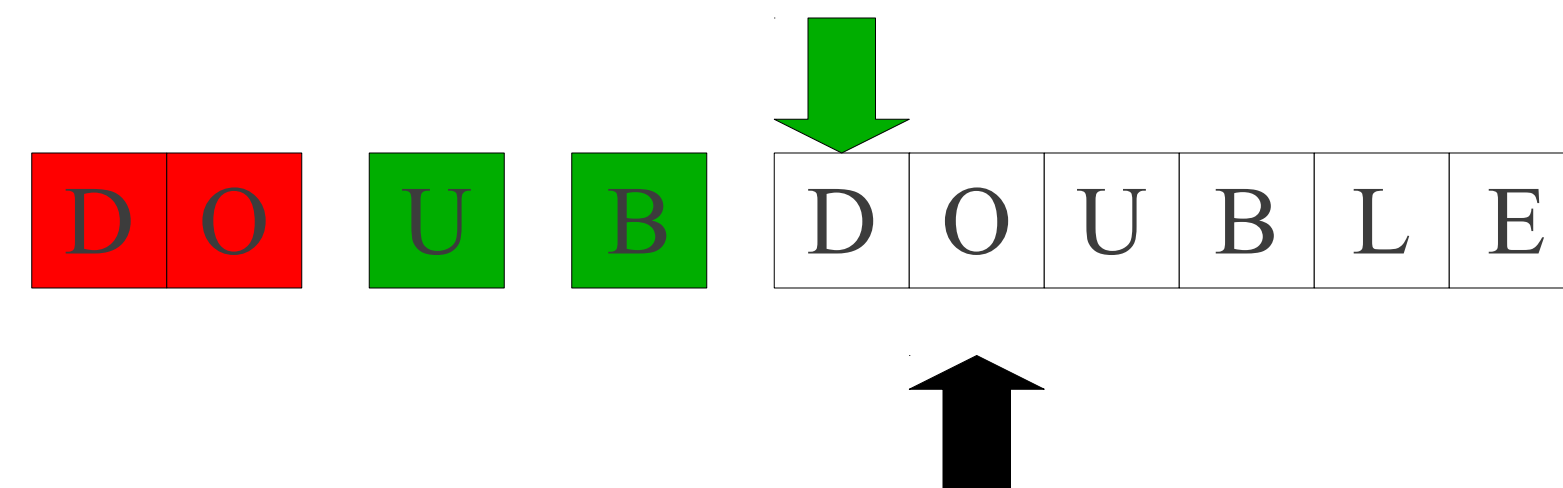
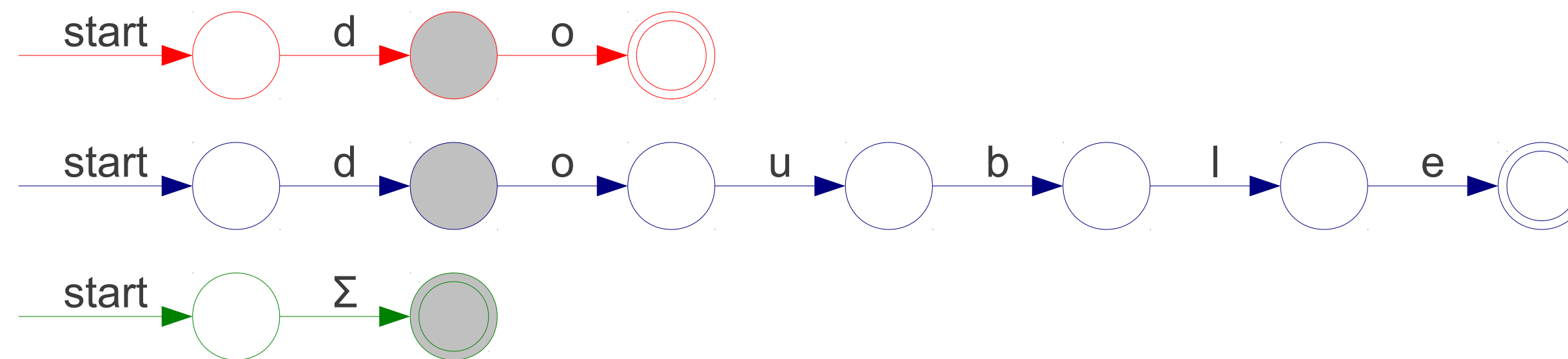
# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery      [A-Za-z]



# Implementing Maximal Munch

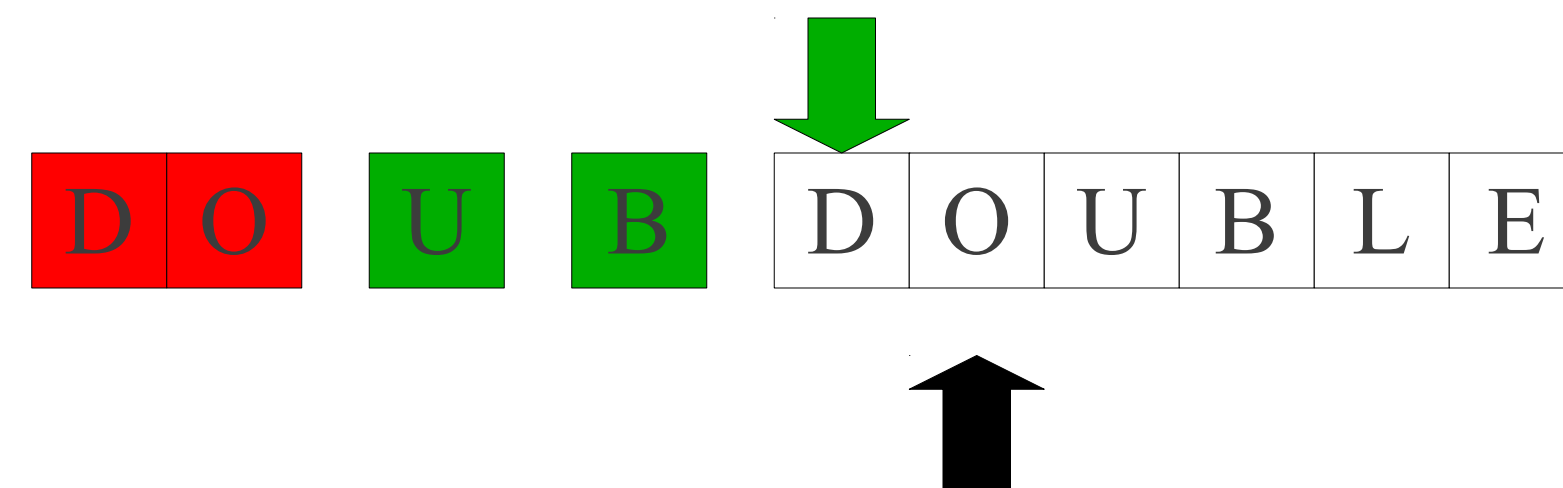
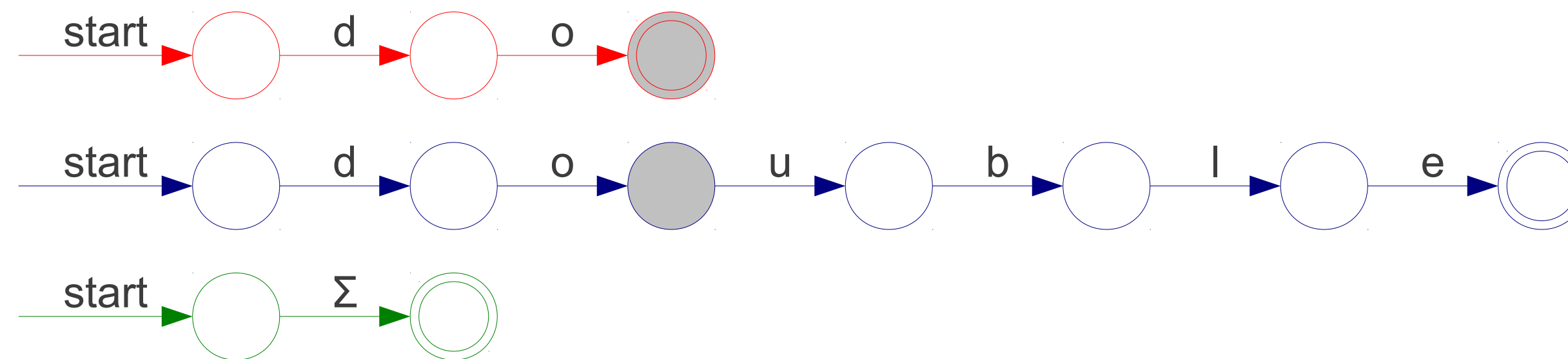
T\_Do            do  
T\_Double       double  
T\_Mystery       [A-Za-z]





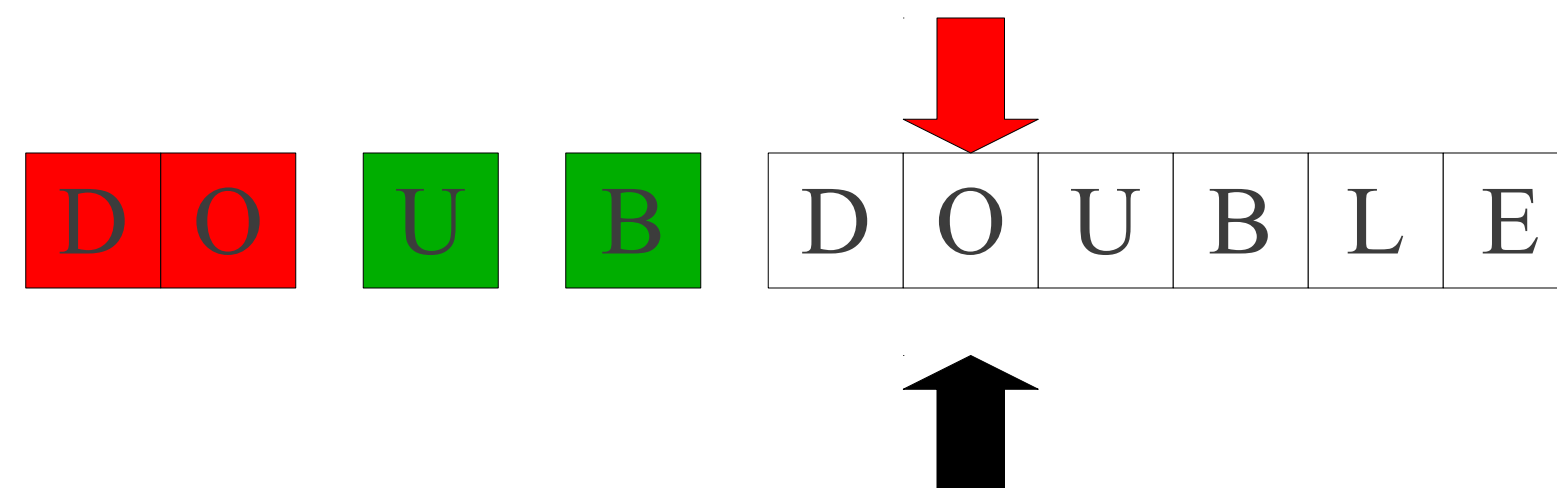
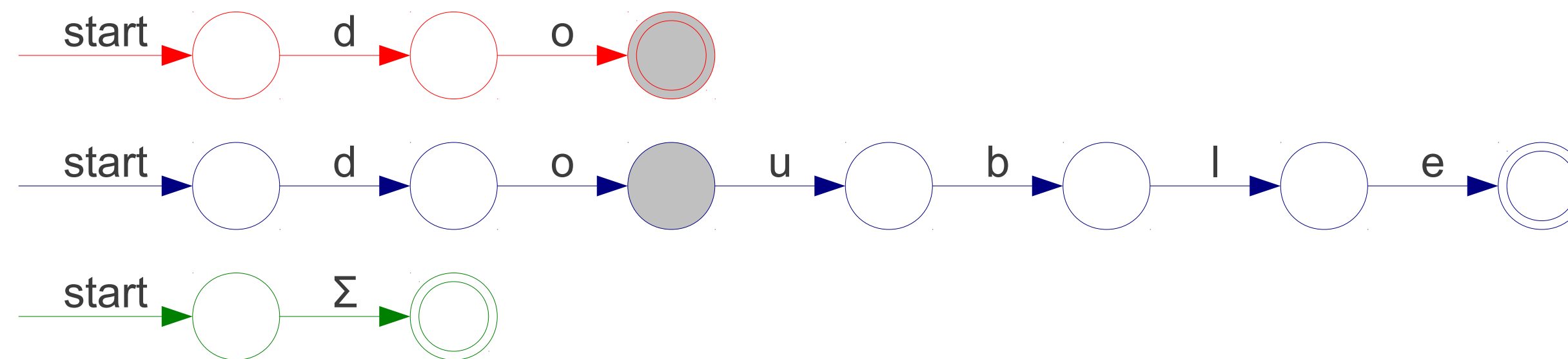
# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery       [A-Za-z]



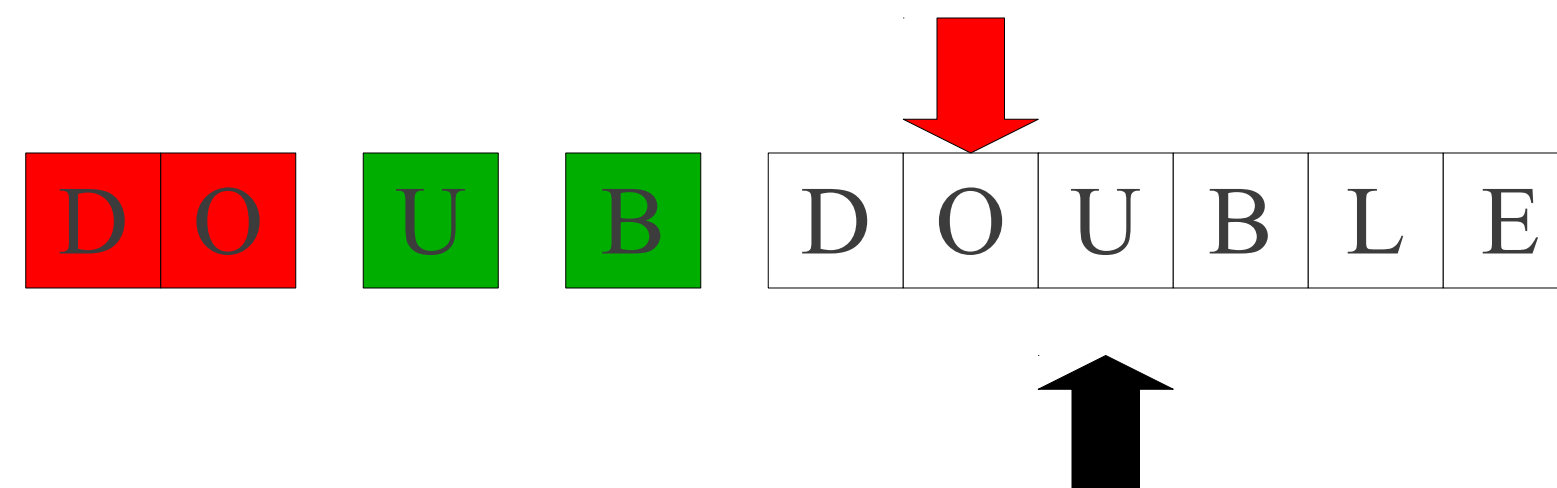
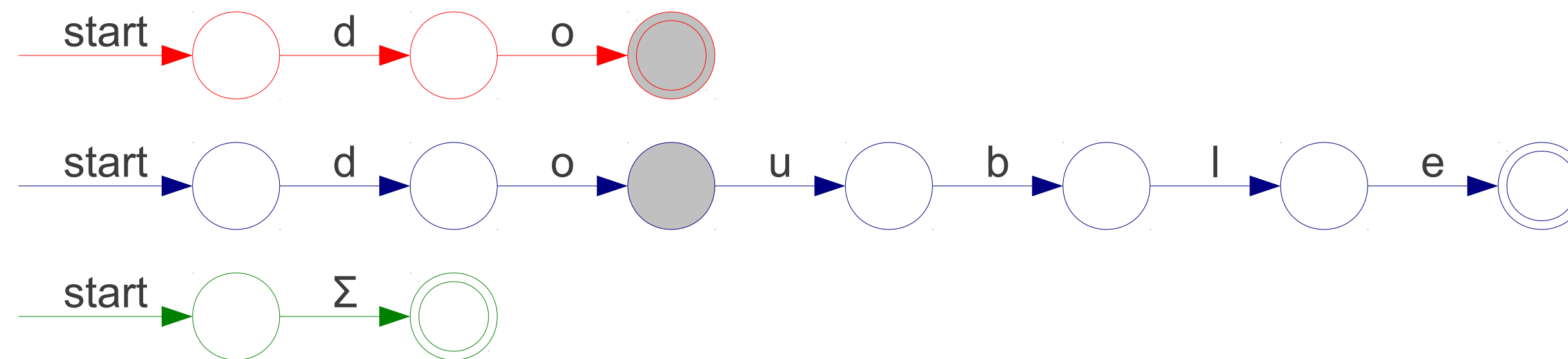
# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery       [A-Za-z]



# Implementing Maximal Munch

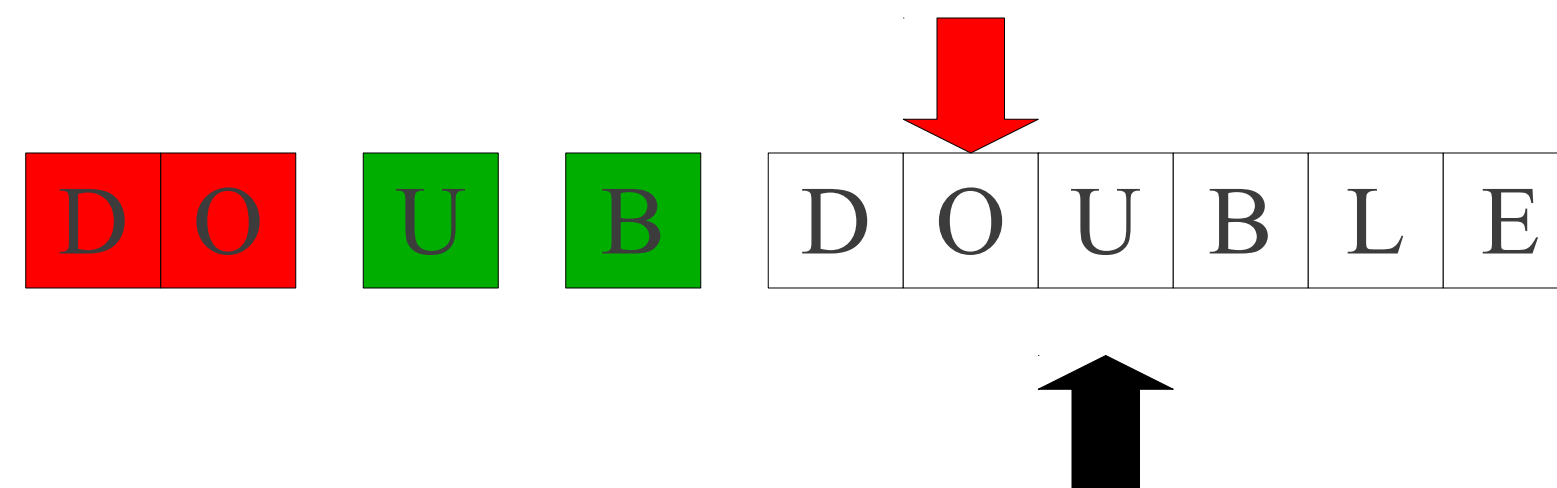
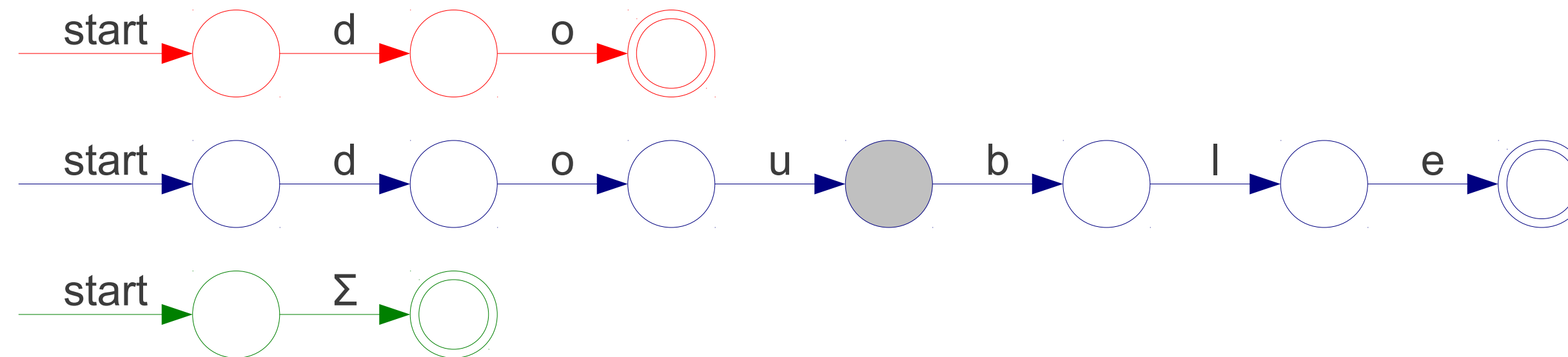
T\_Do            do  
T\_Double       double  
T\_Mystery      [A-Za-z]





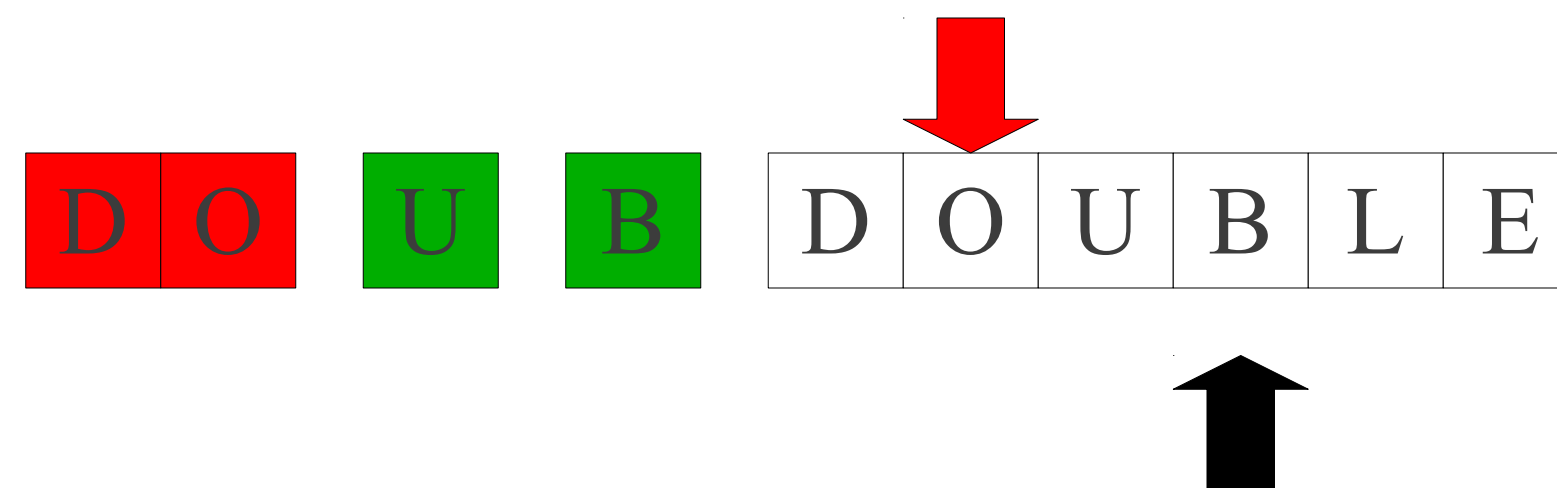
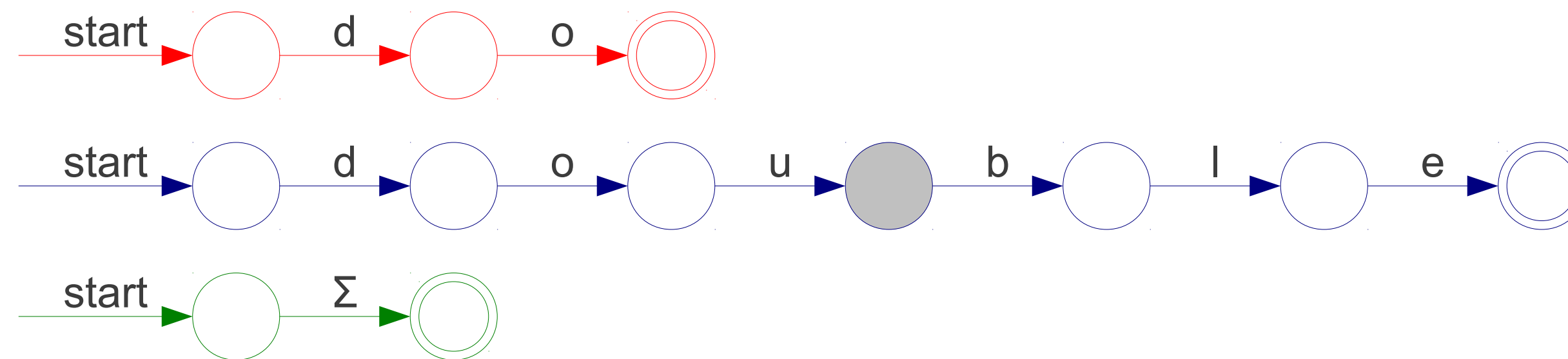
# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery       [A-Za-z]



# Implementing Maximal Munch

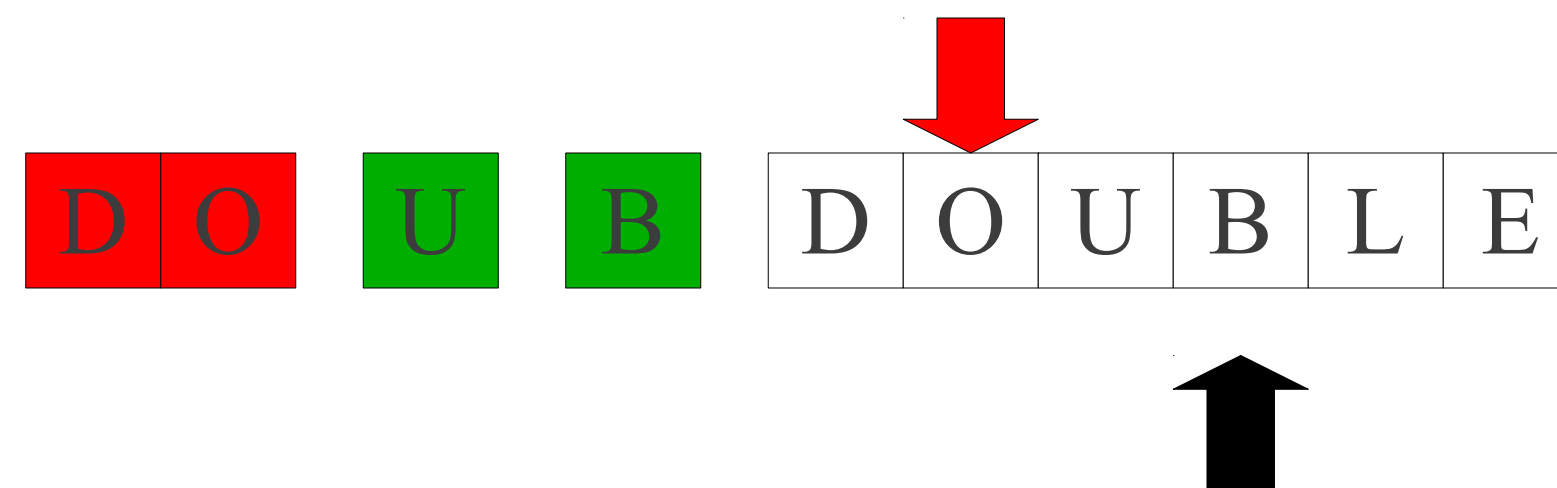
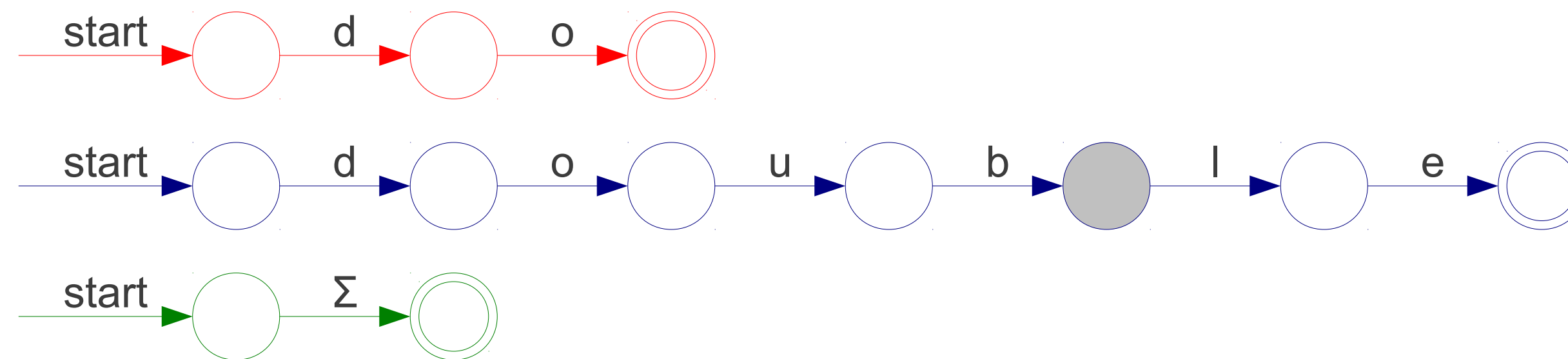
T\_Do            do  
T\_Double       double  
T\_Mystery      [A-Za-z]





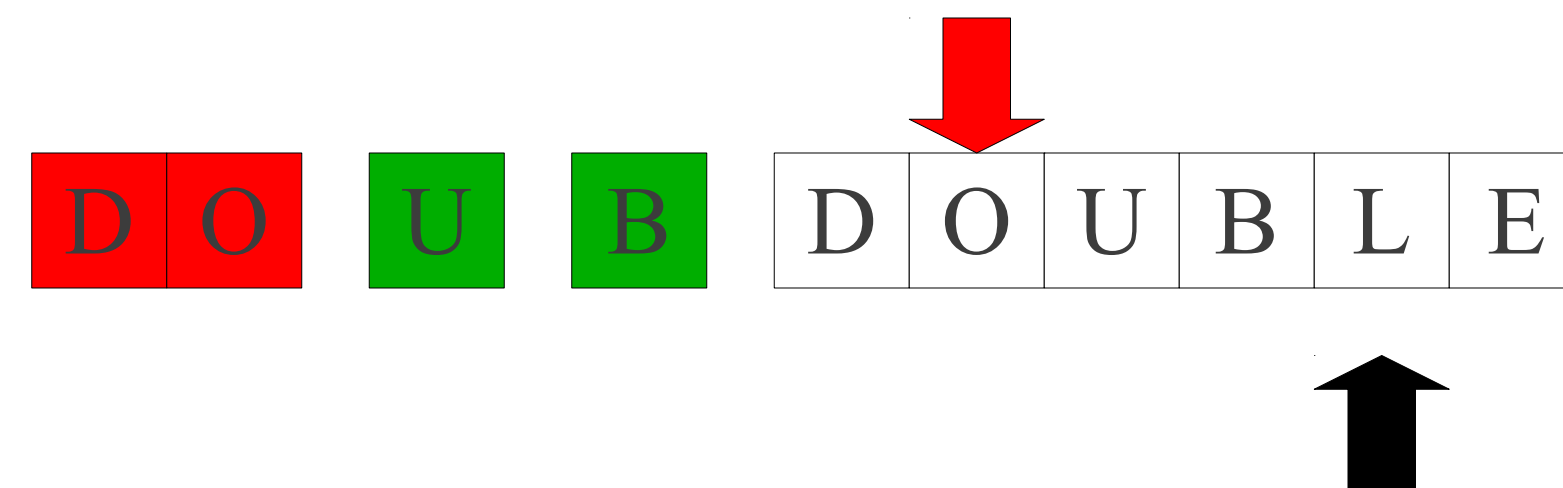
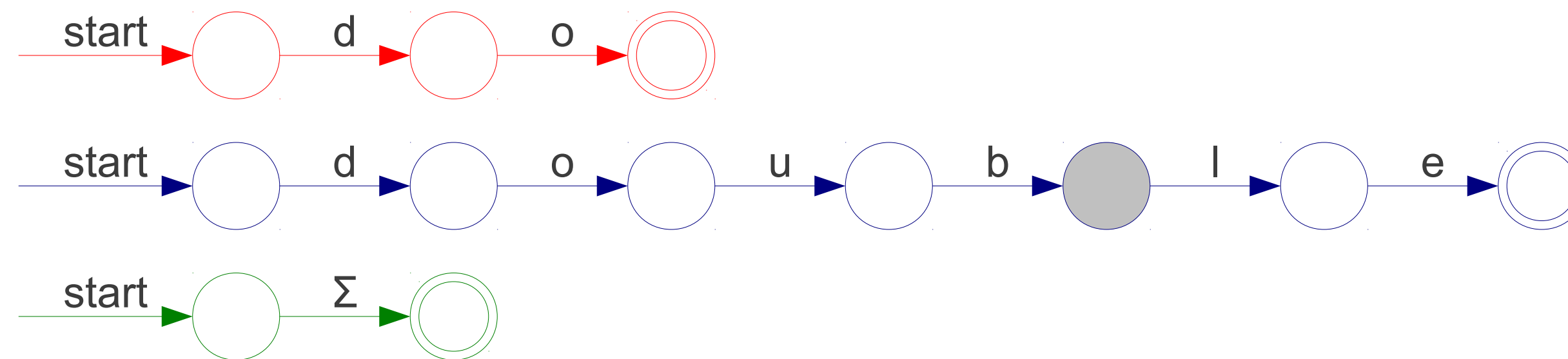
# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery      [A-Za-z]



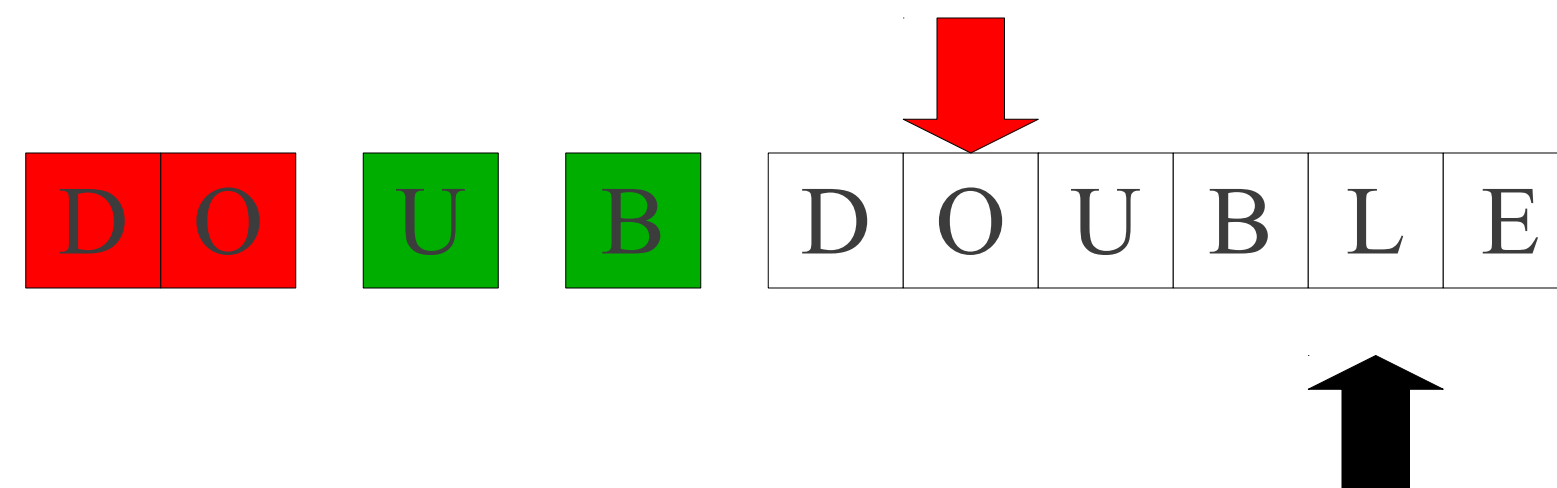
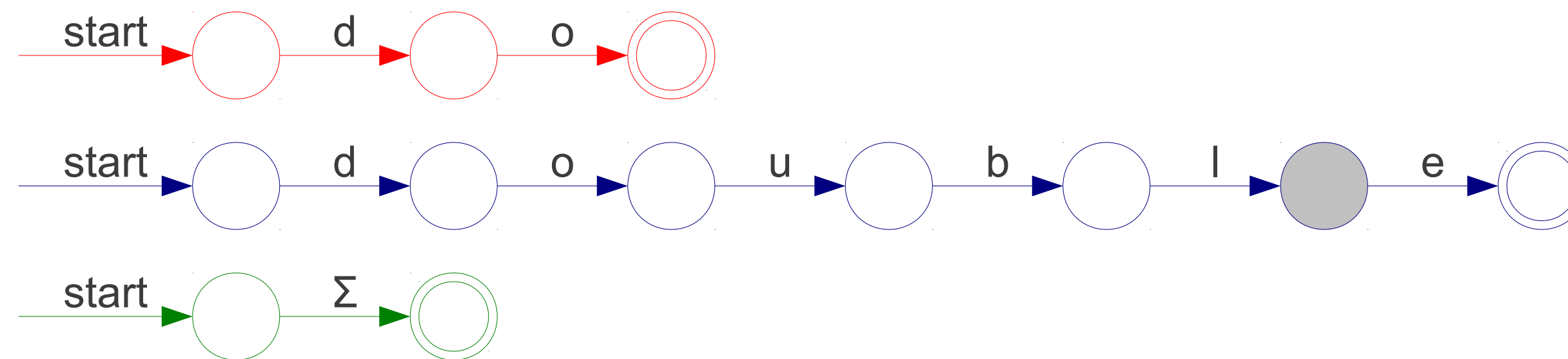
# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery       [A-Za-z]



# Implementing Maximal Munch

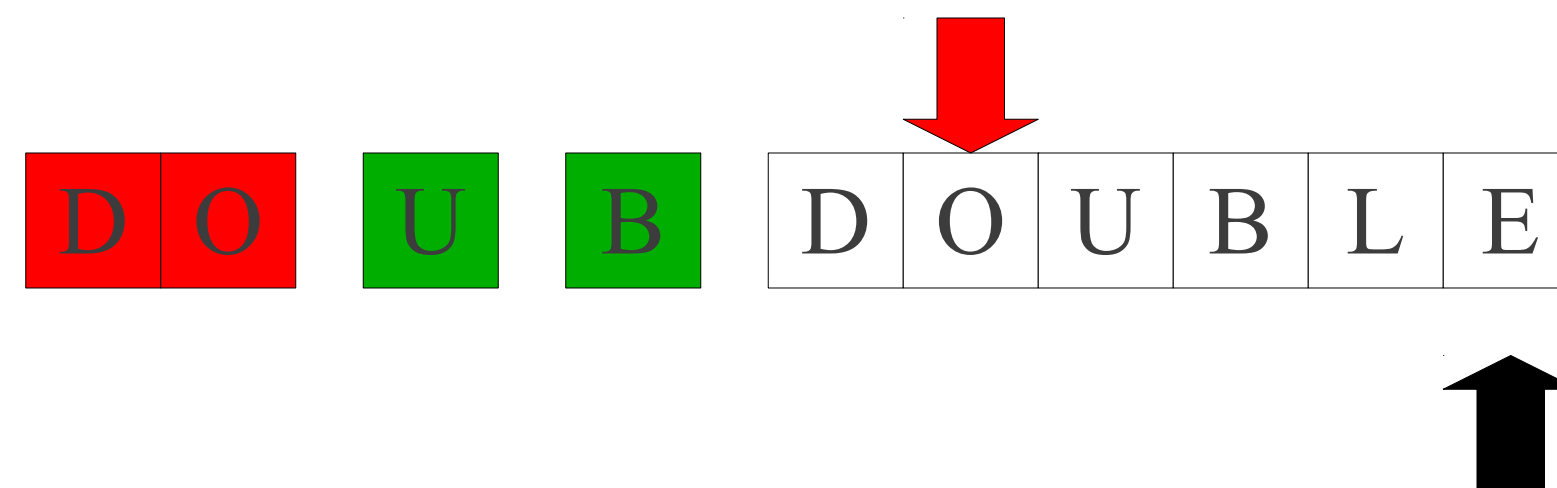
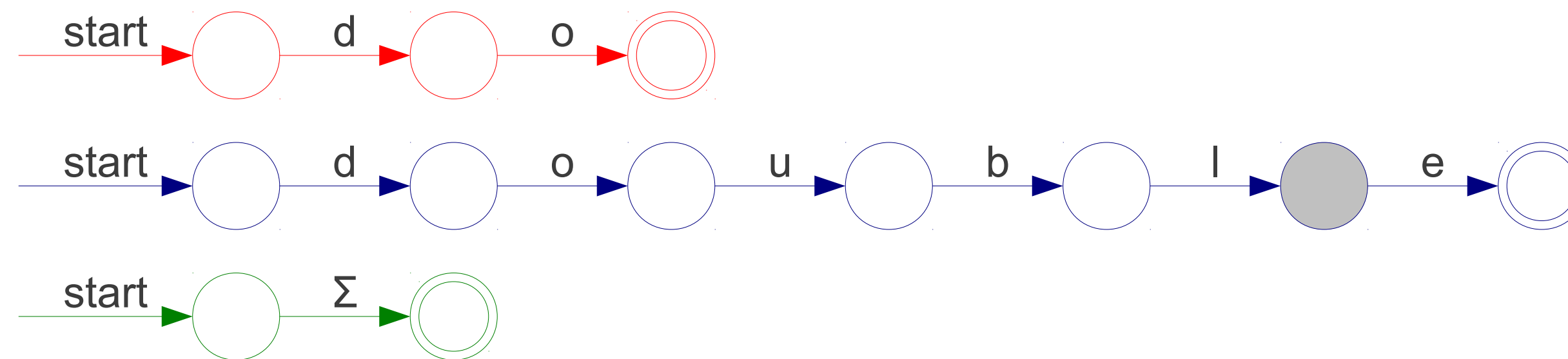
T\_Do            do  
T\_Double       double  
T\_Mystery      [A-Za-z]





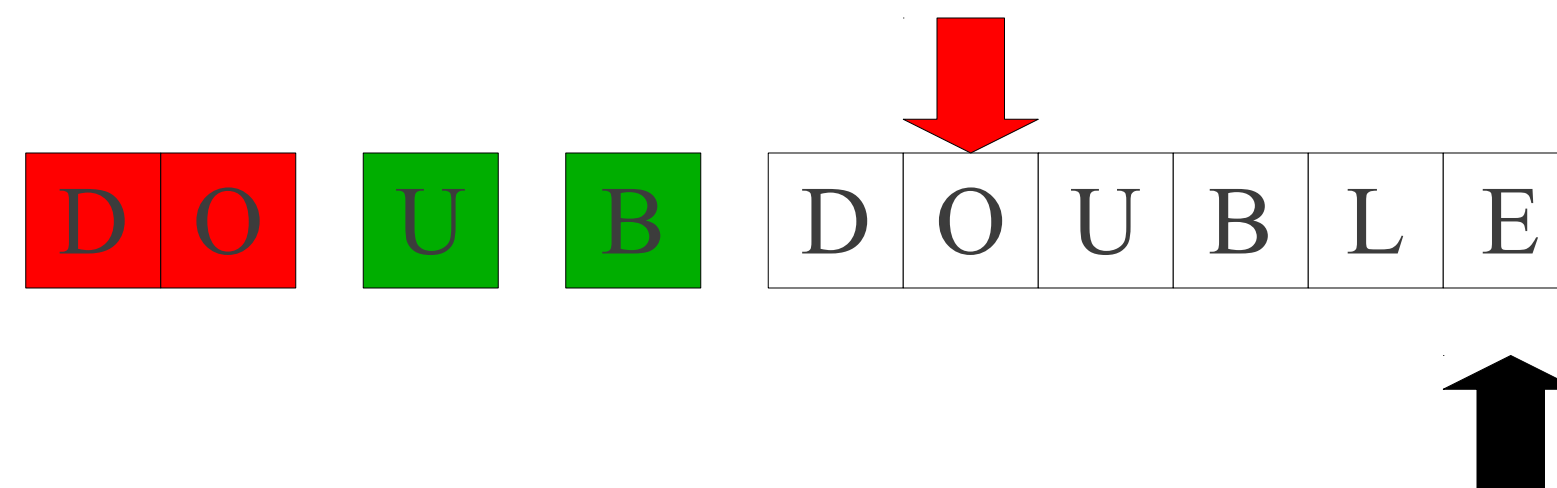
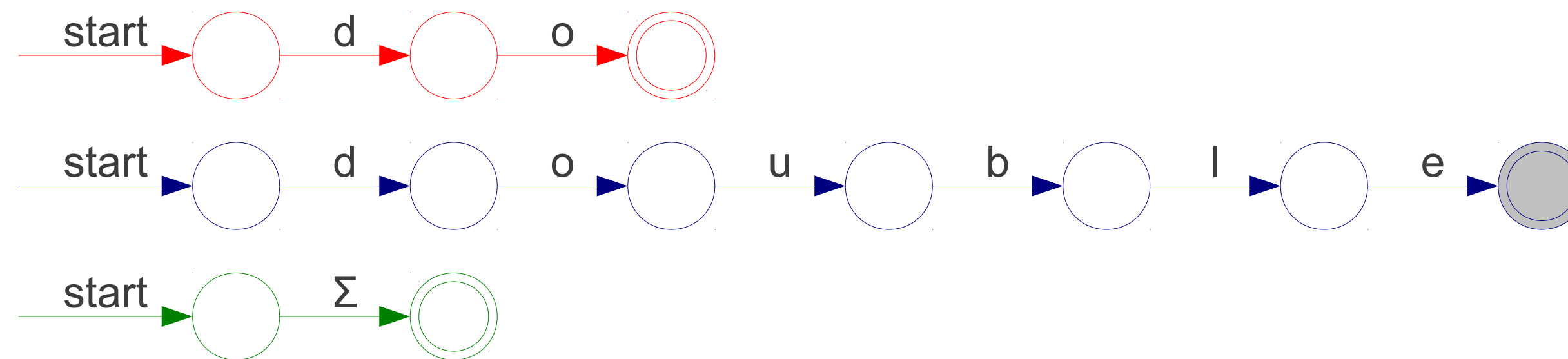
# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery      [A-Za-z]



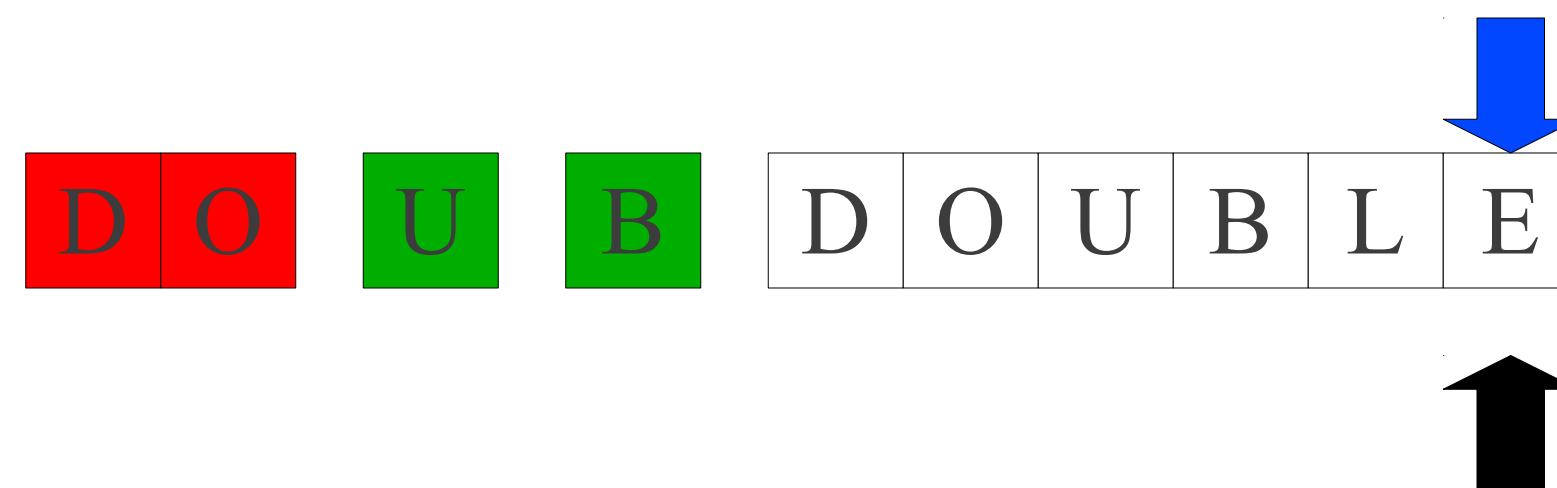
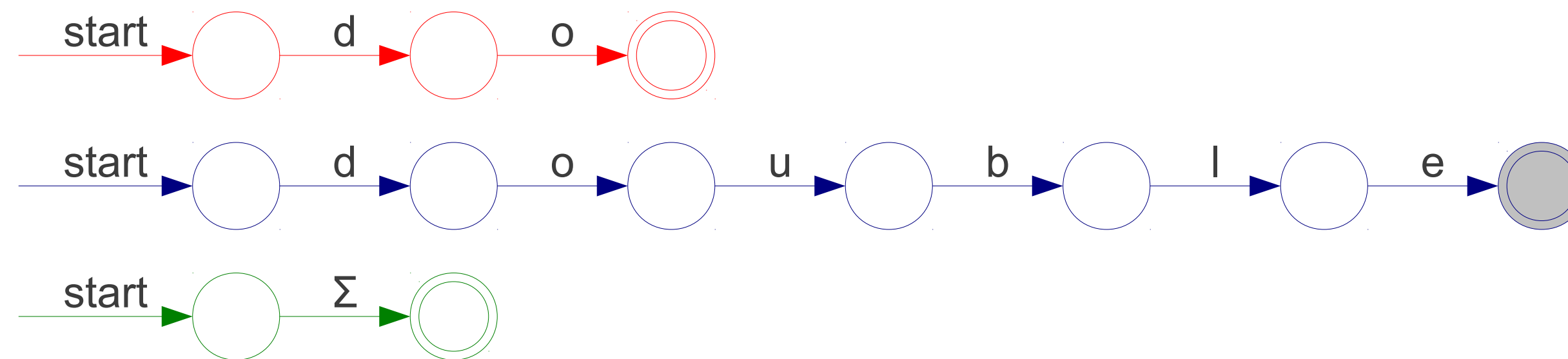
# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery      [A-Za-z]



# Implementing Maximal Munch

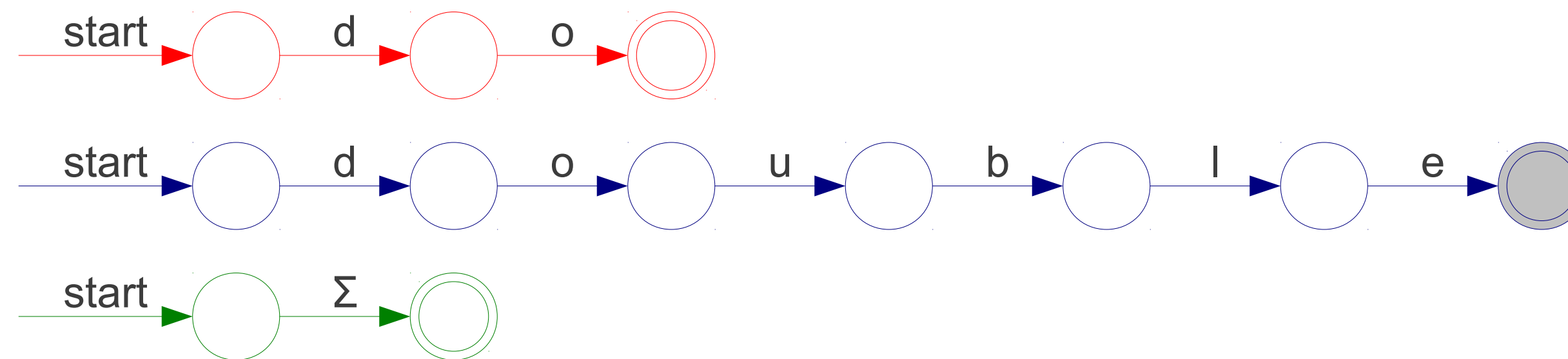
T\_Do            do  
T\_Double       double  
T\_Mystery      [A-Za-z]





# Implementing Maximal Munch

T\_Do            do  
T\_Double       double  
T\_Mystery      [A-Za-z]



D O U B D O U B L E

# Other Conflicts

T\_Do do  
T\_Double double  
T\_Identifier [A-Za-z\_] [A-Za-z0-9\_]\*

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| d | o | u | b | l | e |
|---|---|---|---|---|---|

disambiguate with an ordering between the choices



# Summary

- Lexers scan the input program, grouping substrings into higher level **tokens**
- Lexing is tedious and error-prone to implement manually. Just like assembly code!
- Can use a **lexer generator**, a compiler for a domain-specific language for lexers.
- Use **regular expressions** as syntax, **finite automata** as intermediate representation
- Widely available tools, well known algorithms
  - Caveat: computationally limited, not powerful enough for parsing or semantic analysis. New formalisms next time!

# **Syntactic Analysis (Parsing)**

# Parsing: Finding Syntactic Structure

```
{  
  if (b == 0) a = b;  
  while (a != 1) {  
    print_int(a);  
    a = a - 1;  
  }  
}
```

Source input

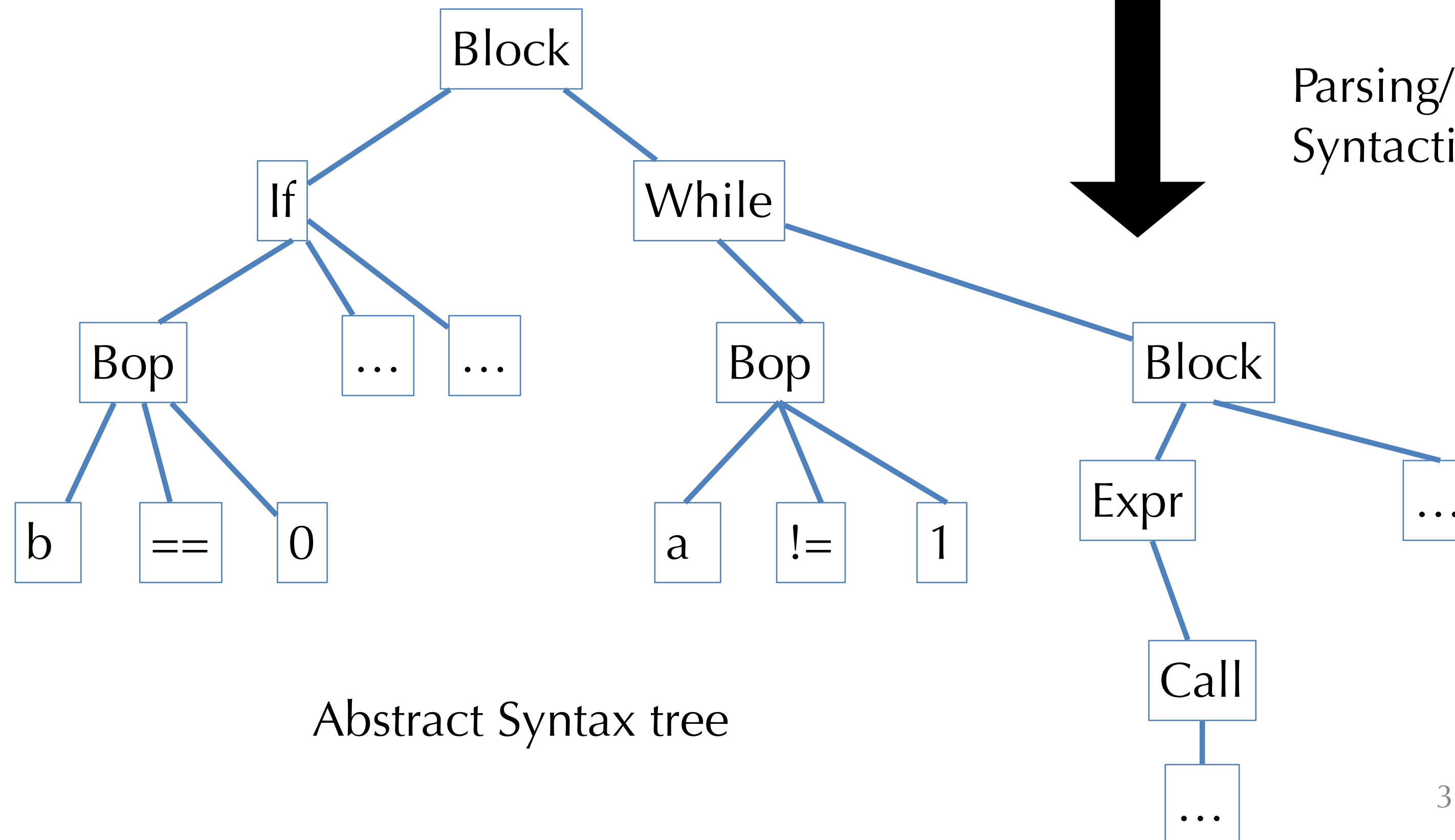
Lexing/  
Lexical Analysis



```
LB IF LP Ident(b) EQ EQ  
Num(0) RP Ident(a) EQ Ident(b)  
SEMI WHILE LP Ident(a) NEQ  
Num(1) RP LB Ident(print_int)  
LP Var(a) RP SEMI Ident(a) EQ  
Ident(a) MINUS Number(1)  
SEMI RB RB
```

Token Stream

Parsing/  
Syntactic Analysis



Abstract Syntax tree



# Syntactic Analysis aka Parsing

The task of the syntactic analysis is to produce an abstract syntax tree from a token stream (the output from lexing), rejecting the input if it is not well-formed.

Similarities to lexing:

1. Input is a string (of tokens rather than characters)
2. Need to identify \*if\* the input is well-formed (language recognition)
3. Need to decide which of possibly multiple outputs to produce (ambiguity)

Differences:

1. Output is a **tree** rather than a string
2. Formalisms for lexing are too weak (regular expressions and finite automata)
3. Reason about ambiguity more directly in our grammar formalism (rather than only relying on tie-break rules, longest match conventions)

# Parser Generators as Compilers

Writing parsers by hand is difficult, so we often use **parser generators**, similar to our lexer generators.

Adapt our formula for parser generators:

1. Design a source **language** for parsers: **context-free grammars**
2. Describe its **semantics**: **CFGs** define the set of **parse trees** for every string (not just a formal language)
3. Transform into an intermediate representation: **stack-based automata**
4. **Generate** code from the IR

# Parser Generators as Compilers

Writing parsers by hand is difficult, so we often use **parser generators**, similar to our lexer generators.

The theory for parser generators is not as clean as lexer generators. Because parser generators are more powerful, they are also more computationally expensive to analyze and compile efficiently.

In practice we work with restricted versions of CFGs that support efficient algorithms and are flexible in practice.



# Limitations of Regular Languages

Regular expressions and finite automata are not powerful enough to identify if an input string is well-formed for realistic programming languages.

Example:

Languages include delimited expressions like

parentheses `"(1 + 2 + (3 + 4)) + 5"`

braces `"fn foo() { 1 }"`

And reject unbalanced examples

`"(1 + 2 + (3 + 4) + 5"` is not valid, unmatched left paren.

# Limitations of Regular Languages

Regular expressions and finite automata are not powerful enough to identify if an input string is well-formed for realistic programming languages.

Example:

Dyck language over just two character alphabet '(' and ')' consists of those words where the parentheses are balanced.

E.g., "(()()(((0))))" but not "(())(())"

Parsing a real language is at least as hard as parsing the Dyck language.

Theorem: The Dyck language is not regular, i.e., there is no regular expression expressing it.



# Limitations of Regular Languages

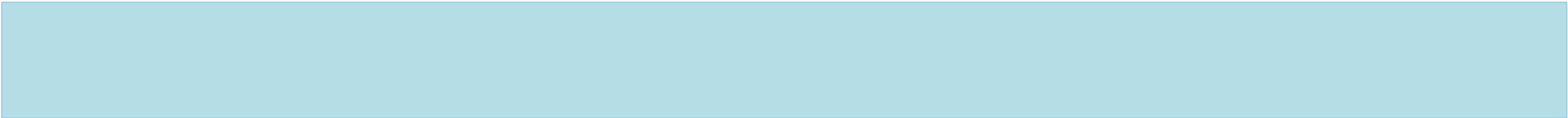
Theorem: The Dyck language is not regular, i.e., there is no regular expression expressing it.

Proof:

Idea: Regular expressions are equivalent to finite automata, which have a finite number of states. Finite states means you can only count so high.

Consider a very long string of open parentheses  $(((((\dots n \text{ times where } n \text{ is } > \text{the number of states of the automaton. Since } n \text{ is } > \text{the number of states, there must be some smaller prefix } (((\dots m \text{ where } m < n \text{ and the automaton reaches the same state.}$

If the automaton decides the Dyck language, then  $((\dots m))) \dots m$  is accepted, but therefore also  $((\dots n))) \dots m$  is also accepted, which is incorrect.



# CONTEXT FREE GRAMMARS

# Context-free Grammars

- Here is a specification of the language of balanced parens:

$$S \mapsto (S)S$$

$$S \mapsto \epsilon$$

Note: Once again we have to take care to distinguish meta-language elements (e.g. “S” and “ $\mapsto$ ”) from object-language elements (e.g. “(” ).\*

- The definition is *recursive* – S mentions itself.
- Idea: “derive” a string in the language by starting with S and rewriting according to the rules:
  - Example:  $S \mapsto (S)S \mapsto ((S)S)S \mapsto ((\epsilon)S)S \mapsto ((\epsilon)S)\epsilon \mapsto ((\epsilon)\epsilon)\epsilon = (())$
- You can replace the “*nonterminal*” S by one of its definitions anywhere
- A context-free grammar accepts a string iff there is a derivation from the start symbol

\* And, since we’re writing this description in English, we are careful distinguish the meta-meta-language (e.g. words) from the meta-language and object-language (e.g. symbols) by using quotes. 403



# CFGs Mathematically

- A Context-free Grammar (CFG) consists of
  - A set of *terminals* (e.g., a lexical token or  $\epsilon$ )
  - A set of *nonterminals* (e.g.,  $S$  and other syntactic variables)
  - A designated nonterminal called the *start symbol*
  - A set of productions:  $\text{LHS} \mapsto \text{RHS}$ 
    - LHS is a nonterminal
    - RHS is a *string* of terminals and nonterminals
- Example: The balanced parentheses language:

$$S \mapsto (S)S$$

$$S \mapsto \epsilon$$

- How many terminals? How many nonterminals? Productions?

# Another Example: Sum Grammar

- A grammar that accepts parenthesized sums of numbers:

$$\begin{array}{lcl} S & \mapsto & E + S \quad | \quad E \\ E & \mapsto & \text{number} \quad | \quad ( S ) \end{array}$$

e.g.:  $(1 + 2 + (3 + 4)) + 5$

- Note the vertical bar ' $|$ ' is shorthand for multiple productions:

|                           |   |                                       |
|---------------------------|---|---------------------------------------|
| $S \mapsto E + S$         | } | 4 productions                         |
| $S \mapsto E$             |   | 2 nonterminals: $S, E$                |
| $E \mapsto \text{number}$ |   | 4 terminals: $(, ), +, \text{number}$ |
| $E \mapsto (S)$           |   | Start symbol: $S$                     |

# Derivations in CFGs

- Example: derive  $(1 + 2 + (3 + 4)) + 5$

- $\underline{S} \mapsto \underline{E} + S$

$$\mapsto (\underline{S}) + S$$

$$\mapsto (\underline{E} + S) + S$$

$$\mapsto (1 + \underline{S}) + S$$

$$\mapsto (1 + \underline{E} + S) + S$$

$$\mapsto (1 + 2 + \underline{S}) + S$$

$$\mapsto (1 + 2 + \underline{E}) + S$$

$$\mapsto (1 + 2 + (\underline{S})) + S$$

$$\mapsto (1 + 2 + (\underline{E} + S)) + S$$

$$\mapsto (1 + 2 + (3 + \underline{S})) + S$$

$$\mapsto (1 + 2 + (3 + \underline{E})) + S$$

$$\mapsto (1 + 2 + (3 + 4)) + \underline{S}$$

$$\mapsto (1 + 2 + (3 + 4)) + \underline{E}$$

$$\mapsto (1 + 2 + (3 + 4)) + 5$$

$$S \mapsto E + S \mid E$$

$$E \mapsto \text{number} \mid ( S )$$

For arbitrary strings  $\alpha, \beta, \gamma$  and production rule  $A \mapsto \beta$  a single step of the derivation is:

$$\alpha A \gamma \mapsto \alpha \beta \gamma$$

( *substitute*  $\beta$  for an occurrence of  $A$  )

In general, there are many possible derivations for a given string

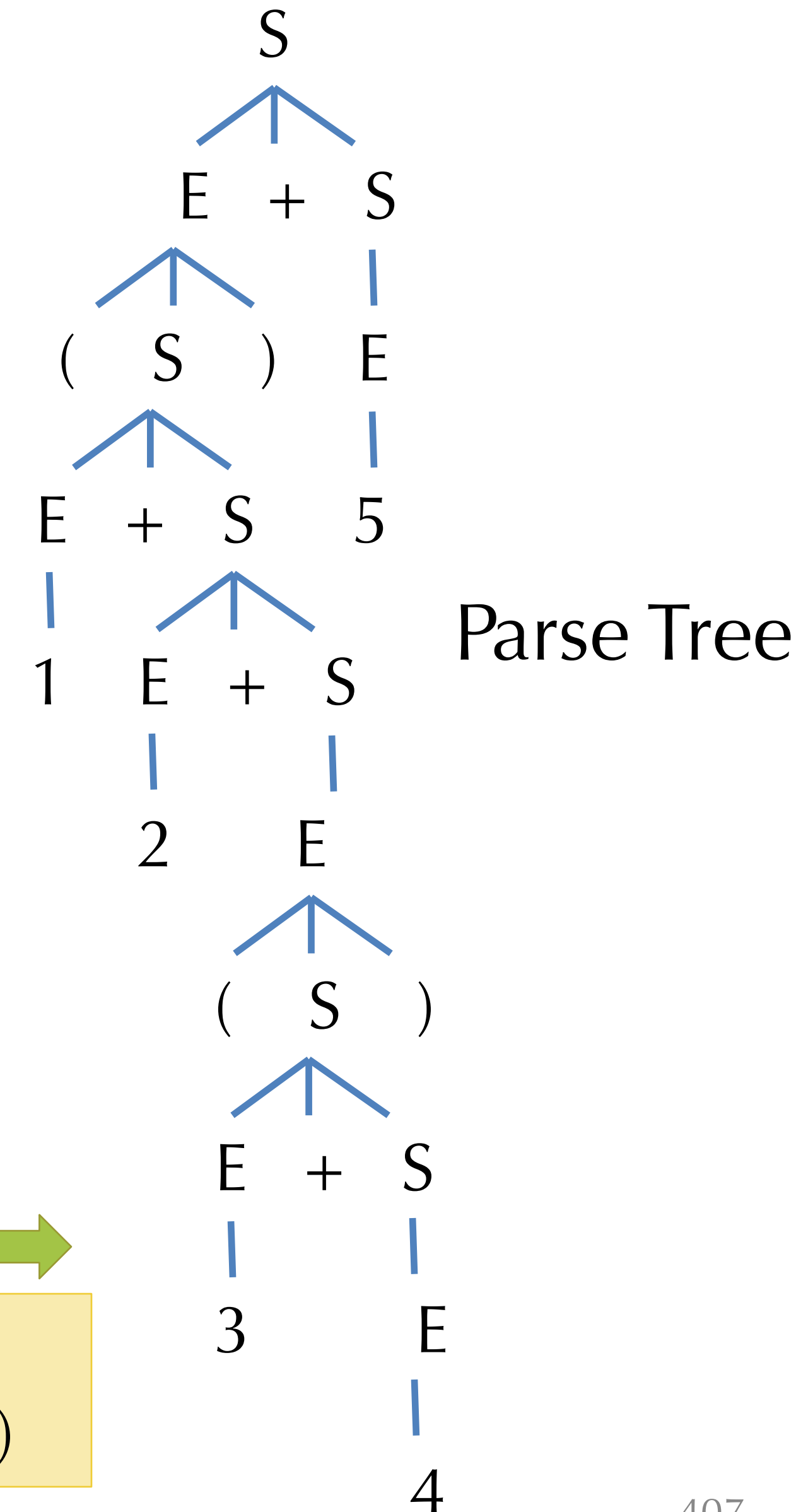
Note: Underline indicates symbol being expanded.



# From Derivations to Parse Trees

- Tree representation of the derivation
- Leaves of the tree are terminals
  - In-order traversal yields the input sequence of tokens
- Internal nodes: nonterminals
- No information about the *order* of the derivation steps
- $(1 + 2 + (3 + 4)) + 5$

$S \mapsto E + S \mid E$   
 $E \mapsto \text{number} \mid ( S )$



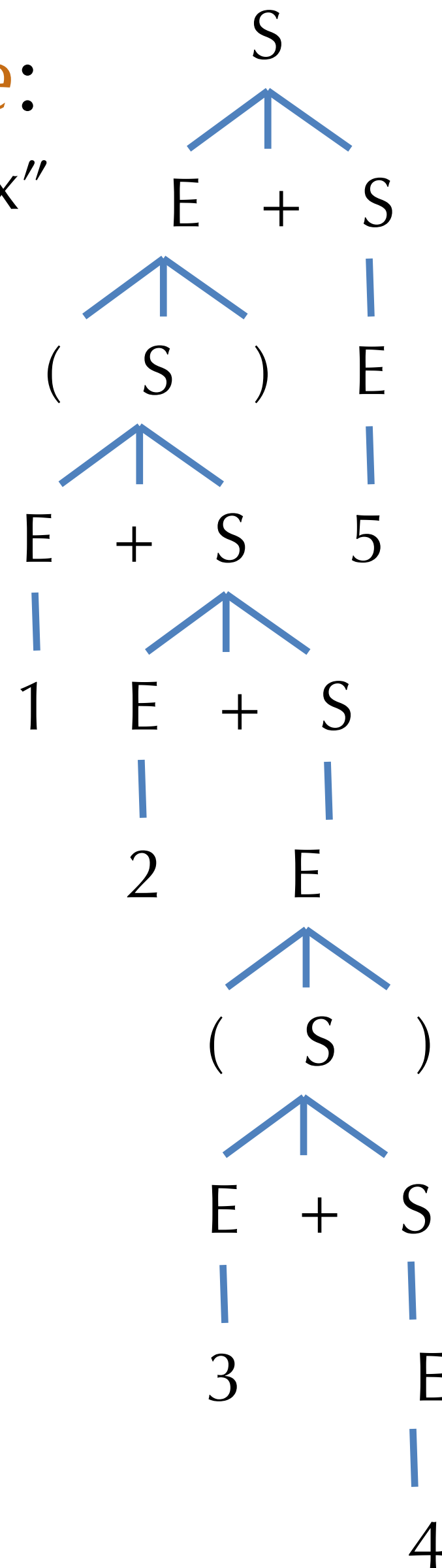




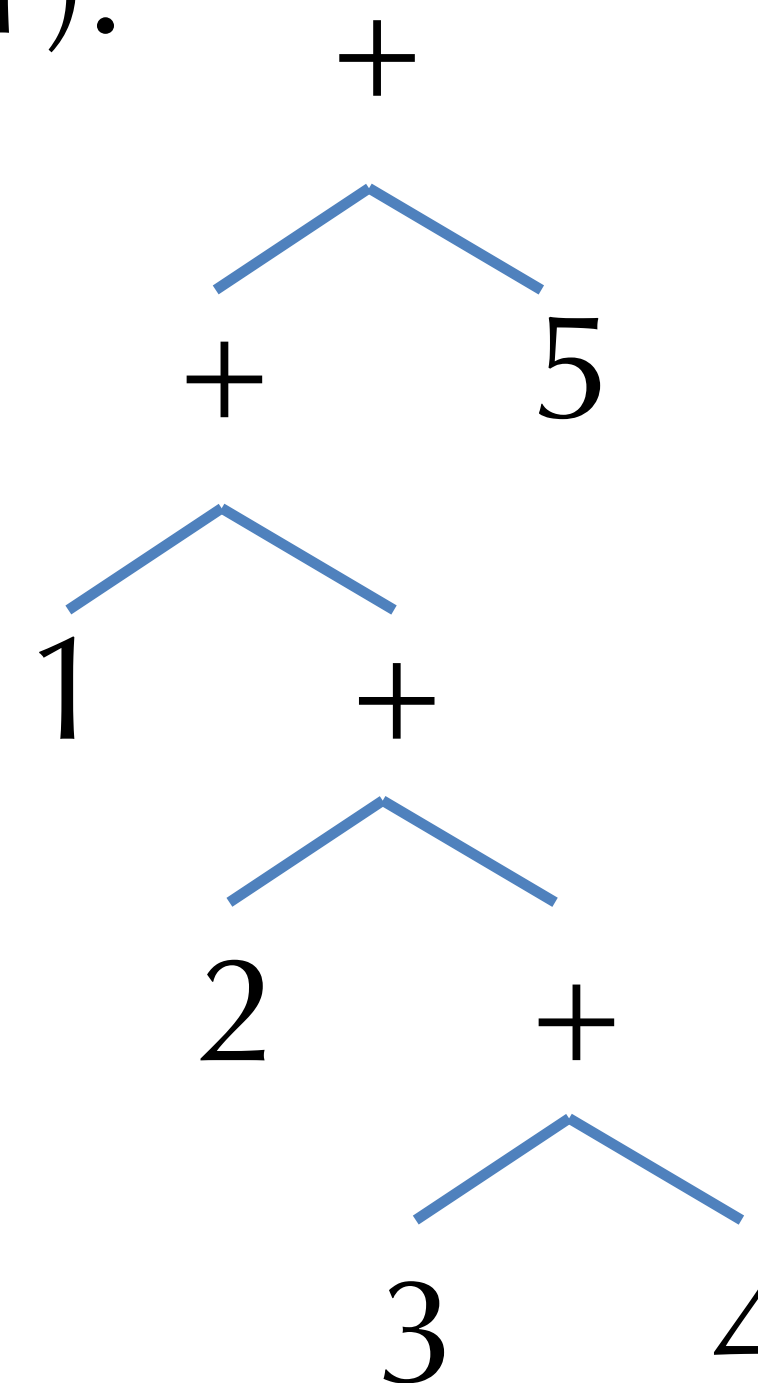
# From Parse Trees to Abstract Syntax

- *Parse tree*:  
“concrete syntax”

- Parse Trees  
enum S {  
  Plus(E, PlusSign, S),  
  Exp(E)  
}  
enum E {  
  Num(number),  
  Paren(LP, S, RP)  
}



- *Abstract syntax tree*  
(AST):



- AST  
enum Exp {  
  Plus(Exp, Exp),  
  Num(number)  
}

- Hides, or *abstracts*,  
unnneeded information.

$S \mapsto E + S \mid E$   
 $E \mapsto \text{number} \mid ( S )$

# Derivation Orders

- Productions of the grammar can be applied in any order.
- There are two standard orders:
  - *Leftmost derivation*: Find the left-most nonterminal and apply a production to it.
  - *Rightmost derivation*: Find the right-most nonterminal and apply a production there.
- Idea: These are **search strategies** for finding a parse tree
  - Both strategies (and any other) yield the same parse tree!
  - Parse tree doesn't contain the information about what order the productions were applied.
  - Just like an enum value doesn't tell you an order in which its subtrees were constructed.

# Example: Left- and rightmost derivations

- Leftmost derivation:

- $\underline{S} \mapsto \underline{E} + S$   
 $\mapsto (\underline{S}) + S$   
 $\mapsto (\underline{E} + S) + S$   
 $\mapsto (1 + \underline{S}) + S$   
 $\mapsto (1 + \underline{E} + S) + S$   
 $\mapsto (1 + 2 + \underline{S}) + S$   
 $\mapsto (1 + 2 + \underline{E}) + S$   
 $\mapsto (1 + 2 + (\underline{S})) + S$   
 $\mapsto (1 + 2 + (\underline{E} + S)) + S$   
 $\mapsto (1 + 2 + (3 + \underline{S})) + S$   
 $\mapsto (1 + 2 + (3 + \underline{E})) + S$   
 $\mapsto (1 + 2 + (3 + 4)) + \underline{S}$   
 $\mapsto (1 + 2 + (3 + 4)) + \underline{E}$   
 $\mapsto (1 + 2 + (3 + 4)) + 5$

- Rightmost derivation:

- $\underline{S} \mapsto E + \underline{S}$   
 $\mapsto E + \underline{E}$   
 $\mapsto \underline{E} + 5$   
 $\mapsto (\underline{S}) + 5$   
 $\mapsto (E + \underline{S}) + 5$   
 $\mapsto (E + E + \underline{S}) + 5$   
 $\mapsto (E + E + \underline{E}) + 5$   
 $\mapsto (E + E + (\underline{S})) + 5$   
 $\mapsto (E + E + (E + \underline{S})) + 5$   
 $\mapsto (E + E + (E + \underline{E})) + 5$   
 $\mapsto (E + E + (\underline{E} + 4)) + 5$   
 $\mapsto (E + \underline{E} + (3 + 4)) + 5$   
 $\mapsto (\underline{E} + 2 + (3 + 4)) + 5$   
 $\mapsto (1 + 2 + (3 + 4)) + 5$

$S \mapsto E + S \mid E$   
 $E \mapsto \text{number} \mid ( S )$



# Loops and Termination

- Some care is needed when defining CFGs
- Consider:

$$\begin{array}{l} S \mapsto E \\ E \mapsto S \end{array}$$

  - This grammar has nonterminal definitions that are “nonproductive”. (i.e. they don’t mention any terminal symbols)
  - There is no finite derivation starting from  $S$ , so the language is empty.
- Consider:

$$S \mapsto ( S )$$

  - This grammar is productive, but again there is no finite derivation starting from  $S$ , so the language is empty
- Easily generalize these examples to a “cycle” of many nonterminals, which can be harder to find in a large grammar
- Upshot: be aware of “vacuously empty” CFG grammars.
  - Every nonterminal should eventually rewrite to an alternative that contains only terminal symbols.

# Regular Expressions to CFGs

Theorem: every Regex can be expressed as a CFG, i.e. there is a CFG that generates exactly the strings in the Regex

- 'a'                 $S \rightarrow a$
- $\emptyset$              $S$  with no productions
- $\epsilon$                  $S \rightarrow \epsilon$
  
- $R_1 \mid R_2$          $S \rightarrow S_1$   
                          $S \rightarrow S_2$   
                         where  $S_1 \rightarrow \dots$  and  $S_2 \rightarrow \dots$  are CFGs for  $R_1, R_2$
  
- $R_1 R_2$              $S \rightarrow S_1 S_2$   
                         where  $S_1 \rightarrow \dots$  and  $S_2 \rightarrow \dots$  are CFGs for  $R_1, R_2$
  
- $R^*$                  $S \rightarrow \epsilon$   
                          $S \rightarrow S_R S$   
                         where  $S_R \rightarrow \dots$  is CFG for  $R$



Associativity, ambiguity, and precedence.

# GRAMMARS FOR PROGRAMMING LANGUAGES



# Associativity

Consider the input:  $1 + 2 + 3$

$S \mapsto E + S \mid E$   
 $E \mapsto \text{number} \mid ( S )$

Leftmost derivation:    Rightmost derivation:

$\underline{S} \mapsto \underline{E} + S$

$\mapsto 1 + \underline{S}$

$\mapsto 1 + \underline{E} + S$

$\mapsto 1 + 2 + \underline{S}$

$\mapsto 1 + 2 + \underline{E}$

$\mapsto 1 + 2 + 3$

$\underline{S} \mapsto E + \underline{S}$

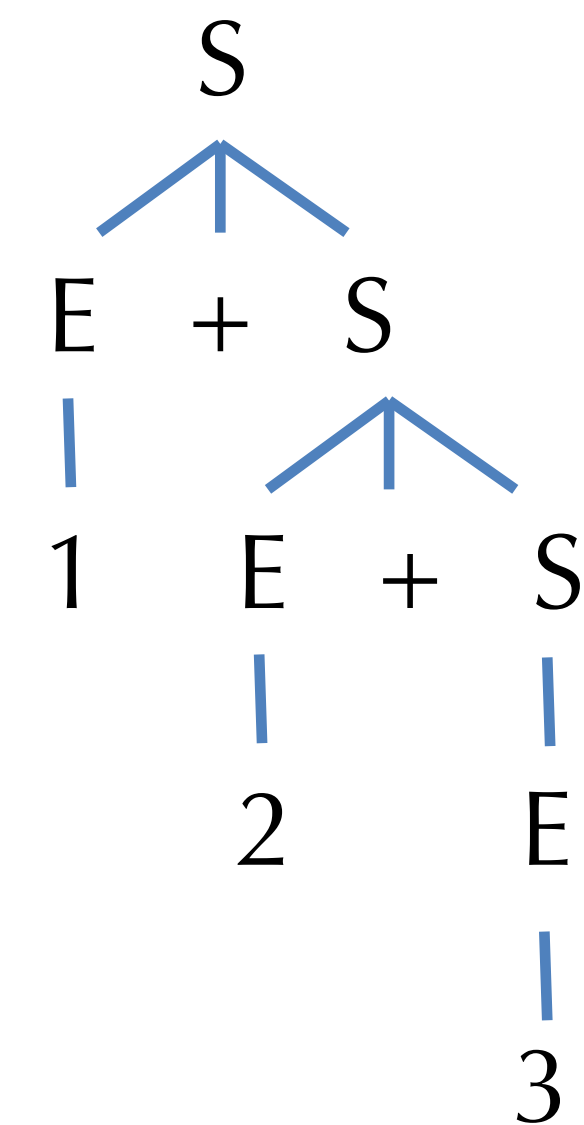
$\mapsto E + E + \underline{S}$

$\mapsto E + E + \underline{E}$

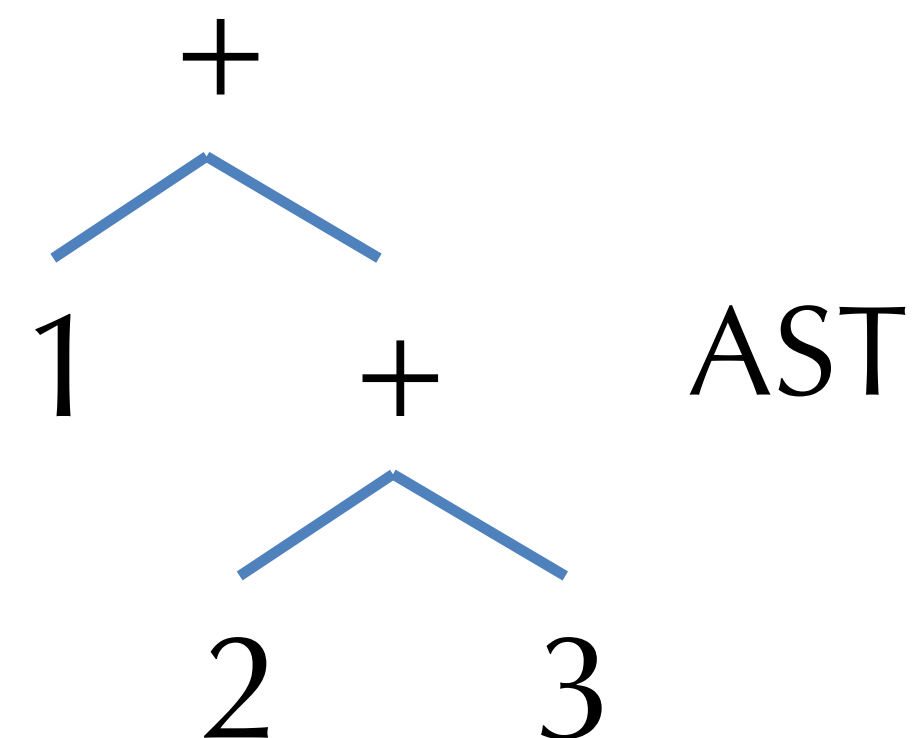
$\mapsto E + \underline{E} + 3$

$\mapsto \underline{E} + 2 + 3$

$\mapsto 1 + 2 + 3$



Parse Tree



AST

# Associativity

- This grammar makes '+' *right associative*...
  - i.e., the abstract syntax tree is the same for both  
1 + 2 + 3 and 1 + (2 + 3)
- Note that the grammar is *right recursive*...

$S \mapsto E + S \mid E$   
 $E \mapsto \text{number} \mid ( S )$

S refers to itself  
on the right of +

- How would you make '+' left associative?
- What are the trees for "1 + 2 + 3"?



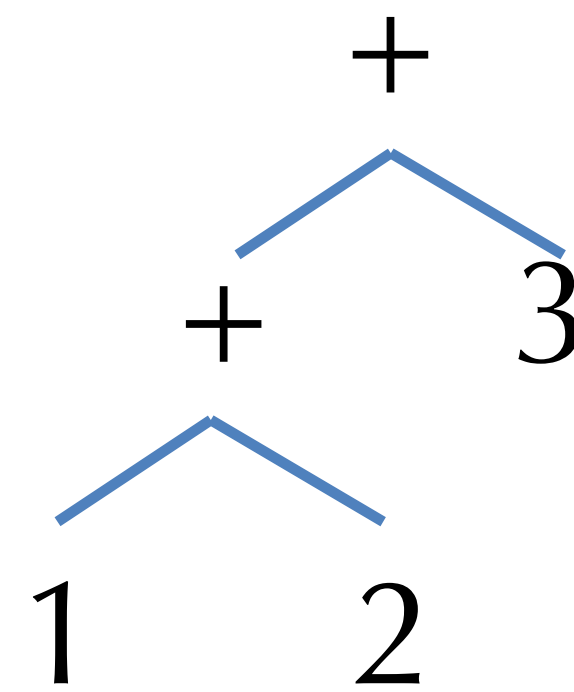
# Ambiguity

- Consider this grammar:

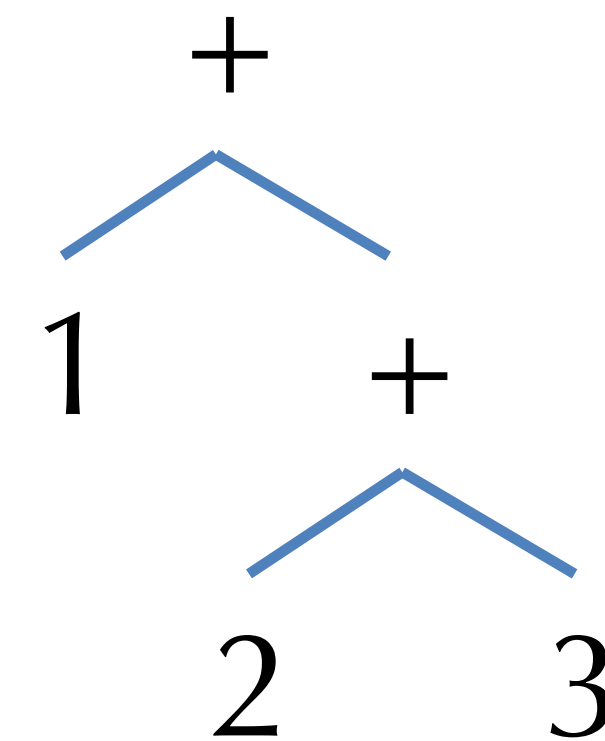
$$S \mapsto S + S \mid (S) \mid \text{number}$$

- Claim: it accepts the *same* set of strings as the previous one.
- What's the difference?
- Consider these *two* leftmost derivations:
  - $\underline{S} \mapsto \underline{S} + S \mapsto 1 + \underline{S} \mapsto 1 + \underline{S} + S \mapsto 1 + 2 + \underline{S} \mapsto 1 + 2 + 3$
  - $\underline{S} \mapsto \underline{S} + S \mapsto \underline{S} + S + S \mapsto 1 + \underline{S} + S \mapsto 1 + 2 + \underline{S} \mapsto 1 + 2 + 3$

- One derivation gives left associativity, the other gives right associativity to '+'
  - Which is which?



AST 1



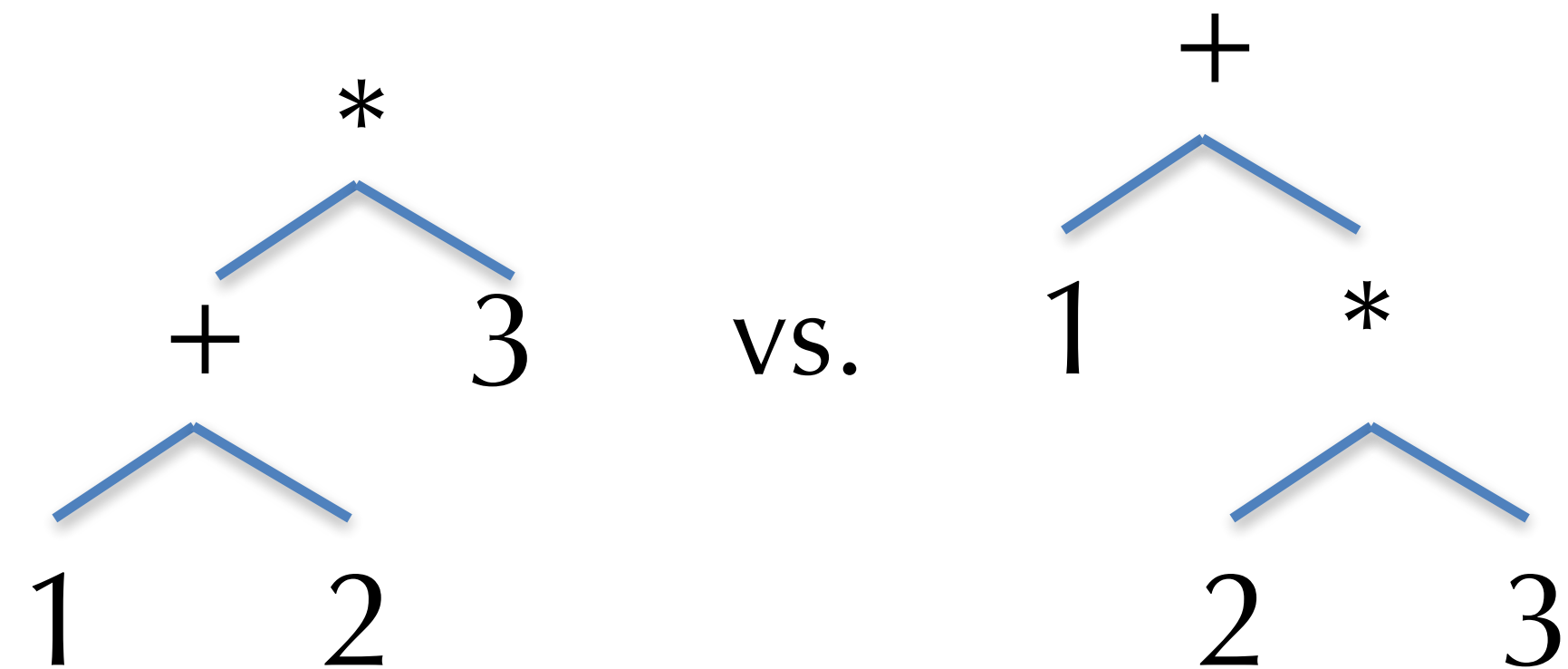
AST 2

# Why do we care about ambiguity?

- The '+' operation is associative, so it doesn't matter which tree we pick.  
Mathematically,  $x + (y + z) = (x + y) + z$ 
  - But, some operations aren't associative. Examples?
- Moreover, if there are multiple operations, ambiguity in the grammar leads to ambiguity in their *precedence*
- Consider:

$S \mapsto S + S \mid S * S \mid (S) \mid \text{number}$

- Input:  $1 + 2 * 3$ 
  - One parse =  $(1 + 2) * 3 = 9$
  - The other =  $1 + (2 * 3) = 7$



# Eliminating Ambiguity

- We can often eliminate ambiguity by adding nonterminals and allowing recursion only on the left (or right) .
- Higher-precedence operators go *farther* from the start symbol.
- Example:

$$S \mapsto S + S \mid S * S \mid ( S ) \mid \text{number}$$

- To disambiguate:
  - Decide (following math) to make '\*' higher precedence than '+'
  - Make '+' left associative
  - Make '\*' right associative
- Note:
  - $S_2$  corresponds to 'atomic' expressions

$$S_0 \mapsto S_0 + S_1 \mid S_1$$

$$S_1 \mapsto S_2 * S_1 \mid S_2$$

$$S_2 \mapsto \text{number} \mid ( S_0 )$$



# Context Free Grammars: Summary

- Context-free grammars allow concise specifications of programming languages.
  - An unambiguous CFG specifies how to parse: convert a token stream to a parse tree
    - Parse trees describe what productions were used to construct the string
    - Derivations describe a trace of what order the productions were used in
  - Ambiguity can (often) be removed by encoding precedence and associativity in the grammar.
- Even with an unambiguous CFG, there may be more than one **derivation**
  - Though all derivations correspond to the same parse tree.
- Next time: parsing CFGs



Searching for derivations.

# LL & LR PARSING

# CFGs Mathematically

- A Context-free Grammar (CFG) consists of
  - A set of *terminals* (e.g., a token or  $\epsilon$ )
  - A set of *nonterminals* (e.g.,  $S$  and other syntactic variables)
  - A designated nonterminal called the *start symbol*
  - A set of productions:  $\text{LHS} \mapsto \text{RHS}$ 
    - LHS is a nonterminal
    - RHS is a *string* of terminals and nonterminals
- Example: The balanced parentheses language:

$$S \mapsto (S)S$$

$$S \mapsto \epsilon$$



# Derivations in CFGs

- Example: derive  $(1 + 2 + (3 + 4)) + 5$

- $\underline{S} \mapsto \underline{E} + S$

$$\mapsto (\underline{S}) + S$$

$$\mapsto (\underline{E} + S) + S$$

$$\mapsto (1 + \underline{S}) + S$$

$$\mapsto (1 + \underline{E} + S) + S$$

$$\mapsto (1 + 2 + \underline{S}) + S$$

$$\mapsto (1 + 2 + \underline{E}) + S$$

$$\mapsto (1 + 2 + (\underline{S})) + S$$

$$\mapsto (1 + 2 + (\underline{E} + S)) + S$$

$$\mapsto (1 + 2 + (3 + \underline{S})) + S$$

$$\mapsto (1 + 2 + (3 + \underline{E})) + S$$

$$\mapsto (1 + 2 + (3 + 4)) + \underline{S}$$

$$\mapsto (1 + 2 + (3 + 4)) + \underline{E}$$

$$\mapsto (1 + 2 + (3 + 4)) + 5$$

$$S \mapsto E + S \mid E$$

$$E \mapsto \text{number} \mid ( S )$$

For arbitrary strings  $\alpha, \beta, \gamma$  and production rule  $A \mapsto \beta$  a single step of the derivation is:

$$\alpha A \gamma \mapsto \alpha \beta \gamma$$

( *substitute*  $\beta$  for an occurrence of  $A$  )

In general, there are many possible derivations for a given string

Note: Underline indicates symbol being expanded.

# Example: Left- and rightmost derivations

- Leftmost derivation:

- $\underline{S} \mapsto \underline{E} + S$   
 $\mapsto (\underline{S}) + S$   
 $\mapsto (\underline{E} + S) + S$   
 $\mapsto (1 + \underline{S}) + S$   
 $\mapsto (1 + \underline{E} + S) + S$   
 $\mapsto (1 + 2 + \underline{S}) + S$   
 $\mapsto (1 + 2 + \underline{E}) + S$   
 $\mapsto (1 + 2 + (\underline{S})) + S$   
 $\mapsto (1 + 2 + (\underline{E} + S)) + S$   
 $\mapsto (1 + 2 + (3 + \underline{S})) + S$   
 $\mapsto (1 + 2 + (3 + \underline{E})) + S$   
 $\mapsto (1 + 2 + (3 + 4)) + \underline{S}$   
 $\mapsto (1 + 2 + (3 + 4)) + \underline{E}$   
 $\mapsto (1 + 2 + (3 + 4)) + 5$

- Rightmost derivation:

- $\underline{S} \mapsto E + \underline{S}$   
 $\mapsto E + \underline{E}$   
 $\mapsto \underline{E} + 5$   
 $\mapsto (\underline{S}) + 5$   
 $\mapsto (E + \underline{S}) + 5$   
 $\mapsto (E + E + \underline{S}) + 5$   
 $\mapsto (E + E + \underline{E}) + 5$   
 $\mapsto (E + E + (\underline{S})) + 5$   
 $\mapsto (E + E + (E + \underline{S})) + 5$   
 $\mapsto (E + E + (E + \underline{E})) + 5$   
 $\mapsto (E + E + (\underline{E} + 4)) + 5$   
 $\mapsto (E + \underline{E} + (3 + 4)) + 5$   
 $\mapsto (\underline{E} + 2 + (3 + 4)) + 5$   
 $\mapsto (1 + 2 + (3 + 4)) + 5$

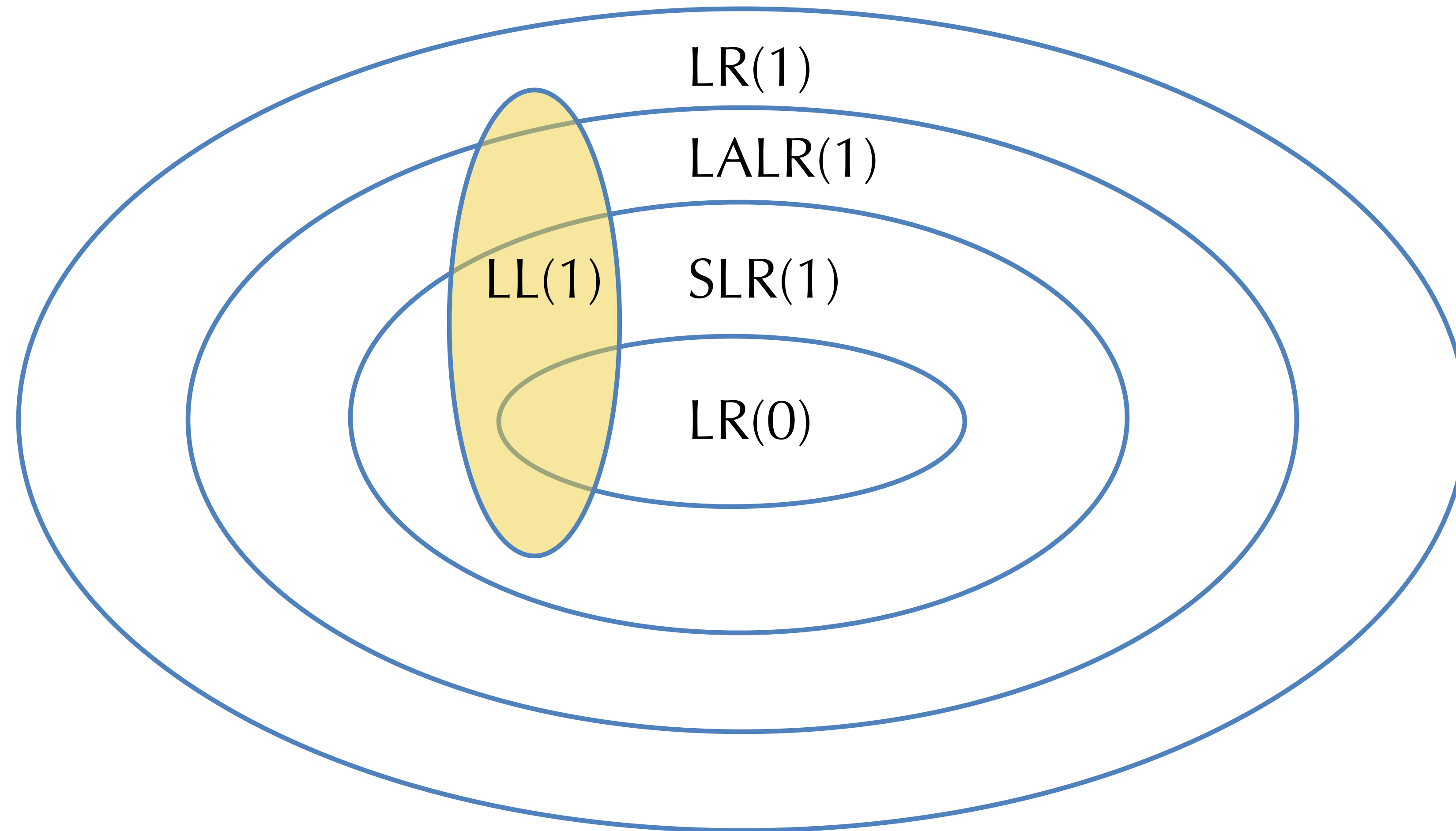
$S \mapsto E + S \mid E$   
 $E \mapsto \text{number} \mid ( S )$



# CFGs In Practice

- Context-free Grammars are elegant, *declarative* specifications, generalizing regular expressions
- A parser for a CFG amounts to a *search procedure* for derivations
- Unlike regular expressions, which are easily compiled to linear time recognizers, practical algorithms for parsing *general* CFGs are  $O(n^3)$  in input string length
  - Compromise: add restrictions to the CFGs
    - Benefit: Linear time
    - Drawback: have to rewrite the grammar to make it fit the restrictions

# Classification of Grammars



# LL(1) GRAMMARS



# Consider finding left-most derivations

- Look at only one input symbol at a time.

$$\begin{aligned} S &\mapsto E + S \mid E \\ E &\mapsto \text{number} \mid ( S ) \end{aligned}$$

| Partly-derived String                   | Look-ahead | <b>Parsed</b> /Unparsed Input |
|-----------------------------------------|------------|-------------------------------|
| <u>S</u>                                | (          | (1 + 2 + (3 + 4)) + 5         |
| $\mapsto$ <u>E</u> + S                  | (          | (1 + 2 + (3 + 4)) + 5         |
| $\mapsto$ ( <u>S</u> ) + S              | 1          | (1 + 2 + (3 + 4)) + 5         |
| $\mapsto$ ( <u>E</u> + S) + S           | 1          | (1 + 2 + (3 + 4)) + 5         |
| $\mapsto$ (1 + <u>S</u> ) + S           | 2          | (1 + 2 + (3 + 4)) + 5         |
| $\mapsto$ (1 + <u>E</u> + S) + S        | 2          | (1 + 2 + (3 + 4)) + 5         |
| $\mapsto$ (1 + 2 + <u>S</u> ) + S       | (          | (1 + 2 + (3 + 4)) + 5         |
| $\mapsto$ (1 + 2 + <u>E</u> ) + S       | (          | (1 + 2 + (3 + 4)) + 5         |
| $\mapsto$ (1 + 2 + ( <u>S</u> )) + S    | 3          | (1 + 2 + (3 + 4)) + 5         |
| $\mapsto$ (1 + 2 + ( <u>E</u> + S)) + S | 3          | (1 + 2 + (3 + 4)) + 5         |
| $\mapsto$ ...                           |            |                               |

# There is a problem

- We want to decide which production to apply based on the look-ahead symbol.
- But, there is a choice:

$$\begin{array}{l} S \mapsto E + S \mid E \\ E \mapsto \text{number} \mid ( S ) \end{array}$$

(1)  $S \mapsto E \mapsto (S) \mapsto (E) \mapsto (1)$

vs.

(1) + 2  $S \mapsto E + S \mapsto (S) + S \mapsto (E) + S \mapsto (1) + S \mapsto (1) + E \mapsto (1) + 2$

- Given the look-ahead symbol: '(' it isn't clear whether to pick  $S \mapsto E$  or  $S \mapsto E + S$  first.

# Grammar is the problem

- Not all grammars can be parsed “top-down” with only a single lookahead symbol.
- *Top-down*: starting from the start symbol (root of the parse tree) and going down
- LL(1) means
  - Left-to-right scanning
  - Left-most derivation,
  - 1 lookahead symbol
- This language isn’t “LL(1)”
- Is it LL(k) for some k?
- What can we do?

$$\begin{aligned} S &\mapsto E + S \mid E \\ E &\mapsto \text{number} \mid ( S ) \end{aligned}$$



# Making a grammar LL(1)

- *Problem:* We can't decide which S production to apply until we see the symbol after the first expression.
- *Solution:* "Left-factor" the grammar. There is a common S prefix for each choice, so add a new non-terminal  $S'$  at the decision point:

$$S \mapsto E + S \mid E$$
$$E \mapsto \text{number} \mid ( S )$$

$$S \mapsto ES'$$
$$S' \mapsto \varepsilon$$
$$S' \mapsto + S$$
$$E \mapsto \text{number} \mid ( S )$$

- Also need to eliminate left-recursion. Why?
- Consider:

$$S \mapsto S + E \mid E$$
$$E \mapsto \text{number} \mid ( S )$$

# LL(1) Parse of the input string

- Look at only one input symbol at a time.

$$S \mapsto ES'$$

$$S' \mapsto \epsilon$$

$$S' \mapsto + S$$

$$E \mapsto \text{number} \mid ( S )$$

| Partly-derived String                 | Look-ahead | Parsed/Unparsed Input |
|---------------------------------------|------------|-----------------------|
| <u>S</u>                              | (          | (1 + 2 + (3 + 4)) + 5 |
| $\mapsto$ <u>E</u> S'                 | (          | (1 + 2 + (3 + 4)) + 5 |
| $\mapsto$ ( <u>S</u> ) S'             | 1          | (1 + 2 + (3 + 4)) + 5 |
| $\mapsto$ ( <u>E</u> S') S'           | 1          | (1 + 2 + (3 + 4)) + 5 |
| $\mapsto$ (1 <u>S'</u> ) S'           | +          | (1 + 2 + (3 + 4)) + 5 |
| $\mapsto$ (1 + <u>S</u> ) S'          | 2          | (1 + 2 + (3 + 4)) + 5 |
| $\mapsto$ (1 + <u>E</u> S') S'        | 2          | (1 + 2 + (3 + 4)) + 5 |
| $\mapsto$ (1 + 2 <u>S'</u> ) S'       | +          | (1 + 2 + (3 + 4)) + 5 |
| $\mapsto$ (1 + 2 + <u>S</u> ) S'      | (          | (1 + 2 + (3 + 4)) + 5 |
| $\mapsto$ (1 + 2 + <u>E</u> S') S'    | (          | (1 + 2 + (3 + 4)) + 5 |
| $\mapsto$ (1 + 2 + ( <u>S</u> )S') S' | 3          | (1 + 2 + (3 + 4)) + 5 |



# Predictive Parsing

- Given an LL(1) grammar:
  - For a given nonterminal, the lookahead symbol uniquely determines the production to apply.
  - Top-down parsing = predictive parsing
  - Driven by a predictive parsing table:  
nonterminal \* input token  $\rightarrow$  production

$T \mapsto S\$$   
 $S \mapsto ES'$   
 $S' \mapsto \epsilon$   
 $S' \mapsto + S$   
 $E \mapsto \text{number} \mid ( S )$

|    | number               | +             | (               | )                  | \$ (EOF)           |
|----|----------------------|---------------|-----------------|--------------------|--------------------|
| T  | $\mapsto S\$$        |               | $\mapsto S\$$   |                    |                    |
| S  | $\mapsto E S'$       |               | $\mapsto E S'$  |                    |                    |
| S' |                      | $\mapsto + S$ |                 | $\mapsto \epsilon$ | $\mapsto \epsilon$ |
| E  | $\mapsto \text{num}$ |               | $\mapsto ( S )$ |                    |                    |

- Note: it is convenient to add a special *end-of-file* token \$ and a start symbol T (top-level) that requires \$.

# How do we construct the parse table?

- Consider a given production:  $A \rightarrow \gamma$
- Construct the set of all input tokens that may appear *first* in strings that can be derived from  $\gamma$ 
  - Add the production  $\rightarrow \gamma$  to the entry  $(A, \text{token})$  for each such token.
- If  $\gamma$  can derive  $\epsilon$  (the empty string), then we construct the set of all input tokens that may *follow* the nonterminal  $A$  in the grammar.
  - Add the production  $\rightarrow \gamma$  to the entry  $(A, \text{token})$  for each such token.
- Note: The grammar is LL(1) *if and only if* all entries have at most one production

# Example

- $\text{First}(T) = \text{First}(S)$
- $\text{First}(S) = \text{First}(E)$
- $\text{First}(S') = \{ + \}$
- $\text{First}(E) = \{ \text{number}, '(' \}$

- $\text{Follow}(S') = \text{Follow}(S)$
- $\text{Follow}(S) = \{ \$, ')' \} \cup \text{Follow}(S')$

$T \mapsto S\$$   
 $S \mapsto ES'$   
 $S' \mapsto \epsilon$   
 $S' \mapsto + S$   
 $E \mapsto \text{number} \mid ( S )$

**Note:** we want the *least* solution to this system of set equations... a *fixpoint* computation. Just like in program analysis!

|    | number                | +             | (               | )                  | \$ (EOF)           |
|----|-----------------------|---------------|-----------------|--------------------|--------------------|
| T  | $\mapsto S\$$         |               | $\mapsto S\$$   |                    |                    |
| S  | $\mapsto E S'$        |               | $\mapsto E S'$  |                    |                    |
| S' |                       | $\mapsto + S$ |                 | $\mapsto \epsilon$ | $\mapsto \epsilon$ |
| E  | $\mapsto \text{num.}$ |               | $\mapsto ( S )$ |                    |                    |



# Converting the table to code

- Define  $n$  mutually recursive functions
  - one for each nonterminal  $A$ : `parse_A`
  - The type of `parse_A` is `() -> Result<AST, Error>` if  $A$  is *not* an auxiliary nonterminal
  - Parse functions for auxiliary nonterminals (e.g.,  $S'$ ) take extra ast's as inputs, one for each nonterminal in the “factored” prefix.
- Each function “peeks” at the lookahead token and then follows the production rule in the corresponding entry.
  - Consume terminal tokens from the input stream
  - Call `parse_X` to create sub-tree for nonterminal  $X$
  - If the rule ends in an auxiliary nonterminal, call it with appropriate ast's. (The auxiliary rule is responsible for creating the ast after looking at more input.)
  - Otherwise, this function builds the ast tree itself and returns it.



|    | number                | +             | (               | )                  | \$ (EOF)           |
|----|-----------------------|---------------|-----------------|--------------------|--------------------|
| T  | $\mapsto S\$$         |               | $\mapsto S\$$   |                    |                    |
| S  | $\mapsto E S'$        |               | $\mapsto E S'$  |                    |                    |
| S' |                       | $\mapsto + S$ |                 | $\mapsto \epsilon$ | $\mapsto \epsilon$ |
| E  | $\mapsto \text{num.}$ |               | $\mapsto ( S )$ |                    |                    |

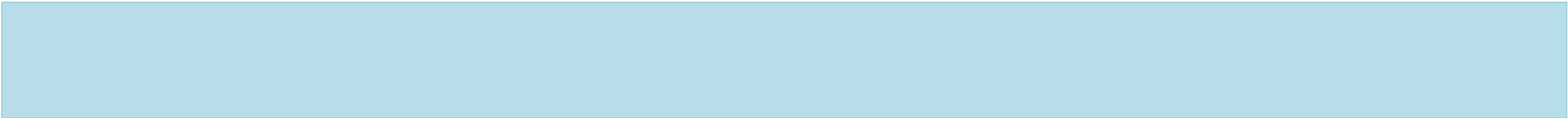
Hand-generated LL(1) code for the table above.

# DEMO: HANDPARSER.RS

# LL(1) Summary

- Top-down parsing that finds the leftmost derivation.
- Language Grammar
  - ⇒ LL(1) grammar (manual rewrite)
  - ⇒ prediction table (intermediate representation)
  - ⇒ recursive-descent parser (code generation)
- Problems:
  - Grammar must be LL(1)
  - Can extend to LL(k) (it just makes the table bigger)
  - Grammar cannot be left recursive (parser functions will loop!)
- Is there a better way?





# LR GRAMMARS

# The Parsing Problem

- The Parsing Problem:
  - Input: a context-free grammar  $G$
  - Output: a **parser** that takes in a string and outputs a parse tree of that string in  $G$  or raises an exception if there is no parse tree.
  - Notice that an ambiguous grammar may be parsed in multiple ways
- In practice: fuse the generation of the parse tree with *semantic actions* that construct the abstract syntax tree
  - The parse tree is usually never “materialized” in memory
- Another “mini-compiler” for a DSL
- Bad news: best algorithms are  $O(n^3)$ 
  - CYK, Earley, GLR algorithms
- Compromise: find restrictions on CFGs that allow for  $O(n)$  parsing
  - Intuition: parsing is a **search problem**, find restrictions that limit the amount of backtracking needed.
  - Cost: more burden on the programmer (i.e., **you**) to adapt their grammar to fit the restriction



# LL(1) Summary

- Top-down parsing that finds the leftmost derivation.
- Language Grammar
  - ⇒ LL(1) grammar (manual rewrite)
  - ⇒ prediction table (intermediate representation)
  - ⇒ recursive-descent parser (code generation)
- Problems:
  - Grammar must be LL(1)
  - Can extend to LL(k) (it just makes the table bigger)
  - Grammar cannot be left recursive (parser functions will loop!)
- Is there a better way?

# Bottom-up Parsing (LR Parsers)

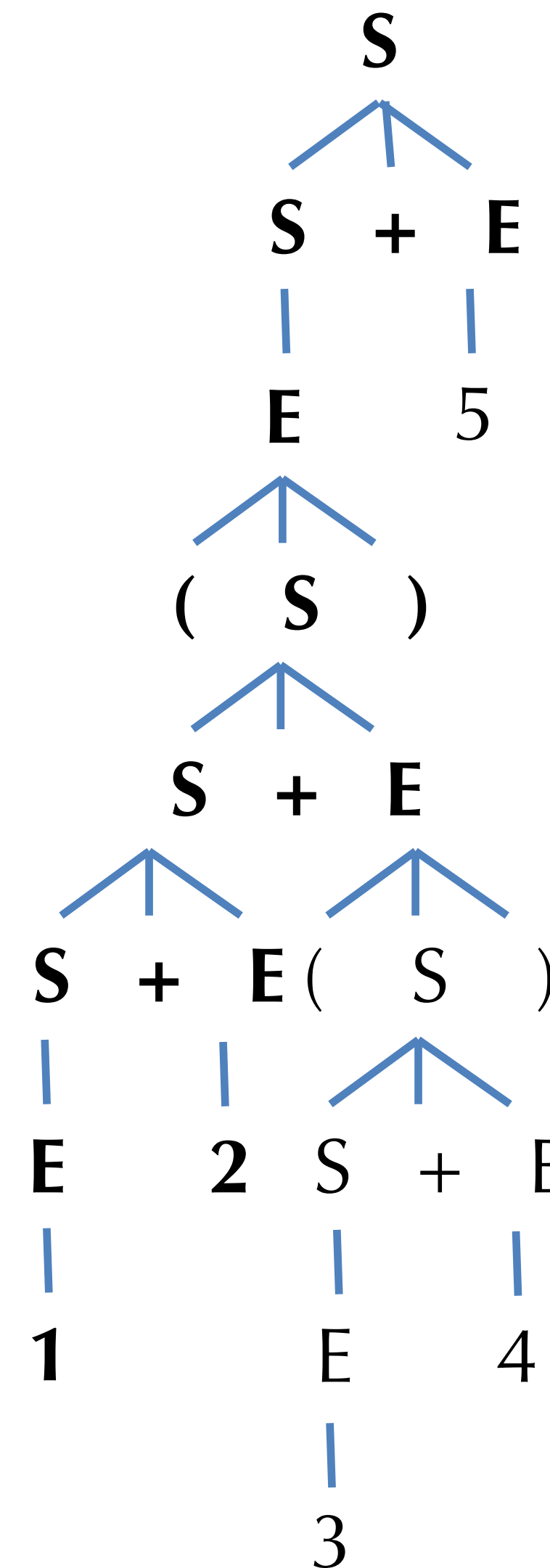
- LR(k) parser:
  - Left-to-right scanning
  - Rightmost derivation
  - k lookahead symbols
- LR grammars are more expressive than LL
  - Can handle left-recursive (and right recursive) grammars; virtually all programming languages
  - Easier to express programming language syntax (no left factoring)
- Technique: “Shift-Reduce” parsers
  - Work bottom up instead of top down
  - Construct right-most derivation of a program in the grammar
  - Used by many parser generators (e.g. yacc, ocamlyacc, lalrpop, etc.)
  - Better error detection/recovery

# Top-down vs. Bottom up

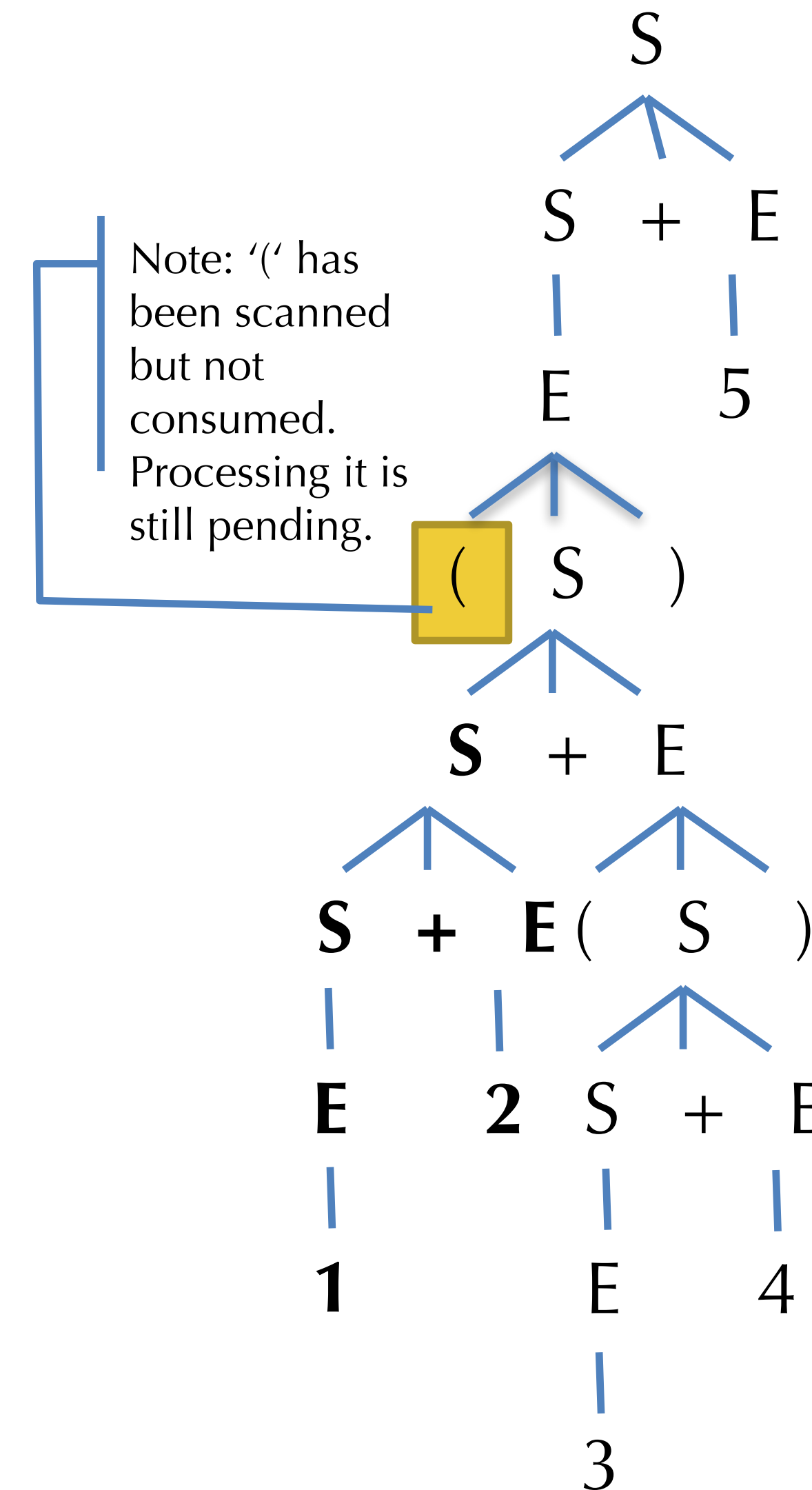
- Consider the left-recursive grammar:

$S \mapsto S + E \mid E$   
 $E \mapsto \text{number} \mid ( S )$

- $(1 + 2 + (3 + 4)) + 5$
- What part of the tree must we know after scanning just  $"(1 + 2"$  ?
- In top-down, must be able to guess which productions to use...



Top-down



Bottom-up



# Progress of Bottom-up Parsing

|                        | Reductions                                                            | Scanned                 | Input Remaining         |
|------------------------|-----------------------------------------------------------------------|-------------------------|-------------------------|
| Rightmost derivation ↑ | $(1 + 2 + (3 + 4)) + 5 \leftarrow$                                    |                         | $(1 + 2 + (3 + 4)) + 5$ |
|                        | $(\underline{\mathbf{E}} + 2 + (3 + 4)) + 5 \leftarrow$               | $($                     | $1 + 2 + (3 + 4)) + 5$  |
|                        | $(\underline{\mathbf{S}} + 2 + (3 + 4)) + 5 \leftarrow$               | $(1$                    | $+ 2 + (3 + 4)) + 5$    |
|                        | $(\mathbf{S} + \underline{\mathbf{E}} + (3 + 4)) + 5 \leftarrow$      | $(1 + 2$                | $+ (3 + 4)) + 5$        |
|                        | $(\underline{\mathbf{S}} + (3 + 4)) + 5 \leftarrow$                   | $(1 + 2$                | $+ (3 + 4)) + 5$        |
|                        | $(\mathbf{S} + (\underline{\mathbf{E}} + 4)) + 5 \leftarrow$          | $(1 + 2 + (3$           | $+ 4)) + 5$             |
|                        | $(\mathbf{S} + (\underline{\mathbf{S}} + 4)) + 5 \leftarrow$          | $(1 + 2 + (3$           | $+ 4)) + 5$             |
|                        | $(\mathbf{S} + (\mathbf{S} + \underline{\mathbf{E}})) + 5 \leftarrow$ | $(1 + 2 + (3 + 4$       | $)) + 5$                |
|                        | $(\mathbf{S} + (\underline{\mathbf{S}})) + 5 \leftarrow$              | $(1 + 2 + (3 + 4$       | $)) + 5$                |
|                        | $(\mathbf{S} + \underline{\mathbf{E}}) + 5 \leftarrow$                | $(1 + 2 + (3 + 4)$      | $) + 5$                 |
|                        | $(\underline{\mathbf{S}}) + 5 \leftarrow$                             | $(1 + 2 + (3 + 4)$      | $) + 5$                 |
|                        | $\underline{\mathbf{E}} + 5 \leftarrow$                               | $(1 + 2 + (3 + 4))$     | $+ 5$                   |
|                        | $\underline{\mathbf{S}} + 5 \leftarrow$                               | $(1 + 2 + (3 + 4))$     | $+ 5$                   |
|                        | $\mathbf{S} + \underline{\mathbf{E}} \leftarrow$                      | $(1 + 2 + (3 + 4)) + 5$ |                         |
|                        | $\mathbf{S}$                                                          |                         |                         |

$S \mapsto S + E \mid E$   
 $E \mapsto \text{number} \mid ( S )$

# Shift/Reduce Parsing

- Parser state:
  - Stack of terminals and nonterminals.
  - Unconsumed input is a string of terminals
  - Current derivation step is      stack + input
- Parsing is a sequence of *shift* and *reduce* operations:
- **Shift**: move look-ahead token to the stack
- **Reduce**: Replace symbols  $\gamma$  at top of stack with nonterminal  $X$  such that  $X \mapsto \gamma$  is a production. (pop  $\gamma$ , push  $X$ )

$$S \mapsto S + E \mid E$$

$$E \mapsto \text{number} \mid ( S )$$

| Stack  | Input                     | Action                            |
|--------|---------------------------|-----------------------------------|
|        | ( 1 + 2 + ( 3 + 4 ) ) + 5 | shift (                           |
| (      | 1 + 2 + ( 3 + 4 ) ) + 5   | shift 1                           |
| (1     | + 2 + ( 3 + 4 ) ) + 5     | reduce: $E \mapsto \text{number}$ |
| (E     | + 2 + ( 3 + 4 ) ) + 5     | reduce: $S \mapsto E$             |
| (S     | + 2 + ( 3 + 4 ) ) + 5     | shift +                           |
| (S +   | 2 + ( 3 + 4 ) ) + 5       | shift 2                           |
| (S + 2 | + ( 3 + 4 ) ) + 5         | reduce: $E \mapsto \text{number}$ |
| (S + E | + ( 3 + 4 ) ) + 5         | reduce: $S \mapsto S + E$         |
| (S     | + ( 3 + 4 ) ) + 5         | shift +                           |

# Shift/Reduce Parsing

- Parser state:
  - Stack of terminals and nonterminals.
  - Unconsumed input is a string of terminals
  - Current derivation step is      stack + input
- Invariant: Stack plus input is a step in building the Rightmost derivation in reverse

$$S \mapsto S + E \mid E$$

$$E \mapsto \text{number} \mid ( S )$$

| Stack  | Input                     | Derivation steps                 |  |
|--------|---------------------------|----------------------------------|--|
|        | ( 1 + 2 + ( 3 + 4 ) ) + 5 | ( 1 + 2 + ( 3 + 4 ) ) + 5        |  |
| (      | 1 + 2 + ( 3 + 4 ) ) + 5   |                                  |  |
| (1     | + 2 + ( 3 + 4 ) ) + 5     |                                  |  |
| (E     | + 2 + ( 3 + 4 ) ) + 5     | ( <u>E</u> + 2 + ( 3 + 4 ) ) + 5 |  |
| (S     | + 2 + ( 3 + 4 ) ) + 5     | ( <u>S</u> + 2 + ( 3 + 4 ) ) + 5 |  |
| (S +   | 2 + ( 3 + 4 ) ) + 5       |                                  |  |
| (S + 2 | + ( 3 + 4 ) ) + 5         |                                  |  |
| (S + E | + ( 3 + 4 ) ) + 5         | (S + <u>E</u> + ( 3 + 4 ) ) + 5  |  |
| (S     | + ( 3 + 4 ) ) + 5         | ( <u>S</u> + ( 3 + 4 ) ) + 5     |  |

Rightmost  
derivation



Simple LR parsing with no look ahead.

# LR(0) GRAMMARS

# LR Parser States

- Goal: know what set of reductions are legal at any given point.
- Idea: Summarize all possible stack prefixes  $\alpha$  as a finite parser state.
  - Parser state is computed by a DFA that reads the stack  $\sigma$ .
  - Accept states of the DFA correspond to unique reductions that apply.
- Example: LR(0) parsing
  - Left-to-right scanning, Right-most derivation, zero look-ahead tokens
  - Too weak to handle many language grammars (e.g. the “sum” grammar)
  - But, helpful for understanding how the shift-reduce parser works.



# Example LR(0) Grammar: Tuples

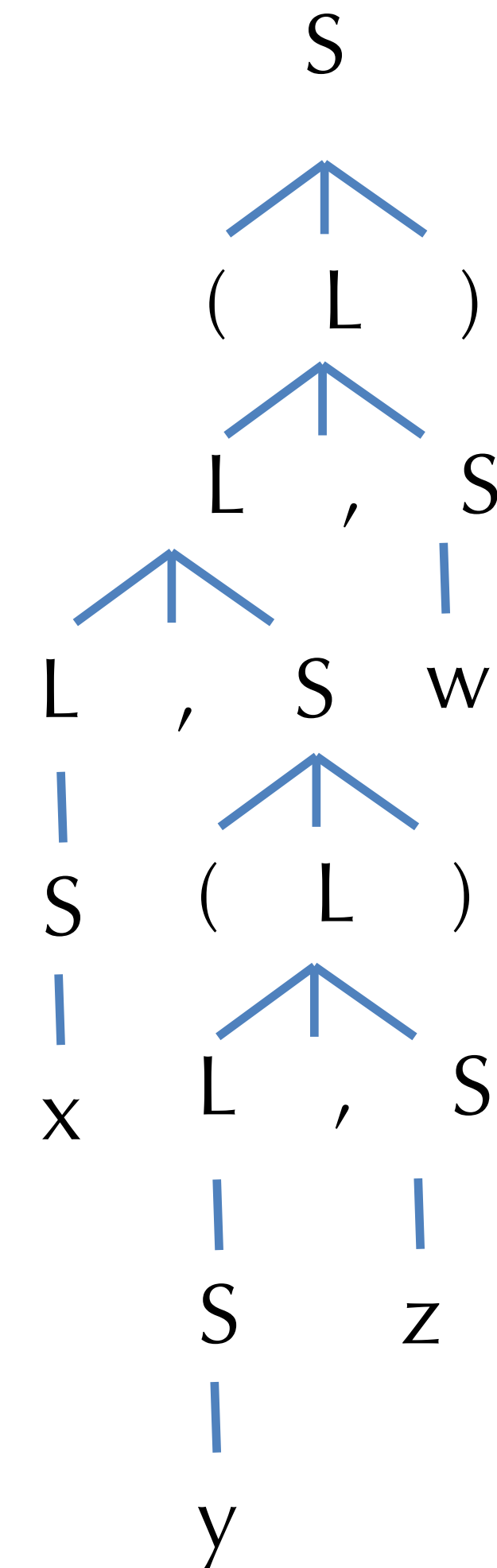
- Example grammar for non-empty tuples and identifiers:

$$\begin{aligned} S &\mapsto ( L ) \mid \text{id} \\ L &\mapsto S \mid L , S \end{aligned}$$

- Example strings:

- x
- (x,y)
- (((x)))
- (x, (y, z), w)
- (x, (y, (z, w)))

Parse tree for:  
(x, (y, z), w)



# Shift/Reduce Parsing

- Parser state:
  - Stack of terminals and nonterminals.
  - Unconsumed input is a string of terminals
  - Current derivation step is      stack + input
- Parsing is a sequence of *shift* and *reduce* operations:
- **Shift**: move look-ahead token to the stack: e.g.

$$S \mapsto ( L ) \mid id$$

$$L \mapsto S \mid L , S$$

| Stack | Input          | Action  |
|-------|----------------|---------|
|       | (x, (y, z), w) | shift ( |
| (     | x, (y, z), w)  | shift x |

- **Reduce**: Replace symbols  $\gamma$  at top of stack with nonterminal  $X$  such that  $X \mapsto \gamma$  is a production. (pop  $\gamma$ , push  $X$ ): e.g.

| Stack | Input        | Action                |
|-------|--------------|-----------------------|
| (x    | , (y, z), w) | reduce $S \mapsto id$ |
| (S    | , (y, z), w) | reduce $L \mapsto S$  |

# Example Run

| Stack     | Input          | Action                       |
|-----------|----------------|------------------------------|
|           | (x, (y, z), w) | shift (                      |
| (         | x, (y, z), w)  | shift x                      |
| (x        | , (y, z), w)   | reduce $S \mapsto \text{id}$ |
| (S        | , (y, z), w)   | reduce $L \mapsto S$         |
| (L        | , (y, z), w)   | shift ,                      |
| (L,       | (y, z), w)     | shift (                      |
| (L, (     | y, z), w)      | shift y                      |
| (L, (y    | , z), w)       | reduce $S \mapsto \text{id}$ |
| (L, (S    | , z), w)       | reduce $L \mapsto S$         |
| (L, (L    | , z), w)       | shift ,                      |
| (L, (L,   | z), w)         | shift z                      |
| (L, (L, z | ), w)          | reduce $S \mapsto \text{id}$ |
| (L, (L, S | ), w)          | reduce $L \mapsto L, S$      |
| (L, (L    | ), w)          | shift )                      |
| (L, (L)   | , w)           | reduce $S \mapsto ( L )$     |
| (L, S     | , w)           | reduce $L \mapsto L, S$      |
| (L        | , w)           | shift ,                      |
| (L,       | w)             | shift w                      |
| (L, w     | )              | reduce $S \mapsto \text{id}$ |
| (L, S     | )              | reduce $L \mapsto L, S$      |
| (L        | )              | shift )                      |
| (L)       |                | reduce $S \mapsto ( L )$     |
| S         |                |                              |

$S \mapsto ( L ) \mid \text{id}$   
 $L \mapsto S \mid L, S$



# Action Selection Problem

- Given a stack  $\sigma$  and a look-ahead symbol  $b$ , should the parser:
  - Shift  $b$  onto the stack (new stack is  $\sigma b$ )
  - Reduce a production  $X \mapsto \gamma$ , where  $\sigma = \alpha\gamma$  resulting in new stack is  $\alpha X$ )?
- Sometimes the parser can reduce but shouldn't
  - For example,  $X \mapsto \varepsilon$  can *always* be reduced
- Sometimes the stack can be reduced in different ways
- Main idea: decide what to do based on a *prefix*  $\alpha$  of the stack plus the look-ahead symbol.
  - The prefix  $\alpha$  is different for different possible reductions since in productions  $X \mapsto \gamma$  and  $Y \mapsto \beta$ ,  $\gamma$  and  $\beta$  might have different lengths.
- Main goal: know what set of reductions are legal at any point.
  - How do we keep track?

# LR(0) States

- An LR(0) *state* is a set of *items* keeping track of progress on possible upcoming reductions.
- An LR(0) *item* is a production from the language with an extra separator “.” somewhere in the right-hand-side

$$\begin{array}{l} S \mapsto ( L ) \mid id \\ L \mapsto S \mid L , S \end{array}$$

- Example items:  $S \mapsto .( L )$  or  $S \mapsto (. L)$  or  $L \mapsto S.$
- Intuition:
  - Stuff before the ‘.’ is already on the stack (beginnings of possible  $\gamma$ ’s to be reduced)
  - Stuff after the ‘.’ is what might be seen next
  - The prefixes  $\alpha$  are represented by the state itself

# Constructing the DFA: Start state & Closure

- First step: Add a new production  $S' \mapsto S\$$  to the grammar
- Start state of the DFA = empty stack, so it contains the item:  
 $S' \mapsto .S\$$
- Closure of a state:
  - Adds items for all productions whose LHS nonterminal occurs in an item in the state just after the  $'.'$
  - The added items have the  $'.'$  located at the beginning (no symbols for those items have been added to the stack yet)
  - Note that newly added items may cause yet more items to be added to the state... keep iterating until a *fixed point* is reached.
- Example:  $\text{CLOSURE}(\{S' \mapsto .S\$\}) = \{S' \mapsto .S\$, S \mapsto .(L), S \mapsto .id\}$
- Resulting “closed state” contains the set of all possible productions that might be reduced next.

$$\begin{array}{l} S' \mapsto S\$ \\ S \mapsto ( L ) \mid id \\ L \mapsto S \mid L , S \end{array}$$



# Example: Constructing the DFA

$S' \mapsto .S\$$

$S' \mapsto S\$$   
 $S \mapsto ( L ) \mid \text{id}$   
 $L \mapsto S \mid L , S$

- First, we construct a state with the initial item  $S' \mapsto .S\$$

# Example: Constructing the DFA

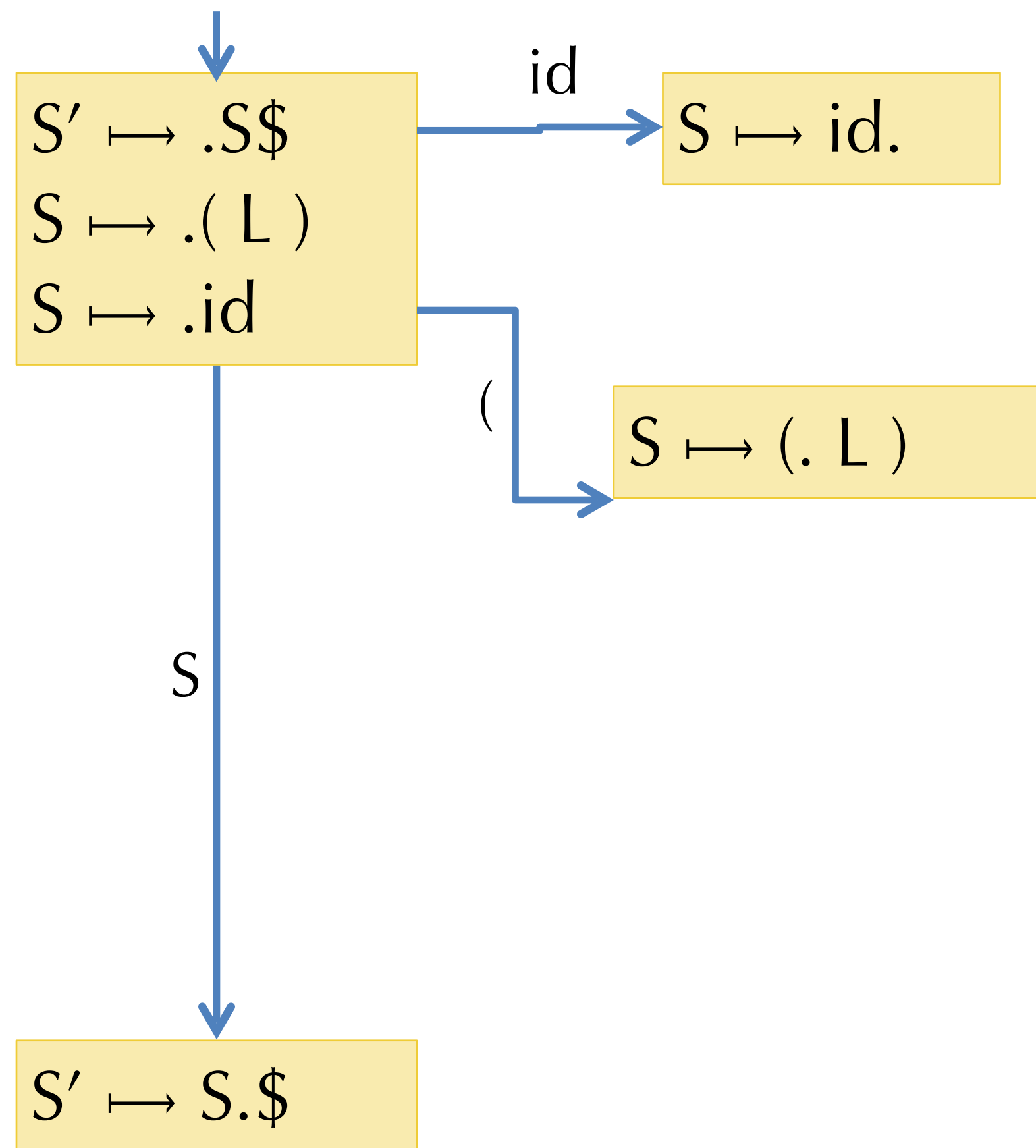
↓  
 $S' \mapsto .S\$$   
 $S \mapsto .( L )$   
 $S \mapsto .id$

$S' \mapsto S\$$   
 $S \mapsto ( L ) \mid id$   
 $L \mapsto S \mid L , S$

- Next, we take the closure of that state:  
 $\text{CLOSURE}(\{S' \mapsto .S\}) = \{S' \mapsto .S, S \mapsto .( L ), S \mapsto .id\}$
- In the set of items, the nonterminal S appears after the '.'
- So we add items for each S production in the grammar



# Example: Constructing the DFA



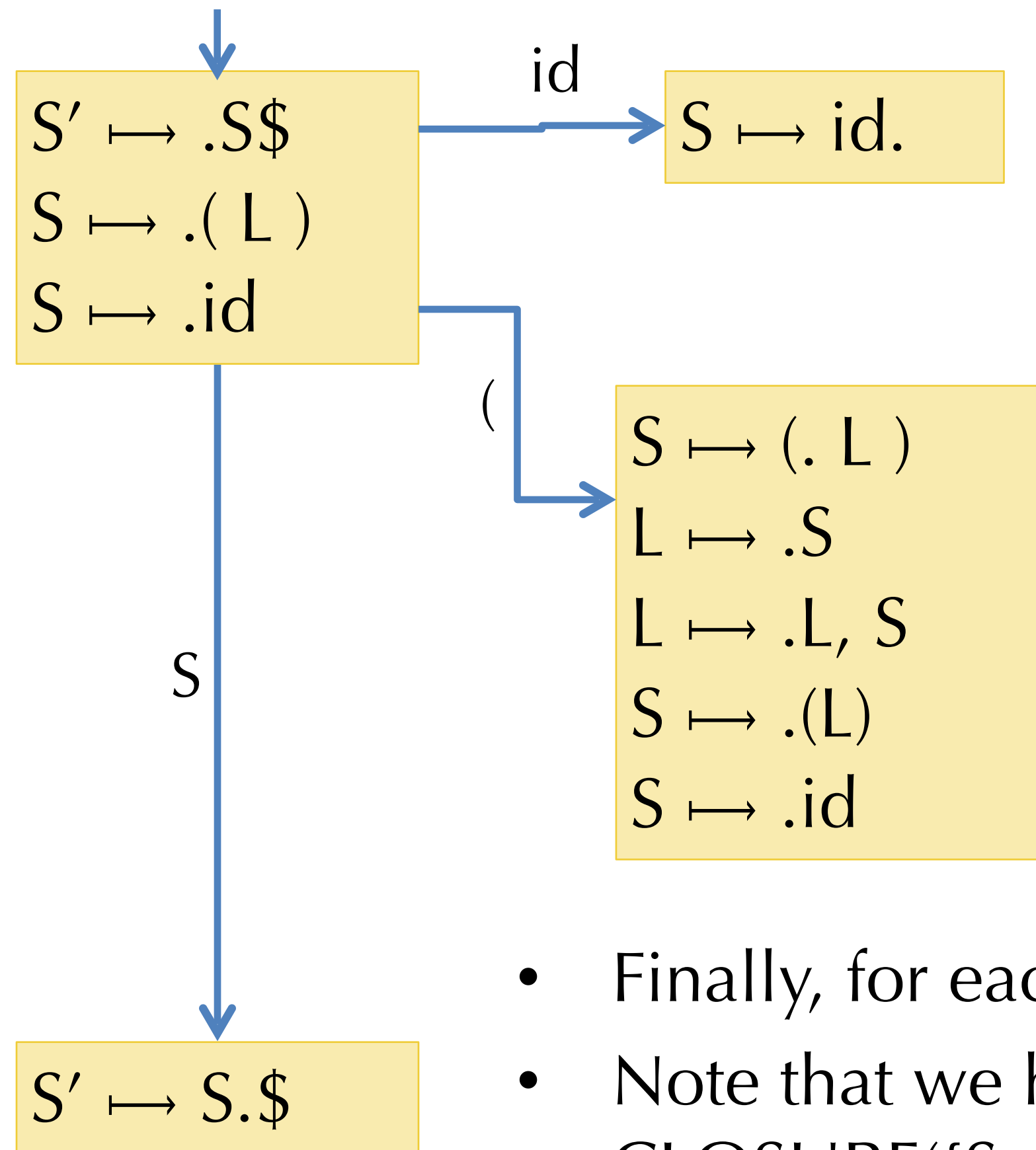
$S' \mapsto S\$$

$S \mapsto (L) \mid id$

$L \mapsto S \mid L, S$

- Next we add the transitions:
- First, we see what terminals and nonterminals can appear after the  $'.'$  in the source state.
  - Outgoing edges have those label.
- The target state (initially) includes all items from the source state that have the edge-label symbol after the  $'.'$ , but we advance the  $'.'$  (to simulate shifting the item onto the stack)

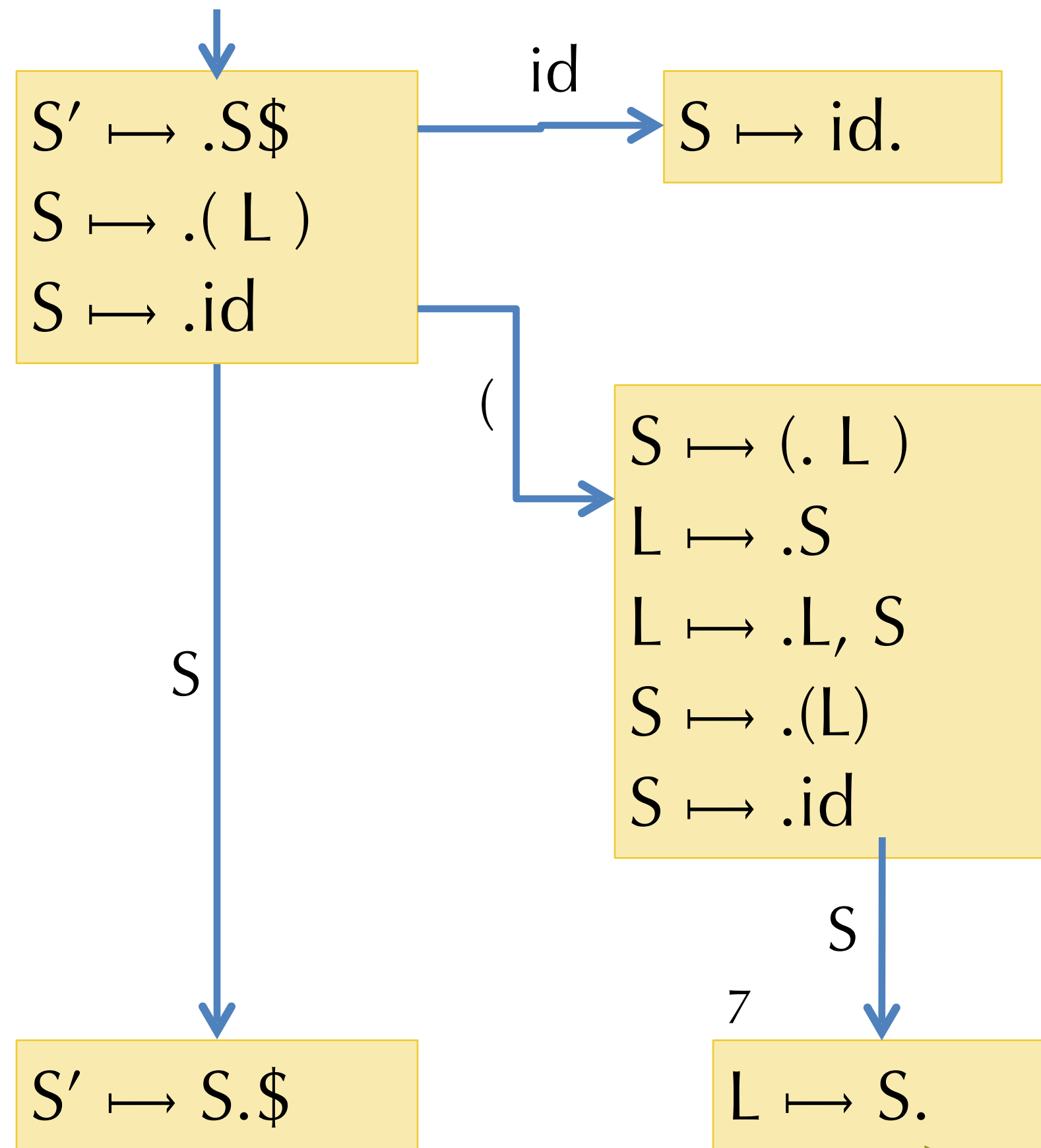
# Example: Constructing the DFA



$S' \mapsto S\$$   
 $S \mapsto (L) \mid id$   
 $L \mapsto S \mid L, S$

- Finally, for each new state, we take the closure.
- Note that we have to perform two iterations to compute  $CLOSURE(\{S \mapsto (.L)\})$ 
  - First iteration adds  $L \mapsto .S$  and  $L \mapsto .L, S$
  - Second iteration adds  $S \mapsto .(L)$  and  $S \mapsto .id$

# Example: Constructing the DFA



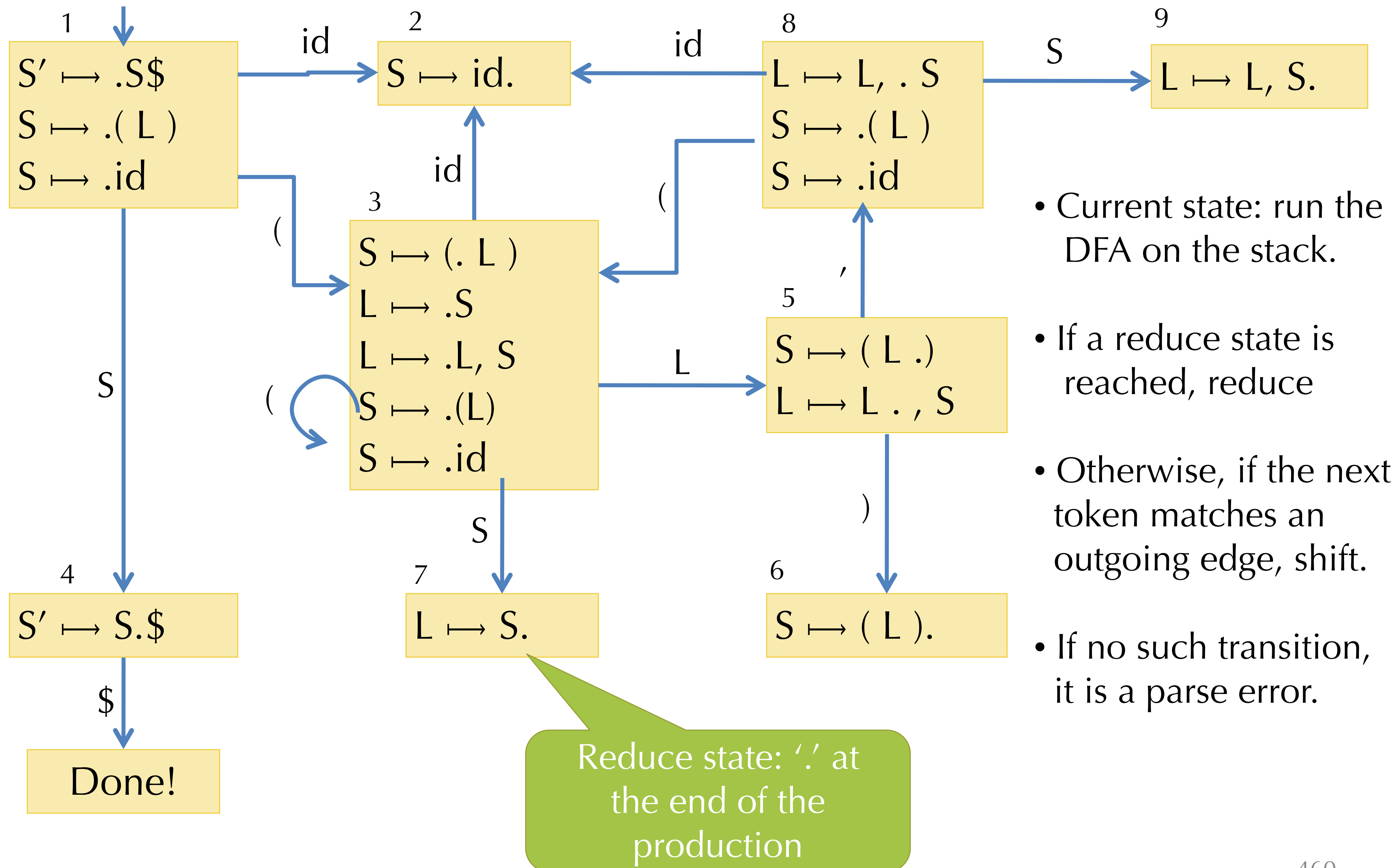
$S' \mapsto S\$$   
 $S \mapsto (L) \mid id$   
 $L \mapsto S \mid L, S$

- If the `.` is at the end of the production, then it is possible to reduce
- otherwise we should shift

Reduce state: `'.'` at the end of the production



# Full DFA for the Example



- Current state: run the DFA on the stack.
- If a reduce state is reached, reduce
- Otherwise, if the next token matches an outgoing edge, shift.
- If no such transition, it is a parse error.

# Using the DFA

- Run the parser stack through the DFA.
- The resulting state tells us which productions might be reduced next.
  - If not in a reduce state, then shift the next symbol and transition according to DFA.
  - If in a reduce state,  $X \mapsto \gamma$  with stack  $\alpha\gamma$ , pop  $\gamma$  and push  $X$ .
- Optimization: No need to re-run the DFA from beginning every step
  - Store the state with each symbol on the stack: e.g.  $_1(3(3L_5)_6$
  - On a reduction  $X \mapsto \gamma$ , pop stack to reveal the state too:  
e.g. From stack  $_1(3(3L_5)_6$  reduce  $S \mapsto ( L )$  to reach stack  $_1(3$
  - Next, push the reduction symbol: e.g. to reach stack  $_1(3S$
  - Then take just one step in the DFA to find next state:  $_1(3S_7$

# Implementing the Parsing Table

Represent the DFA as a table of shape:

state \* (terminals + nonterminals)

- Entries for the “action table” specify two kinds of actions:
  - Shift and goto state n
  - Reduce using reduction  $X \mapsto \gamma$ 
    - First pop  $\gamma$  off the stack to reveal the state
    - Look up X in the “goto table” and goto that state





# Example Parse Table

|   | (               | )               | id              | ,               | \$              | S  | L  |
|---|-----------------|-----------------|-----------------|-----------------|-----------------|----|----|
| 1 | s3              |                 | s2              |                 |                 | g4 |    |
| 2 | $S \mapsto id$  | $S \mapsto id$  | $S \mapsto id$  | $S \mapsto id$  | $S \mapsto id$  |    |    |
| 3 | s3              |                 | s2              |                 |                 | g7 | g5 |
| 4 |                 |                 |                 |                 | DONE            |    |    |
| 5 |                 | s6              |                 | s8              |                 |    |    |
| 6 | $S \mapsto (L)$ | $S \mapsto (L)$ | $S \mapsto (L)$ | $S \mapsto (L)$ | $S \mapsto (L)$ |    |    |
| 7 | $L \mapsto S$   | $L \mapsto S$   | $L \mapsto S$   | $L \mapsto S$   | $L \mapsto S$   |    |    |
| 8 | s3              |                 | s2              |                 |                 | g9 |    |
| 9 | $L \mapsto L,S$ | $L \mapsto L,S$ | $L \mapsto L,S$ | $L \mapsto L,S$ | $L \mapsto L,S$ |    |    |

sx = shift and goto state x

gx = goto state x (used when we reduce)

# Example

- Parse the token stream:  $(x, (y, z), w)\$$

| Stack         | Stream             | Action (according to table) |
|---------------|--------------------|-----------------------------|
| $\epsilon_1$  | $(x, (y, z), w)\$$ | s3                          |
| $\epsilon_1($ | $x, (y, z), w)\$$  | s2                          |
| $\epsilon_1($ | $, (y, z), w)\$$   | Reduce: $S \rightarrow id$  |
| $\epsilon_1($ | $, (y, z), w)\$$   | g7 (from state 3 follow S)  |
| $\epsilon_1($ | $, (y, z), w)\$$   | Reduce: $L \rightarrow S$   |
| $\epsilon_1($ | $, (y, z), w)\$$   | g5 (from state 3 follow L)  |
| $\epsilon_1($ | $, (y, z), w)\$$   | s8                          |
| $\epsilon_1($ | $(y, z), w)\$$     | s3                          |
| $\epsilon_1($ | $y, z), w)\$$      | s2                          |



# LR(0) Limitations

- An LR(0) machine only works if states with reduce actions have a *single* reduce action.
  - In such states, the machine *always* reduces (ignoring lookahead)
- With more complex grammars, the DFA construction will yield states with shift/reduce and reduce/reduce conflicts:

OK

$S \mapsto ( L ).$

shift/reduce

$S \mapsto ( L ).$   
 $L \mapsto .L , S$

reduce/reduce

$S \mapsto L , S.$   
 $S \mapsto , S.$

- Such conflicts can often be resolved by using a look-ahead symbol: SLR(1) or LR(1)

# Examples

- Consider the left associative and right associative “sum” grammars:

left

$$\begin{aligned} S &\mapsto S + E \mid E \\ E &\mapsto \text{number} \mid ( S ) \end{aligned}$$

right

$$\begin{aligned} S &\mapsto E + S \mid E \\ E &\mapsto \text{number} \mid ( S ) \end{aligned}$$

- One is LR(0) the other isn't... which is which and why?
- What kind of conflict do you get? Shift/reduce or Reduce/reduce?

# Examples

- Consider the left associative and right associative “sum” grammars:

left

$$\begin{aligned} S &\mapsto S + E \mid E \\ E &\mapsto \text{number} \mid ( S ) \end{aligned}$$

right

$$\begin{aligned} S &\mapsto E + S \mid E \\ E &\mapsto \text{number} \mid ( S ) \end{aligned}$$

- One is LR(0) the other isn't... which is which and why?
- What kind of conflict do you get? Shift/reduce or Reduce/reduce?

If the stack is a single E, then the state is

$$\begin{aligned} S &\mapsto E . + S \\ S &\mapsto E . \end{aligned}$$

shift-reduce conflict: we can either shift the + or reduce the E to an S.  
LR(0) parser can't decide



# SLR(1) (“simple” LR) Parsers

- What conflicts are there in LR(0) parsing?
  - reduce/reduce conflict: an LR(0) state has two reduce actions
  - shift/reduce conflict: an LR(0) state mixes reduce and shift actions
- Can we use lookahead to disambiguate?
- SLR(1) – uses the same DFA construction as LR(0)
  - modifies the actions based on lookahead. More powerful
- Suppose reducing an A nonterminal is possible in some state:
  - compute Follow(A) for the given grammar
  - if the lookahead symbol is in Follow(A), then reduce, otherwise shift
  - can disambiguate between reduce/reduce conflicts if the follow sets are disjoint

# LR(1) Parsing

- Yet more powerful than SLR(1)
- Algorithm is similar to LR(0) DFA construction:
  - LR(1) state = set of LR(1) items
  - An LR(1) item is an LR(0) item + a set of look-ahead symbols:
$$A \mapsto \alpha.\beta, \mathcal{L}$$
- LR(1) closure is a little more complex:
- Form the set of items just as for LR(0) algorithm.
- Whenever a new item  $C \mapsto .\gamma$  is added because  $A \mapsto \beta.C\delta, \mathcal{L}$  is already in the set, we need to compute its look-ahead set  $\mathcal{M}$ :
  1. The look-ahead set  $\mathcal{M}$  includes  $\text{FIRST}(\delta)$   
(the set of terminals that may start strings derived from  $\delta$ )
  2. If  $\delta$  derive  $\epsilon$  (i.e. it is nullable), then the look-ahead  $\mathcal{M}$  also contains  $\mathcal{L}$

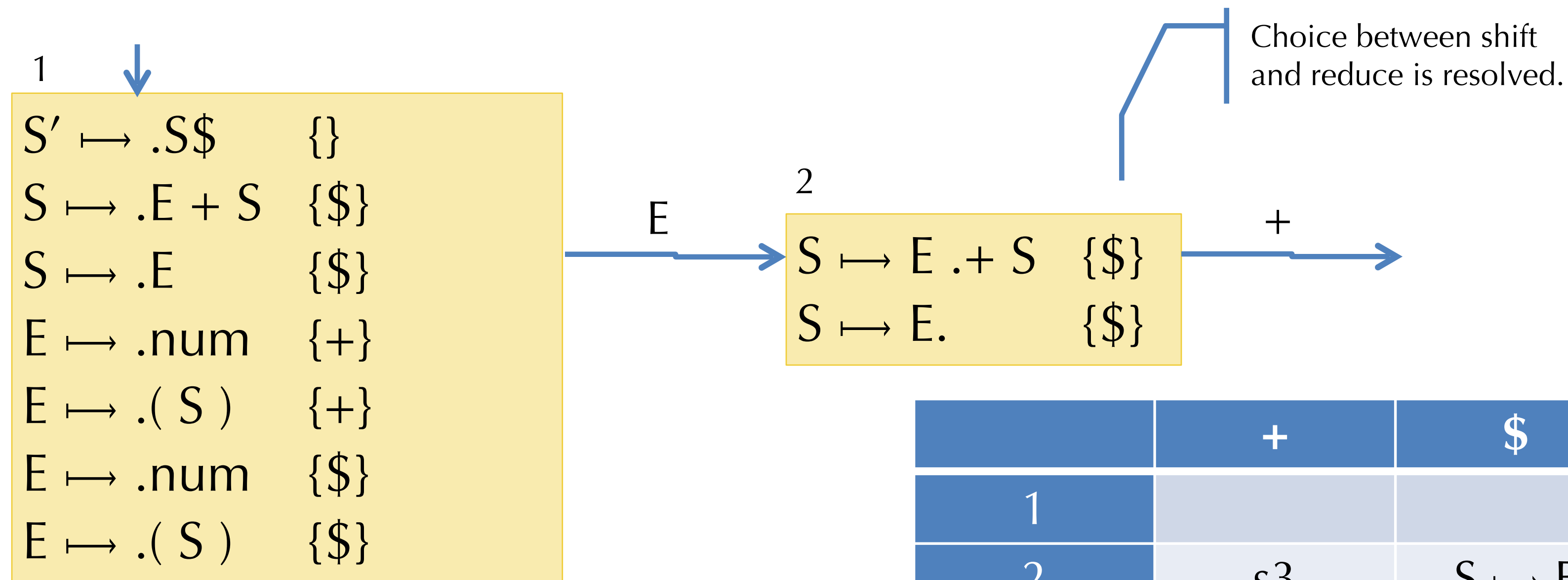
# Example Closure

$$\begin{aligned} S' &\mapsto S\$ \\ S &\mapsto E + S \mid E \\ E &\mapsto \text{number} \mid ( S ) \end{aligned}$$

- Start item:  $S' \mapsto .S\$$  ,  $\{\}$
- Since  $S$  is to the right of a  $'.'$ , add:  
 $S \mapsto .E + S$  ,  $\{\$\}$       Note:  $\{\$\}$  is  $\text{FIRST}(\$)$   
 $S \mapsto .E$  ,  $\{\$\}$
- Need to keep closing, since  $E$  appears to the right of a  $'.'$  in  $' .E + S'$ :  
 $E \mapsto .\text{number}$  ,  $\{+\}$       Note:  $+$  added for reason 1  
 $E \mapsto .( S )$  ,  $\{+\}$        $\text{FIRST}(+ S) = \{+\}$
- Because  $E$  also appears to the right of  $'.'$  in  $' .E'$  we get:  
 $E \mapsto .\text{number}$  ,  $\{\$\}$       Note:  $\$$  added for reason 2  
 $E \mapsto .( S )$  ,  $\{\$\}$        $\delta$  is  $\epsilon$
- All items are distinct, so we're done



# Using the DFA



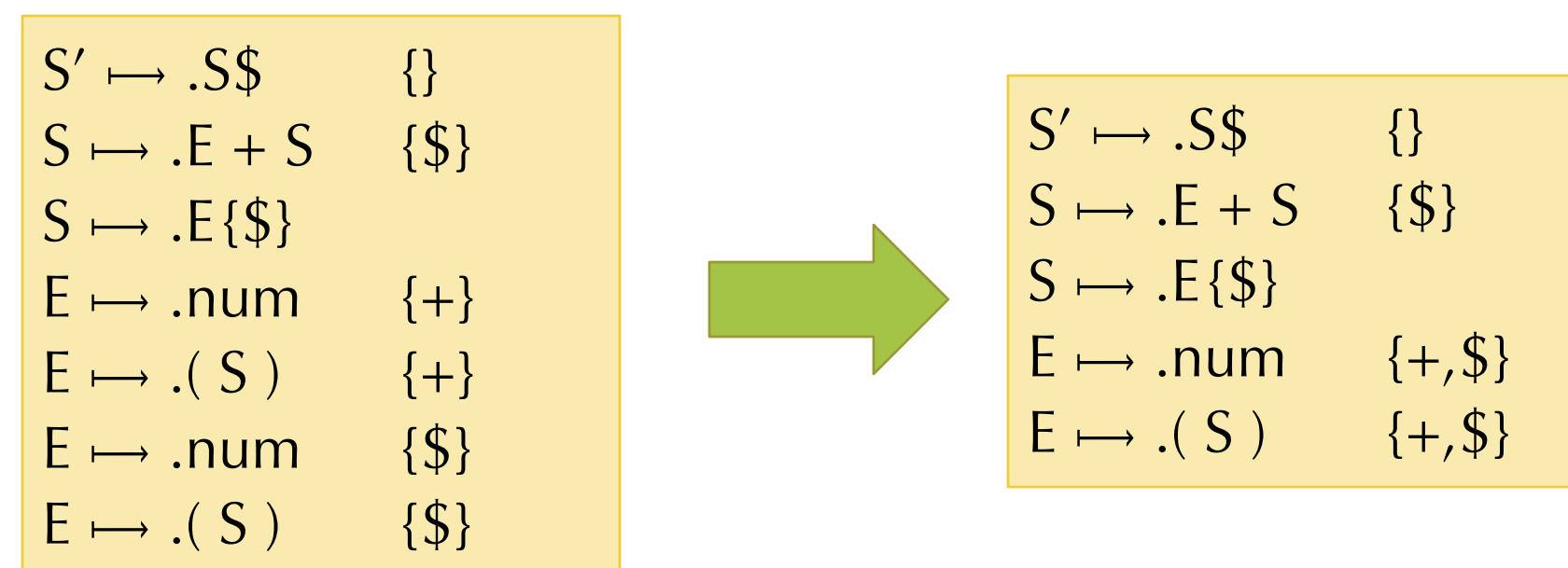
|   | +  | \$            | E  |
|---|----|---------------|----|
| 1 |    |               | g2 |
| 2 | s3 | $S \mapsto E$ |    |

Fragment of the Action & Goto tables

- The behavior is determined if:
  - There is no overlap among the look-ahead sets for each reduce item, and
  - None of the look-ahead symbols appear to the right of a '.'

# LR variants

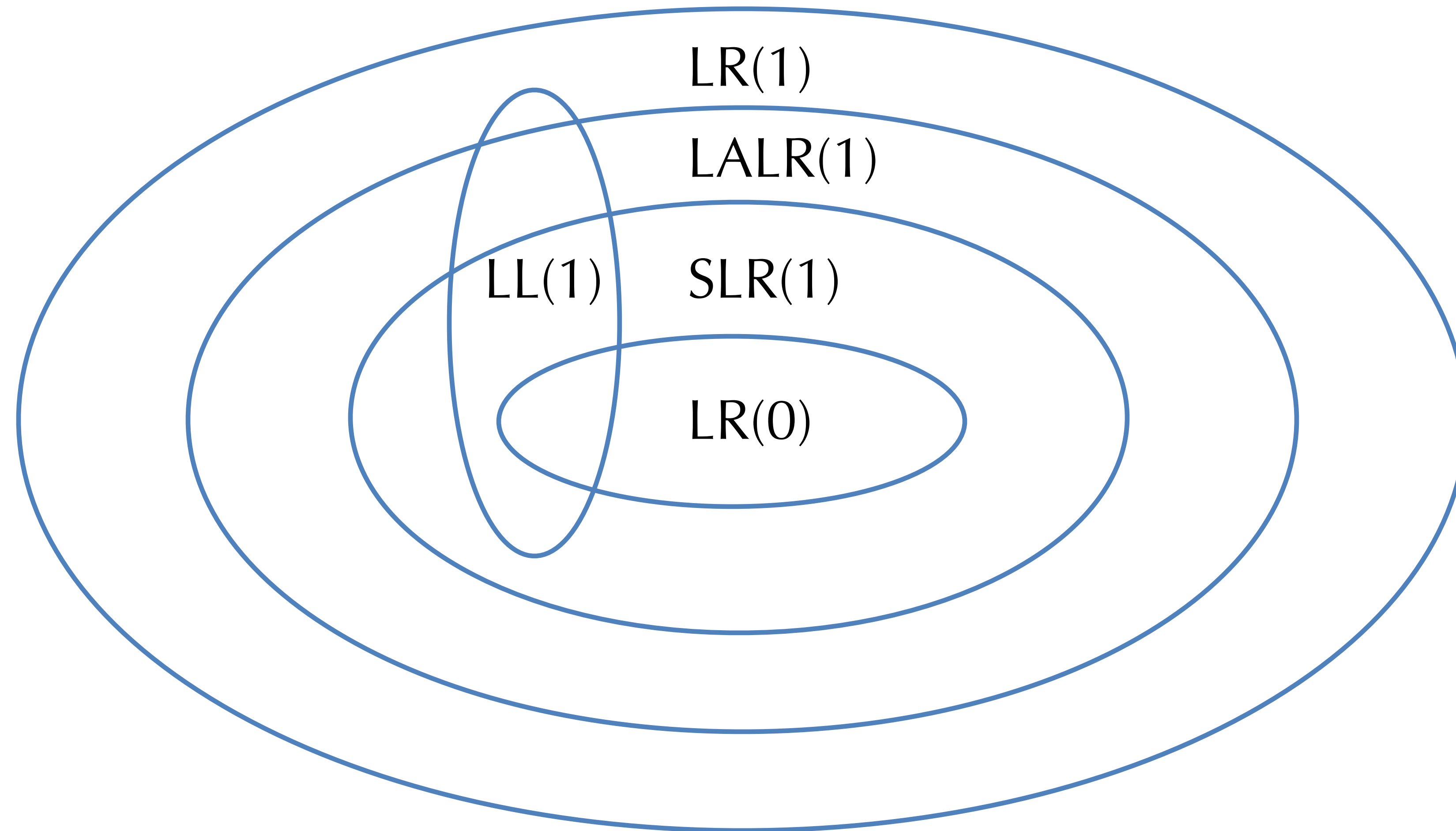
- LR(1) gives maximal power out of a 1 look-ahead symbol parsing table
  - DFA + stack is a push-down automaton
- In practice, LR(1) tables are big.
  - Some modern implementations (e.g., menhir) directly generate code rather than use tables
- LALR(1) = “Look-ahead LR”
  - Merge any two LR(1) states whose items are identical except for the look-ahead sets:



- Such merging can lead to nondeterminism (e.g., reduce/reduce conflicts), but
  - Results in a much smaller parse table and works well in practice
  - This is the usual technology for automatic parser generators: yacc, ocaml yacc
- GLR = “Generalized LR” parsing
  - Efficiently compute the set of *all* parses for a given input
  - Later passes should disambiguate based on other context



# Classification of Grammars



# Wrap Up

# What Have We Learned

- Frontend (lexing, parsing, semantic validation)
- Static single assignment intermediate representation
- Backend code generation, x86 assembly code
- Programming language features: lexical scope, procedures, tail calls, dynamic typing, heap allocation, closures.
- Programming lessons
  - working with an evolving codebase
  - implementing programs from rich specifications
  - testing!



# What We Didn't Cover

## (Lots)

- Other lexing/parsing techniques: PEGs, Earley Parsing, Regular expression derivatives
- Macro systems: C style pre-processors, Scheme/Rust style macros
- Static Types systems: unification, overloading
- Implementing objects
- Concurrency/parallelism
- Complex control flow mechanisms: Exceptions, generators, first-class continuations
- JIT compilation, bytecode interpreters
- Other IRs: continuation-passing style, 3address codes
- Efficient data structures for compiler IRs and analyses
- Sophisticated code generation: tiling

# Where to Learn More?

## Classes

- Classes here at UM:
  - EECS 583: Graduate Compilers. Practical experience with LLVM, read research paper, implement more sophisticated optimizations.
  - EECS 490 and 590: undergraduate and graduate Programming languages. More on type systems, programming language features, functional programming, mathematical reasoning
  - My EECS 598 in Fall 2025: Category Theory. Mathematics of programming language semantics. Prove logical reasoning is sound, prove that programs are well-behaved.

# Where to Learn More?

## Open Source

- Language Implementations are typically open source. Rust in particular has a very active and welcoming community
- Common intermediate languages: LLVM, Cranelift and MLIR, all based on versions of SSA representation
- Compiler frontend projects: Tree-sitter, LALRpop



# Where to Learn More?

## Academic Research

- Michigan PL and Software engineering (MPLSE)
  - [mplse.org](http://mplse.org)
  - Reach out to me if you are interested in research on programming language design, implementation and semantics!
- Academic Conferences
  - PLDI (Programming Language **Design** and **Implementation**)
  - POPL (**P**rin**c**iples of Programming Languages)
  - ICFP (**F**unctional Programming)
  - OOPSLA (**O**bject-oriented...)
  - CC (Compiler Construction)
  - Many more...

# Thank You

- Thanks to Yuchen for great work as GSI.
- Thank you for making this fun and rewarding to teach.
- Please fill out teaching evaluations!
  - Always trying to improve the course. Please let us know what you thought was working/not working.