



EECS 483: Compiler Construction

Lecture 26:
Class Wrap Up and Exam Review

April 21
Winter Semester 2025

Plan for Today

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

Exam Review

Final Exam Info

- Final Exam on Wednesday 04/30 4-6pm
 - Location:
 - DOW 1010 (uniqname starts with A-L)
 - DOW 1014 (uniqname starts with M-Z)
 - Topics: Assignments 4 and 5, lecture material after spring break.
 - Exam Review on Monday, 04/21
 - 1 page of notes, double sided ok, printed or written ok.
 - Practice material:
<https://maxsnew.com/teaching/eecs-483-wn24/syllabus.html> questions about lexing/parsing/analysis/optimization (some are in midterms, some are in finals)
Appel book, Dragon book linked on webpage have exercises on lexing/parsing and optimization/register allocation

Second Half Course Major Topics

1. Datatypes

- tagging schemes
- heap allocation
- assertions

2. 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

3. Lexing

- Tokens
- Regular expressions
- Non-deterministic and deterministic finite automata
- limitations

4. Parsing

- Context-free grammars
- Parse trees and abstract syntax trees
- Derivations
- Associativity, precedence
- Top-down parsing
 - LL(1)
 - conversion to LL(1), left-factoring, etc
- Bottom-up parsing
 - LR(1)
 - First set, follow set, nullability
 - shift/reduce, reduce/reduce conflicts

Datatypes

Representing Dynamically Typed Values

In Adder/Boa/Cobra, all runtime values were integers.

In Diamondback, a runtime value must have both a type tag and a value that matches the type tag

How should we represent tags and values in our compiled program?

Representing Dynamically Typed Values

Approach 3: compromise

A snake value is a 64-bit value.

Use the least significant bits of the value as a **tag**.

Represent simple data like integers, booleans within the 64-bits

Represent large datatypes like arrays, closures, structs as pointers to the heap

Upside: use stack allocation more often

Downside: can't fit 64 bits and a tag...

Roughly the approach used in high-performance Javascript engines (v8) as well as some garbage-collected typed languages (OCaml)

Representing Dynamically Typed Values

To implement our compiler, we need to specify

1. How each of our Snake values are represented at runtime
2. How to implement the primitive operations on these representations

Integers

Implement a snake integer as a 63-bit signed integer followed by a 0 bit to indicate that the value is an integer

Number	Representation
1	0b0000000_0000....0000_00000010
6	0b0000000_0000....0000_00001100
-1	0b1111111_1111....1111_11111110

i.e., represent a 63-bit integer n as the 64-bit integer $2 * n$

Booleans

The least significant bit must be 1 to distinguish from integers

Use least significant bits `0b01` to distinguish from integers and other datatypes

Use the remaining 62 bits to encode true and false as before as 1 and 0

Number	Representation
true	<code>0b0000000_0000....000_0000101</code>
false	<code>0b0000000_0000....000_0000001</code>

$2^{62} - 2$ bit patterns are therefore "junk" in this format

Boxed Data

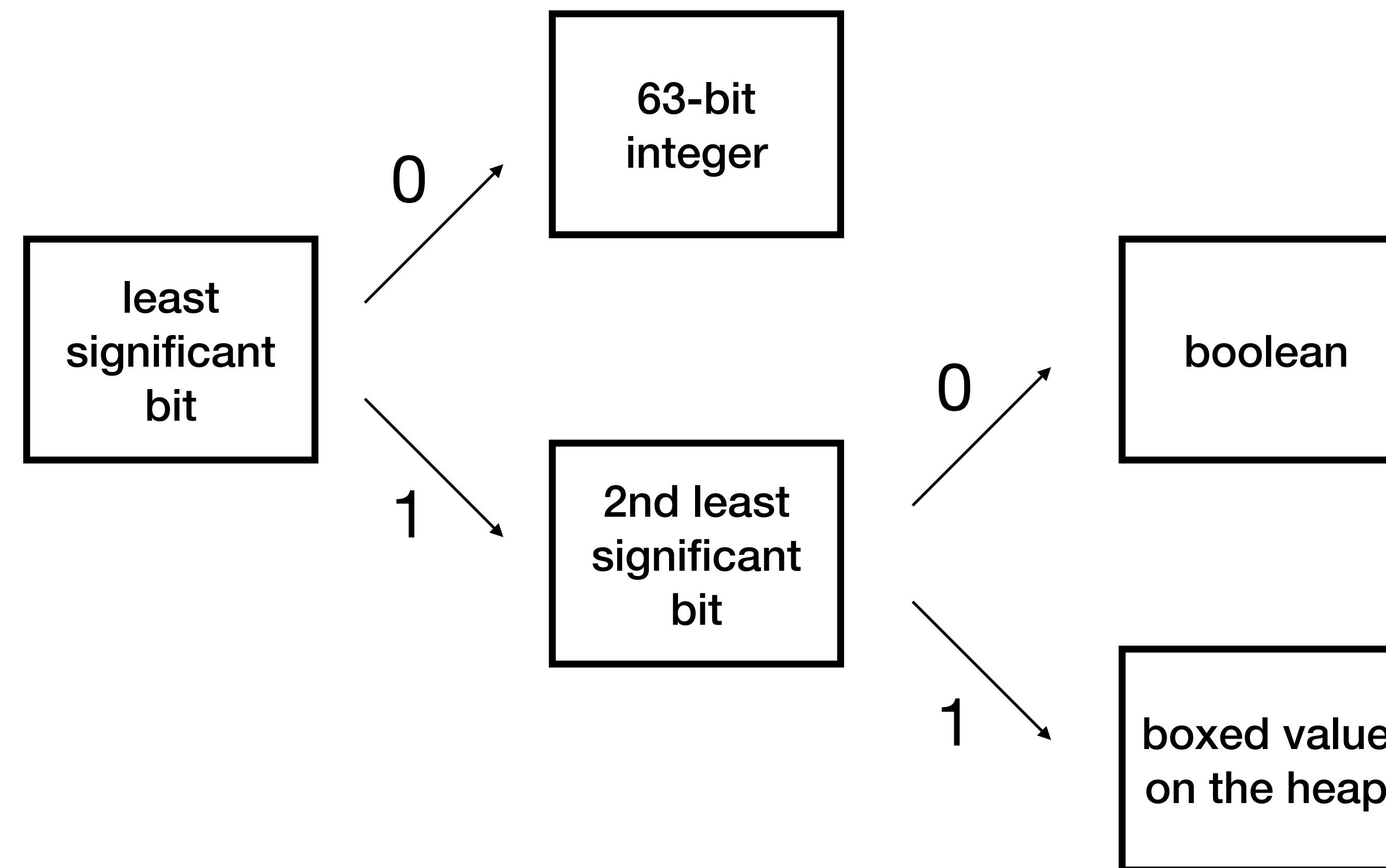
The least significant bit must be 1 to distinguish from integers

Use least significant bits `0b11` to distinguish from booleans.

Use remaining 62-bits to encode a pointer to the data on the heap

Why is this ok? Discuss more thoroughly on Wednesday

Representing Dynamically Typed Values



Compiling Dynamic Typing

We know what the source semantics is and what kind of assembly code we want to generate.

In implementing the compiler, we now we have a design choice: in what phase of the compiler do we actually "implement" dynamic typing?

1. Implement everything in x86 code generation
2. Implement everything in lowering to SSA
3. Implement in multiple passes

Compiling Dynamic Typing

Approach 3: implement dynamic typing in multiple passes

In lowering to SSA, make some aspects of dynamic typing explicit but leave the tag checking as primitive operations.

Implement the tag checking in the x86 code generation.

Compiling Dynamic Typing

Approach 3: implement **some** dynamic typing in SSA lowering

Diamondback

$x * y$

SSA

`assertInt(x)`

`assertInt(y)`

`half = x >> 1`

`r = half * y`

`ret r`

Insert type tag assertions in SSA, implement bit-twiddling manually

Compiling Dynamic Typing

Optimization opportunity

Diamondback

```
def fact(x):
    if x == 0:
        1
    else:
        x * fact(x - 1)
in
fact(7)
```

SSA

```
...
tmp1 = x - 1
tmp2 = call fact(tmp1)
assertInt(x)
assertInt(tmp2)
r = x * tmp2
ret r
```

with a simple **static analysis** determine that
x, tmp2 always have the correct tag for an
Int. Remove unnecessary assertions

Allocating Arrays

Where should the contents of our arrays be stored?

- Stack?
- Heap?

Heap Allocation

The heap contains data whose lifetime is not tied to a local stack frame.

This makes the usage of the data more flexible, but complicates the question of when the data is **deallocated**.

For today, let's assume we do not deallocate memory.

A strategy used in some specialized applications (missiles)

Today's simple heap model: the heap is a large region of memory, disjoint from the stack, some of it is used, and we have a pointer to the next available portion of memory.

Implementing Arrays

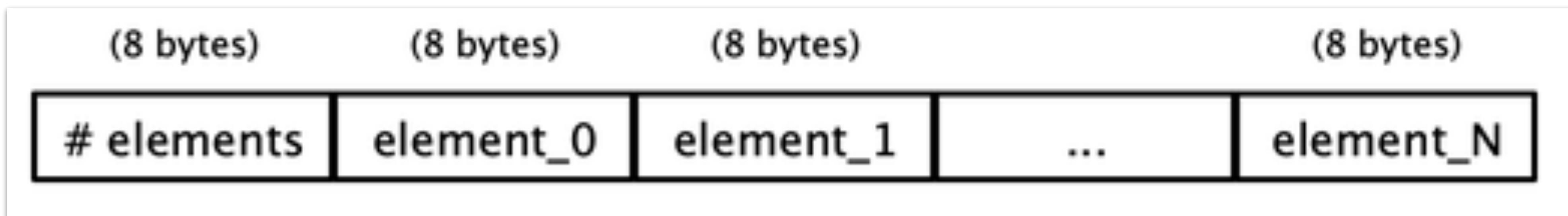
When we implement arrays, we have two different representations to define:

1. How they are stored as "objects" in the heap
2. How they are represented as Snake values

Arrays as Objects

What data does an array need to store?

1. Need to layout the values sequentially so we can implement get/set
2. Need to store the **length** of the array to implement length as well as bounds checking for get/set.



Arrays as Values

The Snake value we store on the stack for an array is a **tagged pointer** to the array stored on the heap.

We overwrite the 2 least significant bits of the pointer with the tag 0b11.

This is safe, as long as those 2 least significant bits of the pointer contain no information, i.e., if they are always 0.

2 least significant bits of a pointer are 0 means the address is a multiple of 4, meaning the address is at a 4-byte alignment.

All arrays on our heap take up size that is a multiple of 8 bytes, so as long as the base of the heap is 4-byte aligned, we maintain this invariant.

Optimization

Where Were We?

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

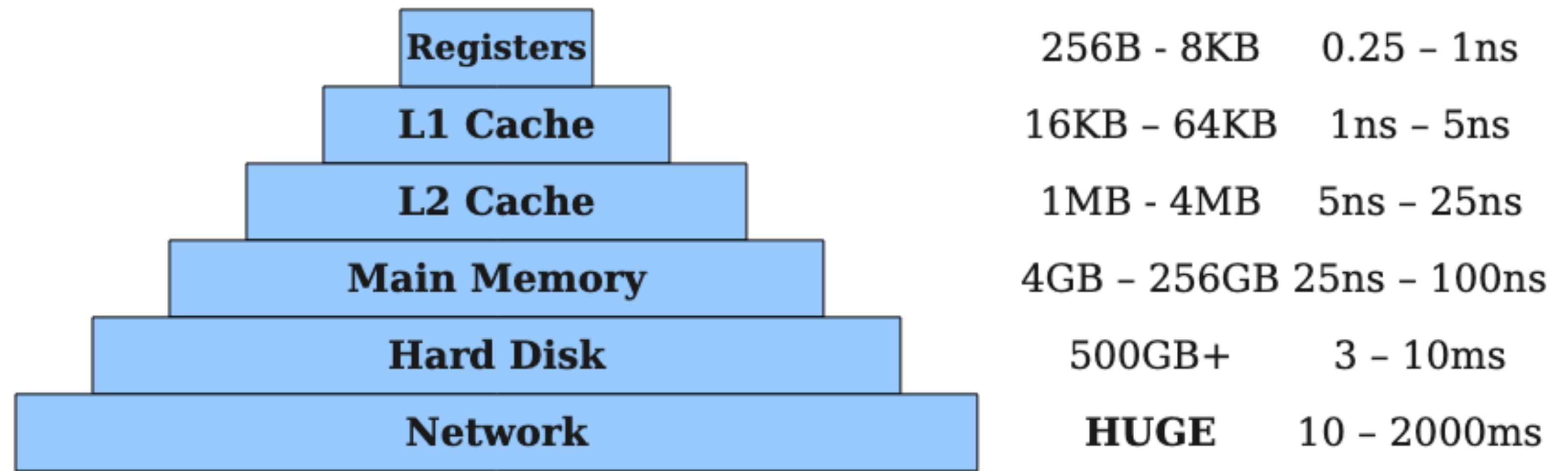
But our generated code has high **constant factors** in its complexity:

- 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

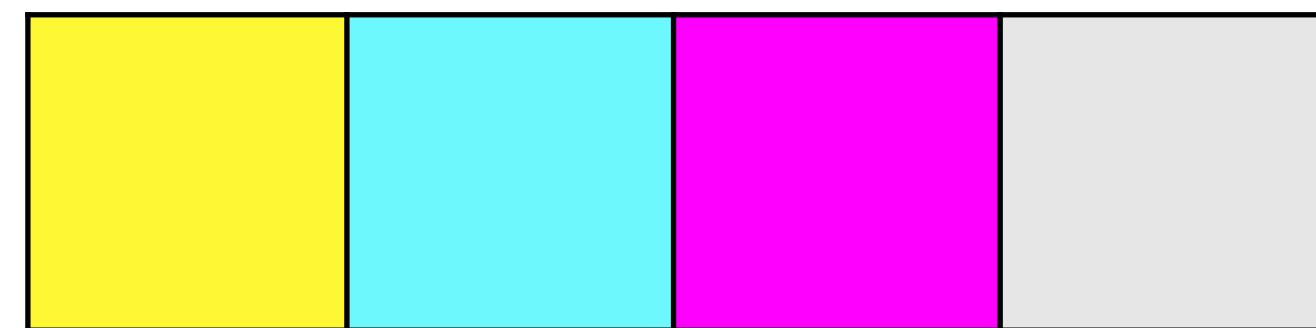
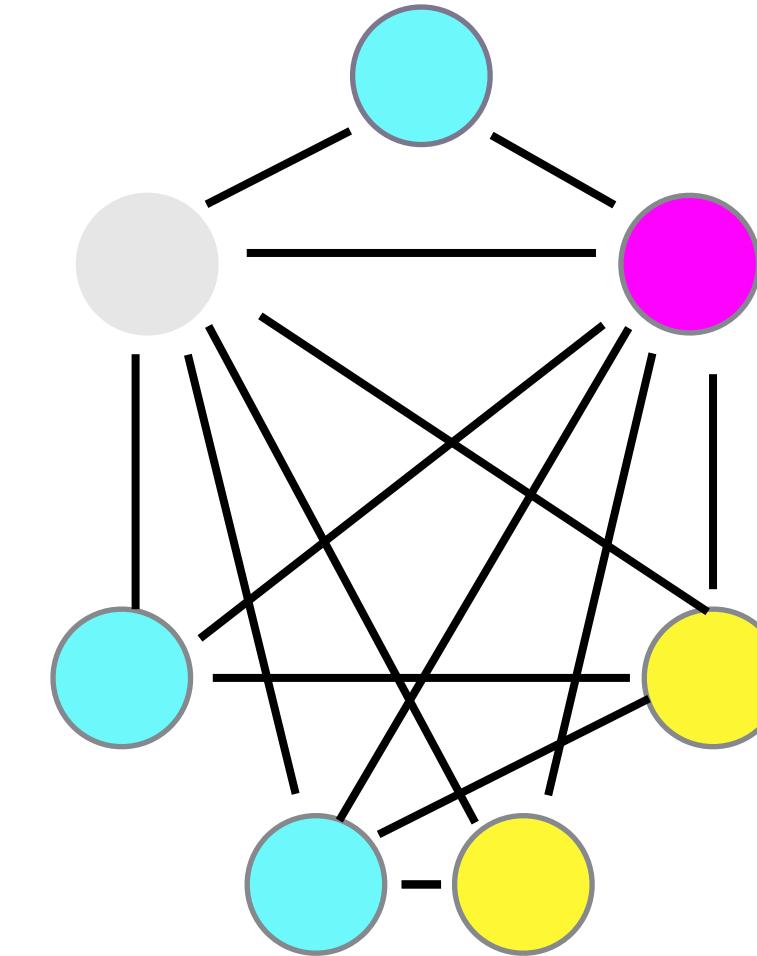
High computational complexity: often the slowest part of the compiler

Register Allocation: 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

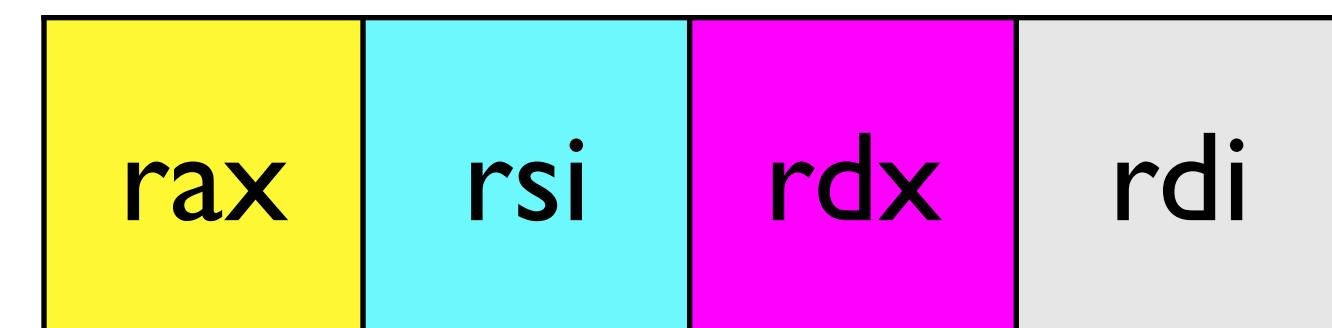
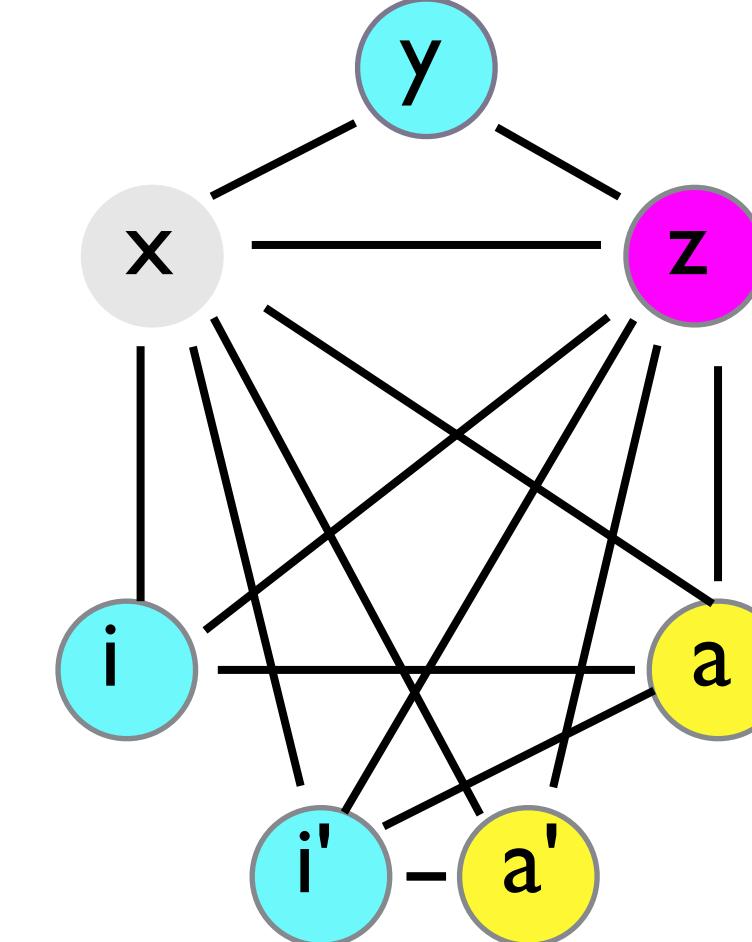
Graph Coloring register allocation:

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

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

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

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 well-formed.

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

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

Liveness Analysis: Example

```
f(x,y,z):  
1 loop(i,a):  
5   thn():  
6     r = a * z  
7   ret r  
8 else():  
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	Round 2
1:	{x, z}	1: ?
2:	{y}	2: ?
3:	{a, i, x, z}	3: ?
4:	{a, b, i, x, z}	4: ?
5/6:	{a, z}	5/6: ?
7:	{r}	7: ?
8/9:	{a, i, x}	8/9: ?
10:	{a, i', x}	10: ?
11:	{a', i'}	11: ?

Liveness Analysis: Example

```
f(x,y,z):  
1 loop(i,a):  
5   thn():  
6     r = a * z  
7   ret r  
8 else():  
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:	{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

```
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:	{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'

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
- Parameters of a block all must mutually conflict, as they are assigned to simultaneously.

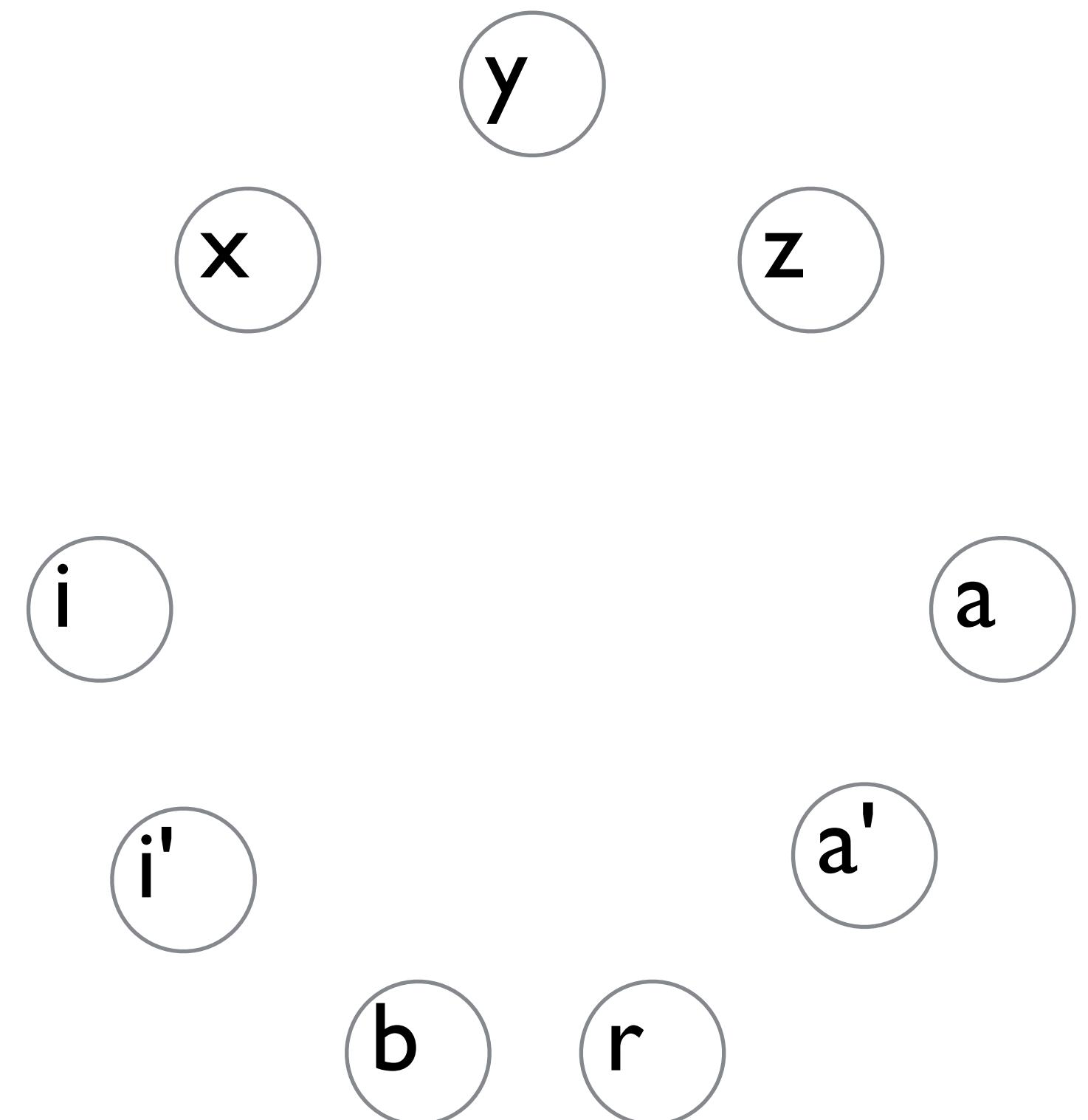
This is an overapproximation to true conflicts

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}

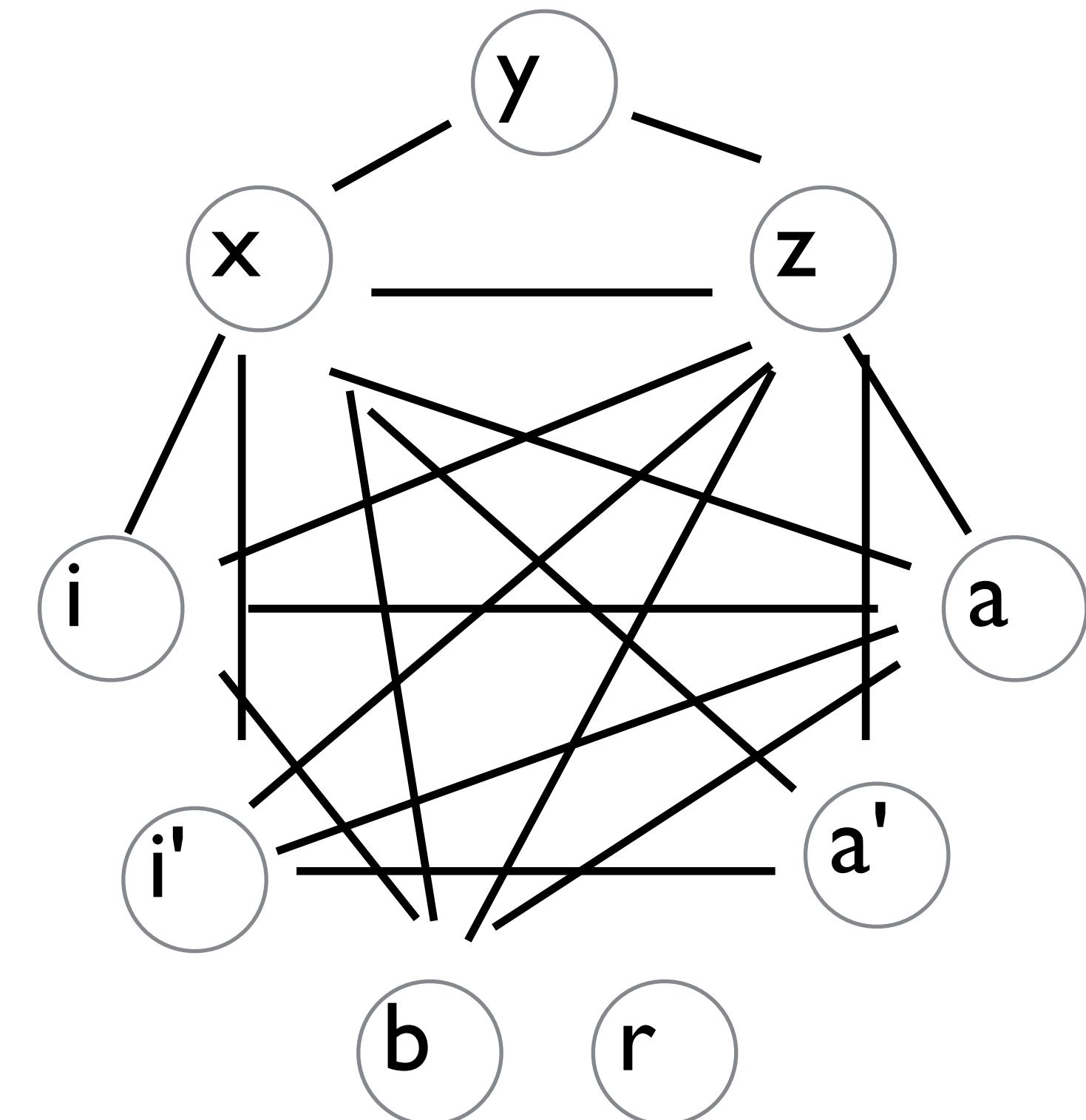


Conflict Analysis

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}



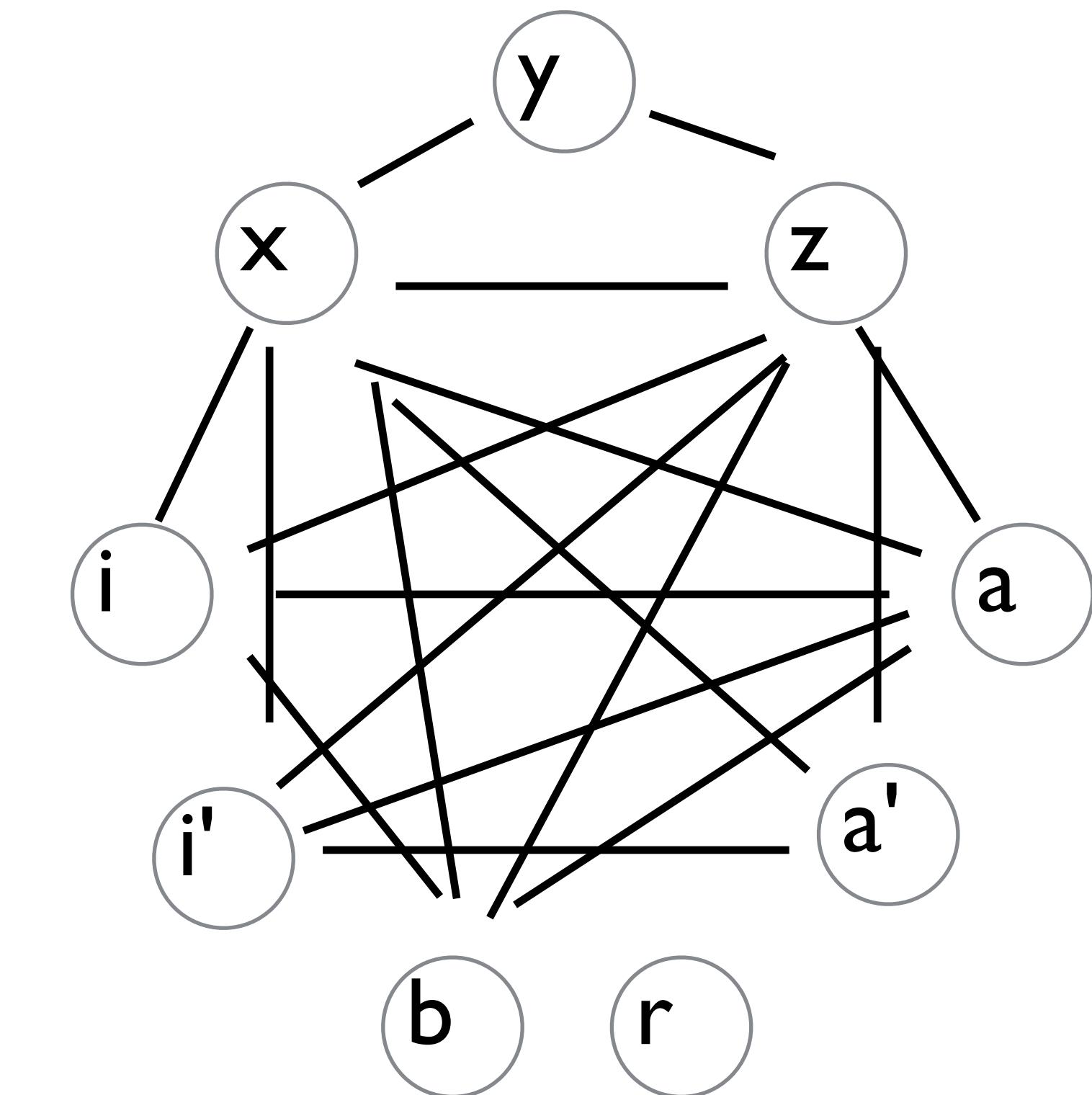
SSA Program

```
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}

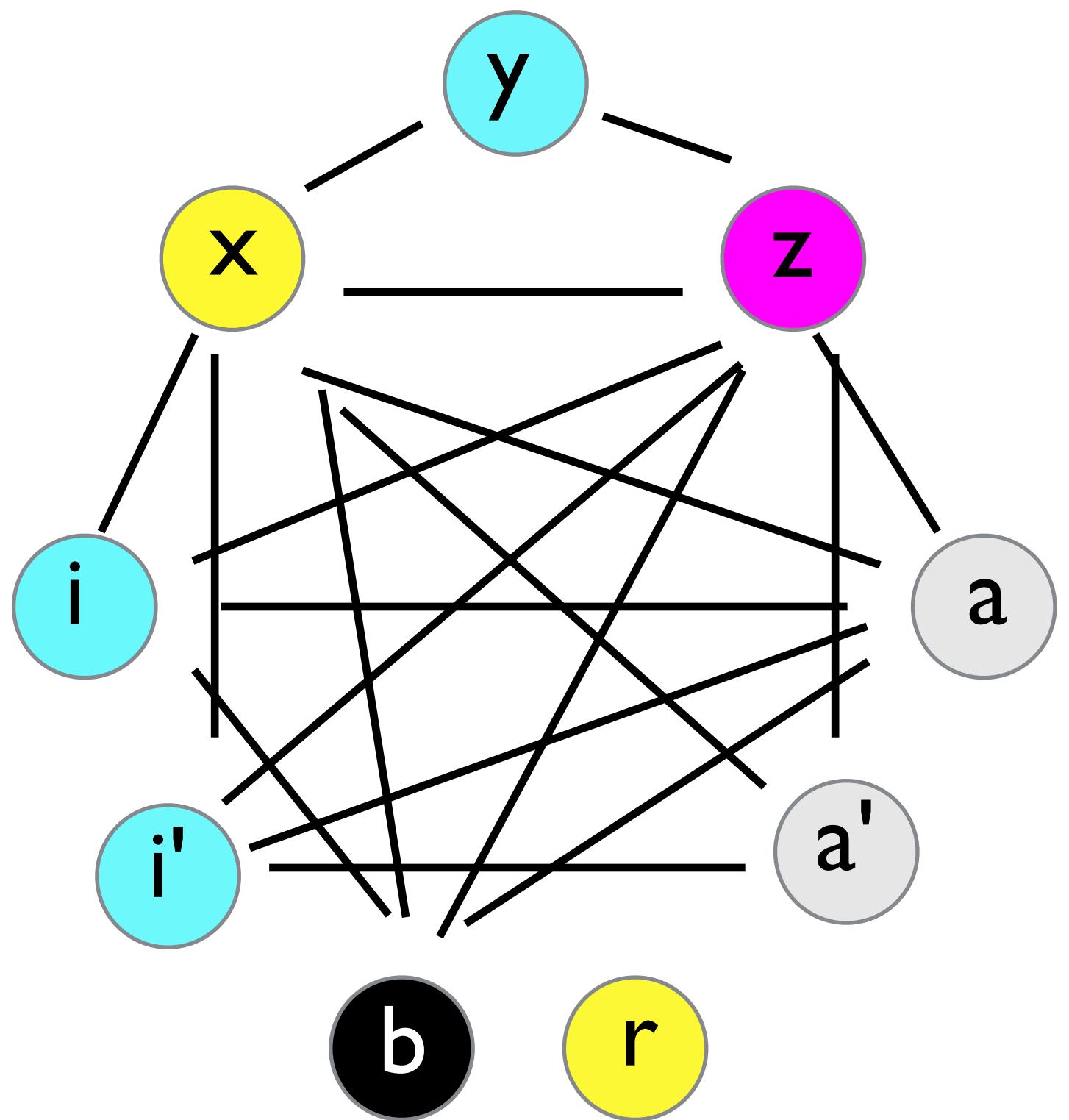
Interference Graph



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)
```

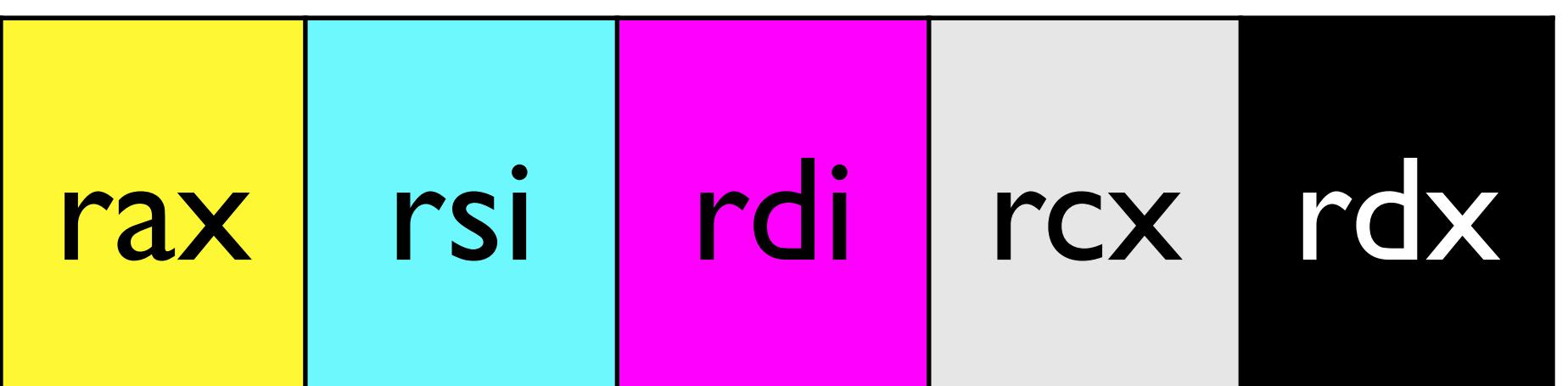
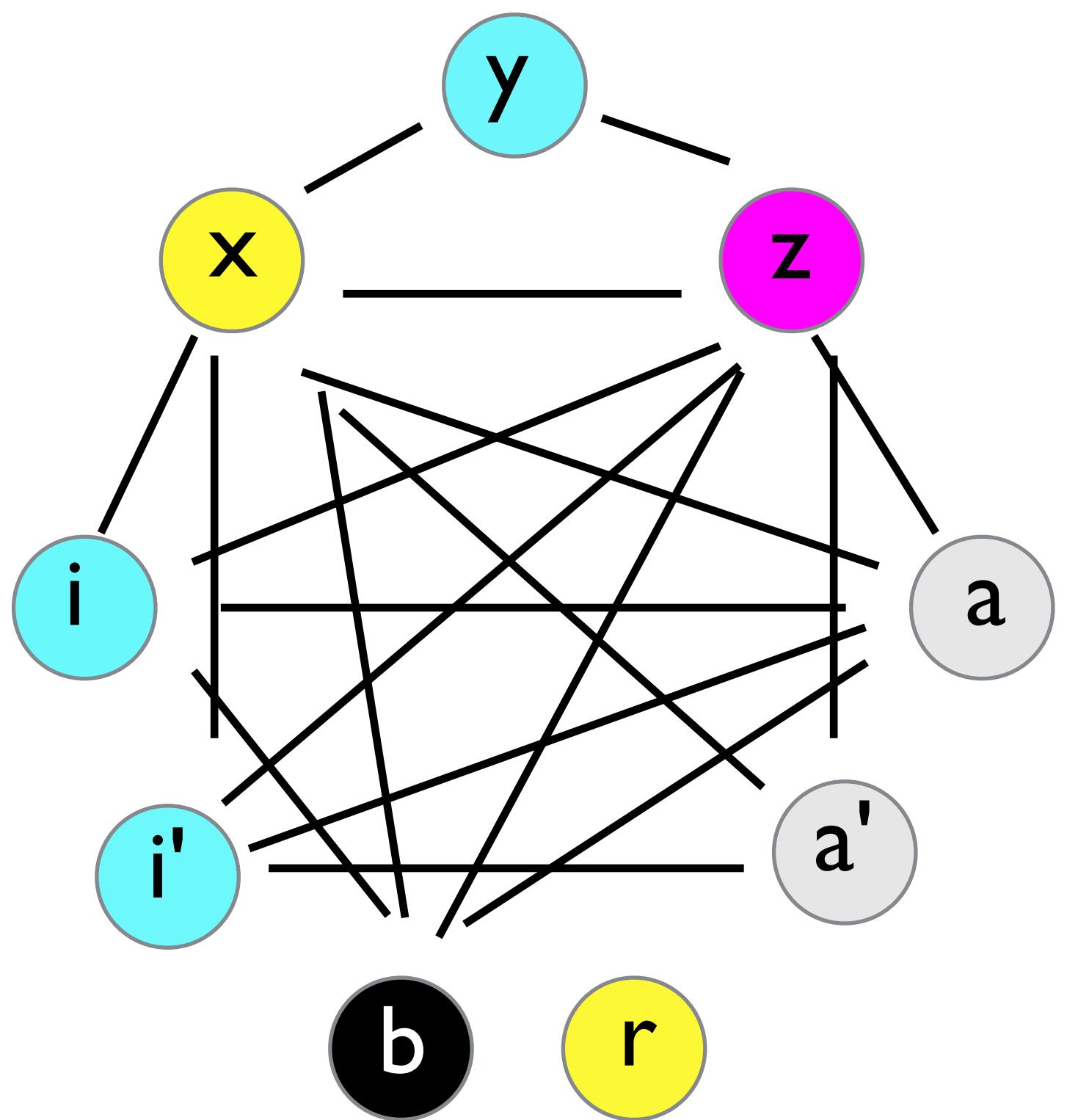
Interference Graph



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

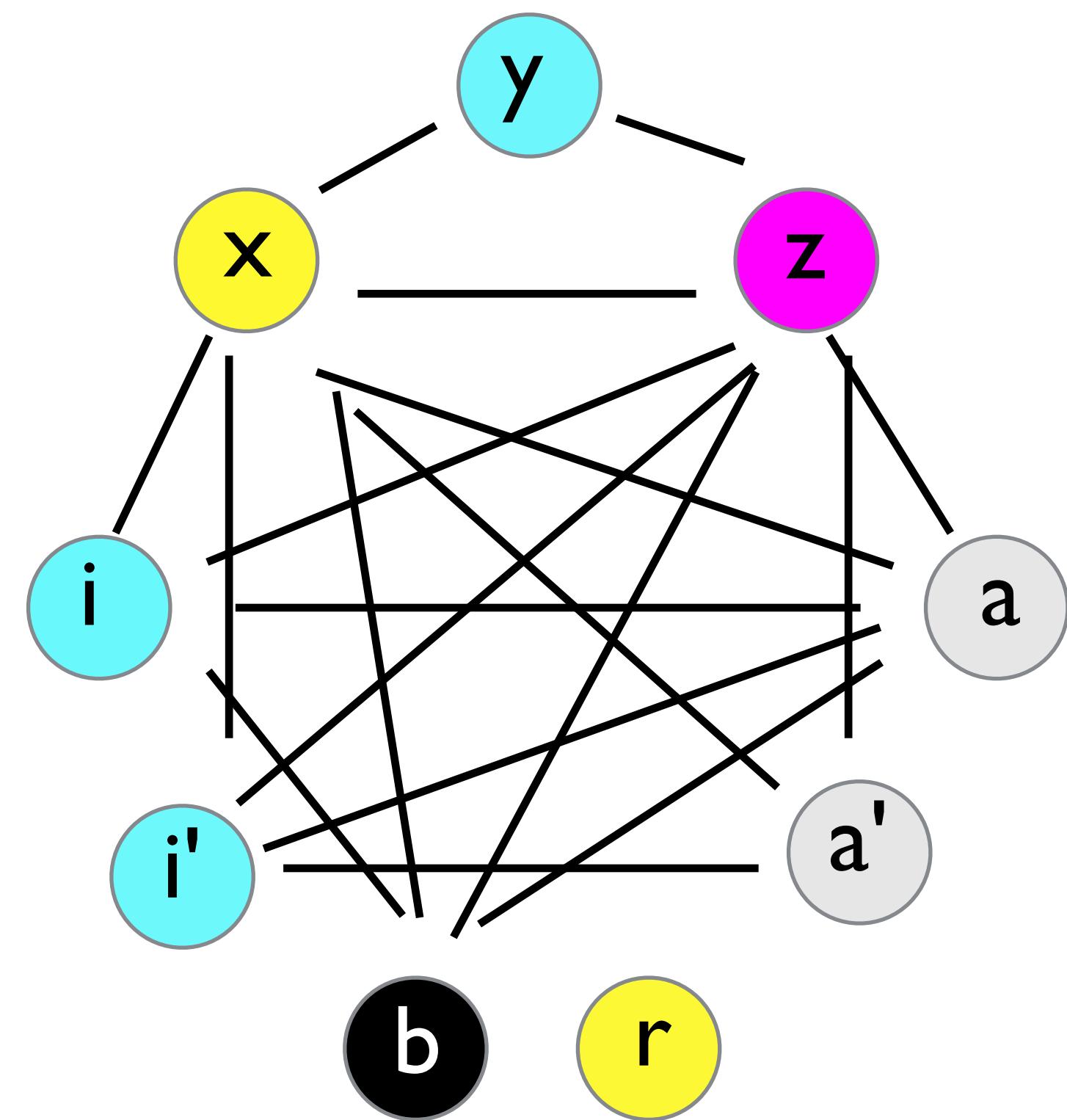


SSA Program

```
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

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**
- For every graph, there is a elimination ordering such that Chaitin's algorithm produces an optimal coloring
 - therefore finding this optimal elimination ordering for a general graph is NP-complete

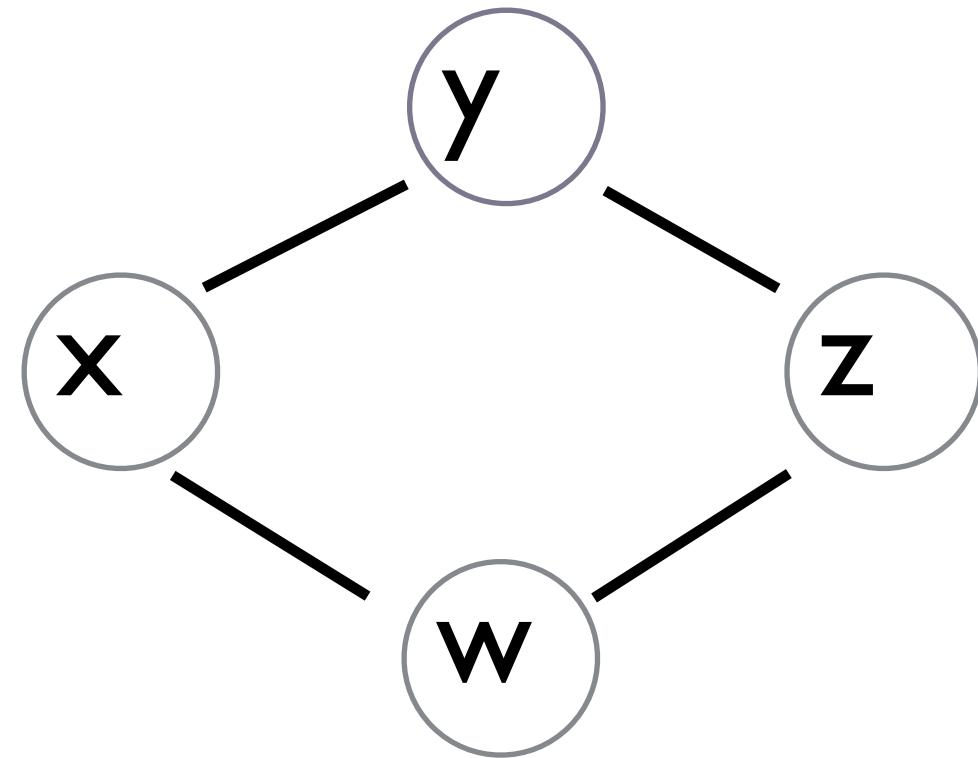
Graph Coloring SSA Programs

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

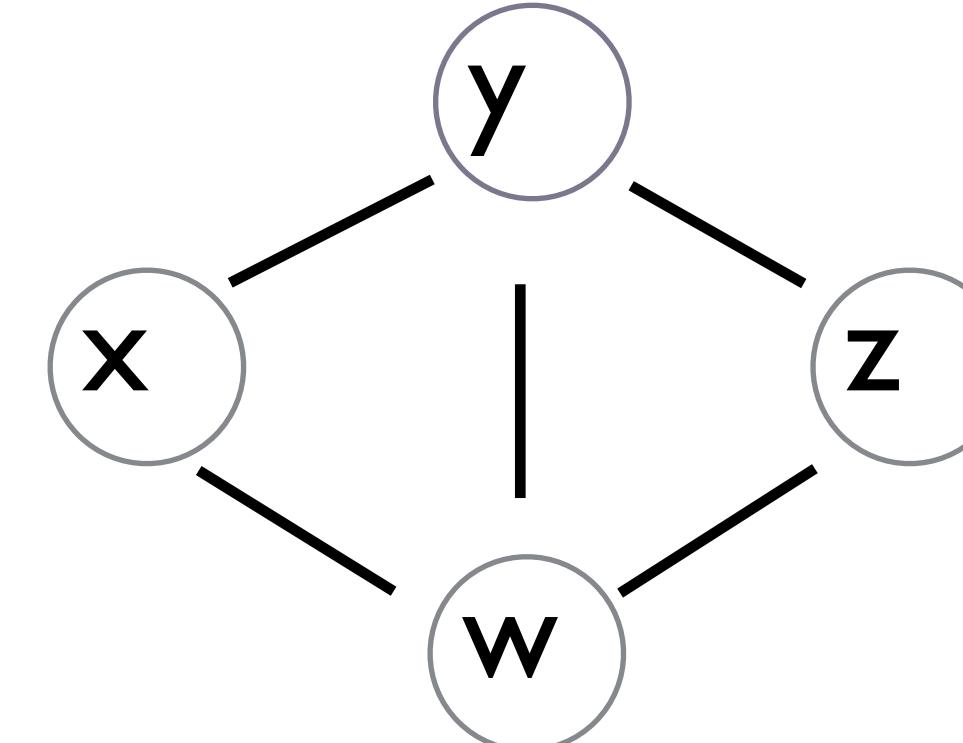
Graph Coloring SSA Programs

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 ≥ 4 nodes has a **chord**



Not chordal



chordal

Coloring Chordal Graphs

Theorem: Every chordal graph has a **perfect elimination ordering**

Coloring Chordal Graphs

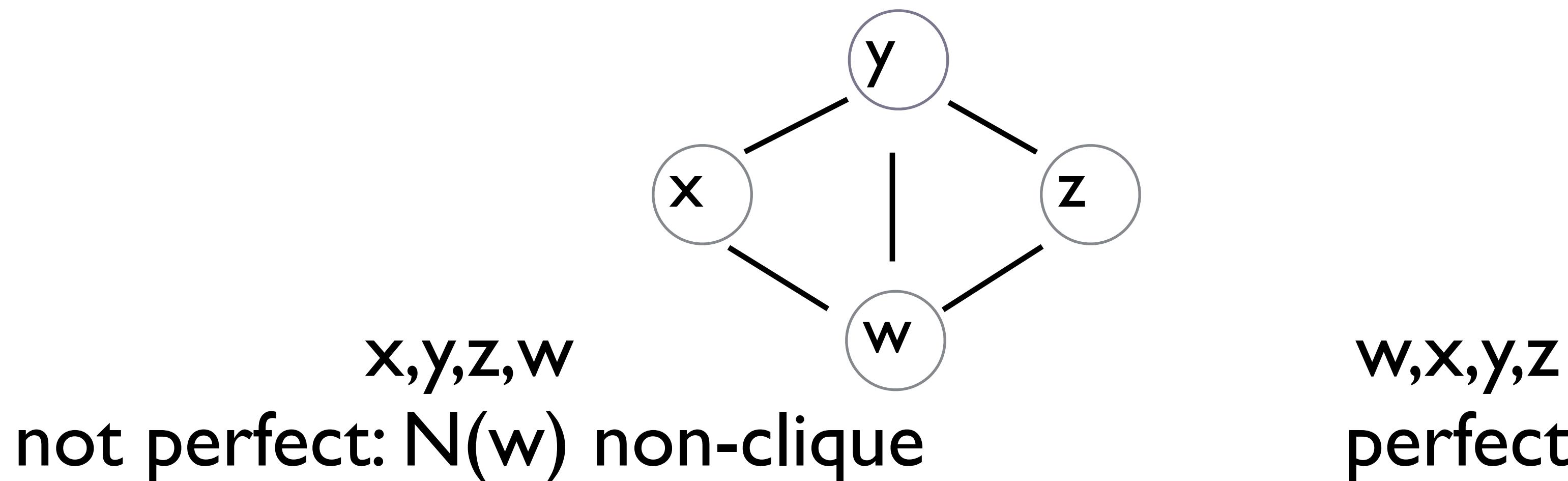
Theorem: Every chordal graph has a **perfect elimination ordering**

- 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

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



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

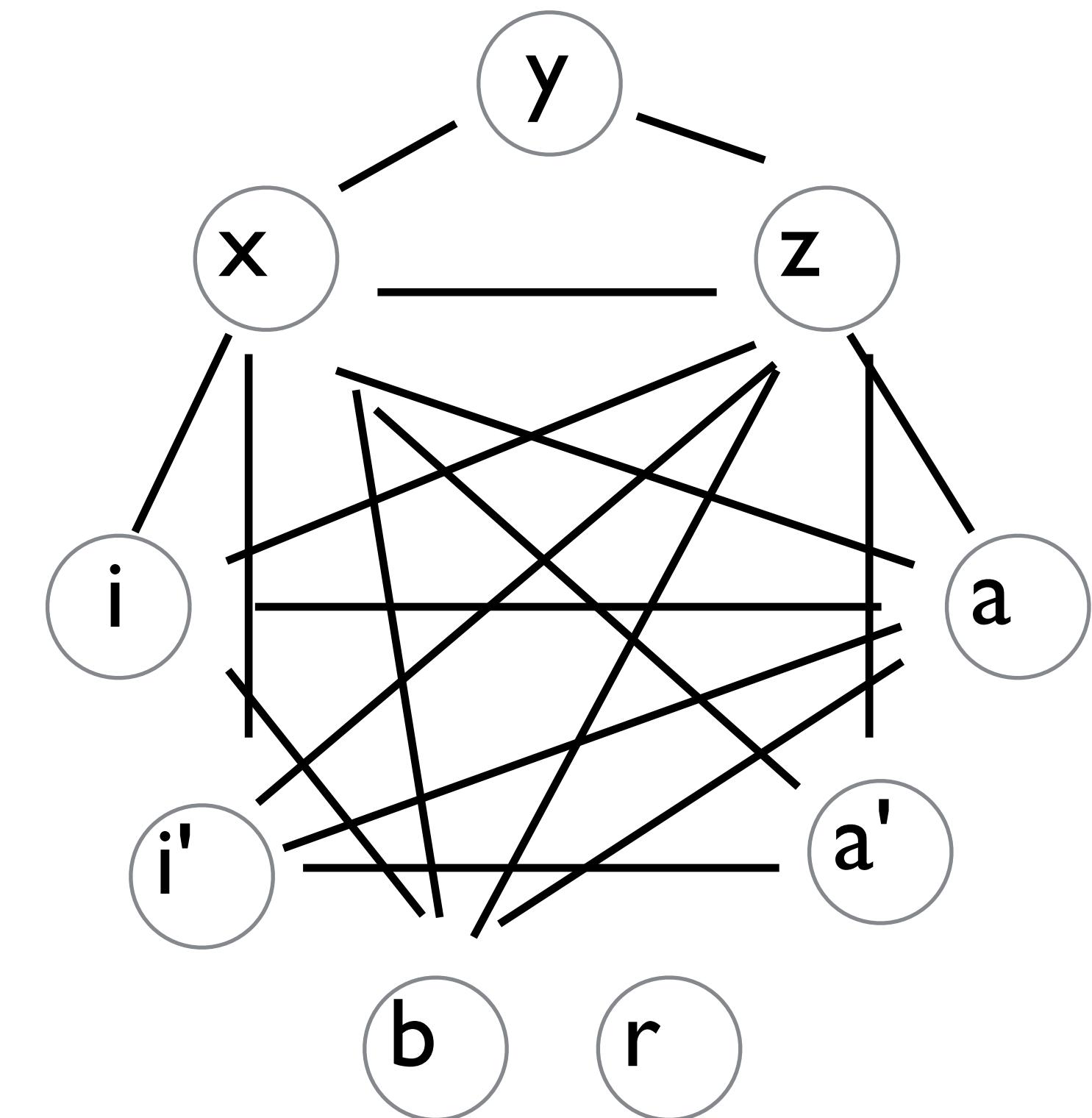
a variable x dominates y if x is in scope when y is defined
(includes simultaneous binding)

- 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)
```

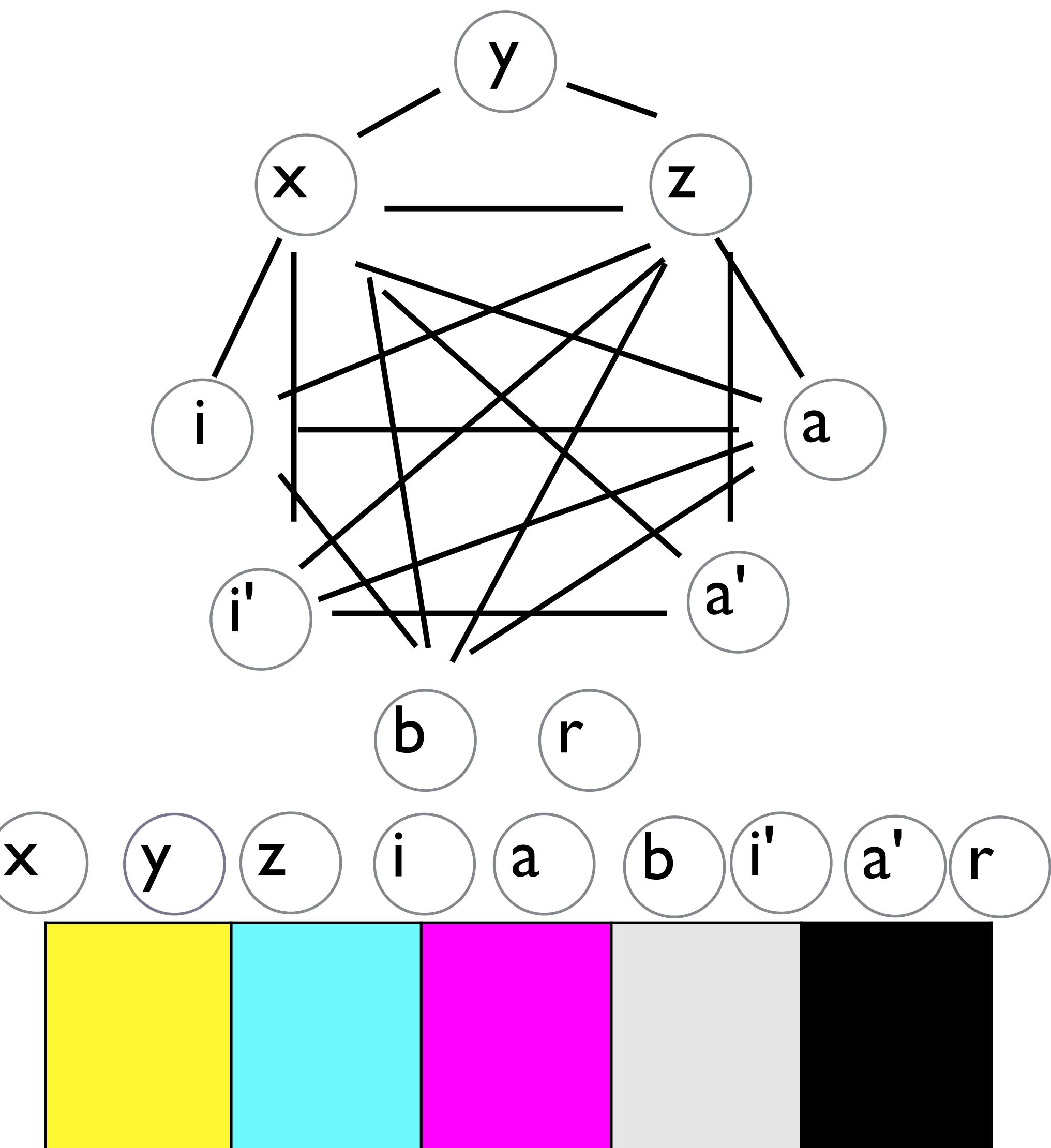
Interference Graph



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)
```

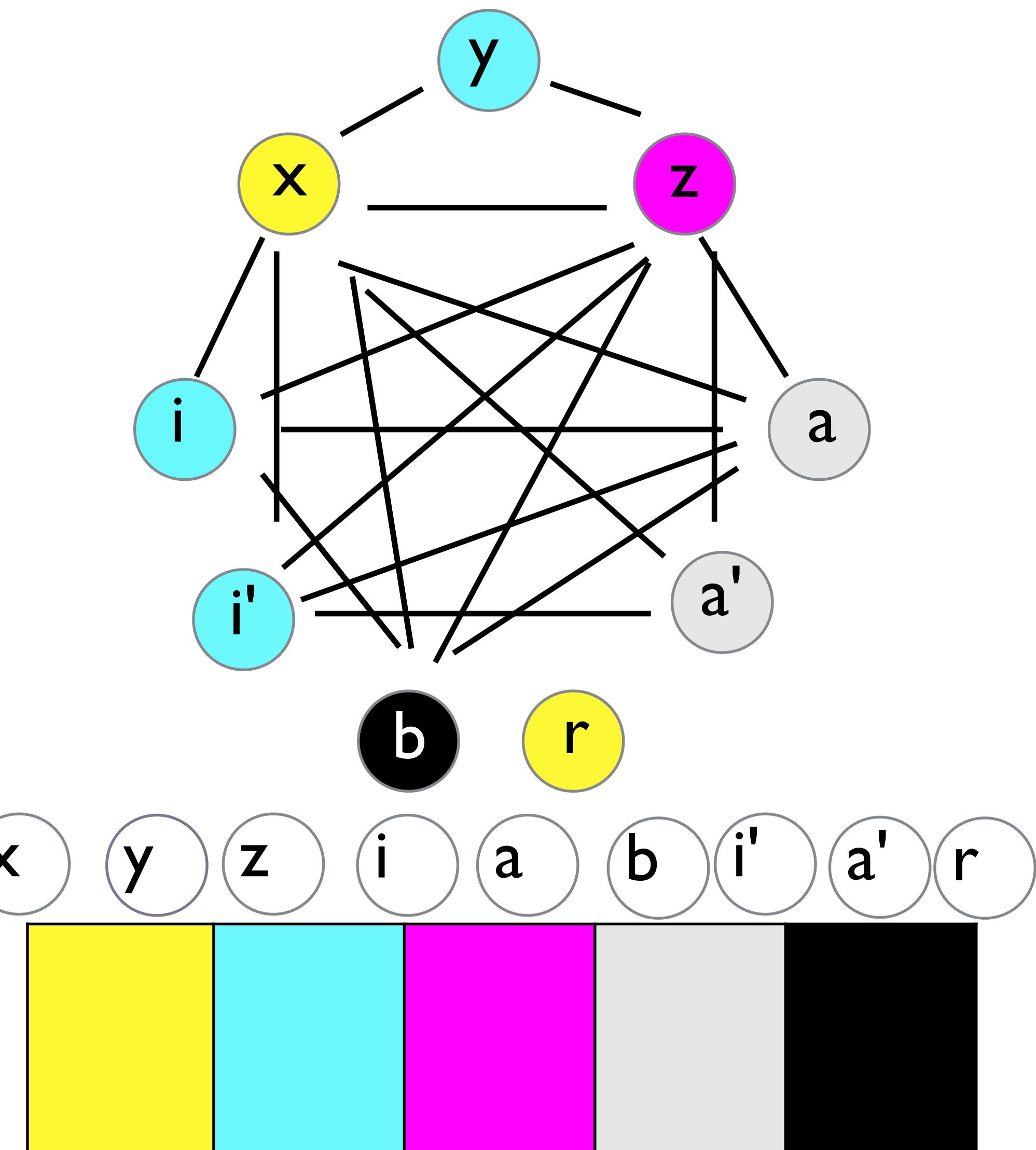
Interference Graph



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)]`

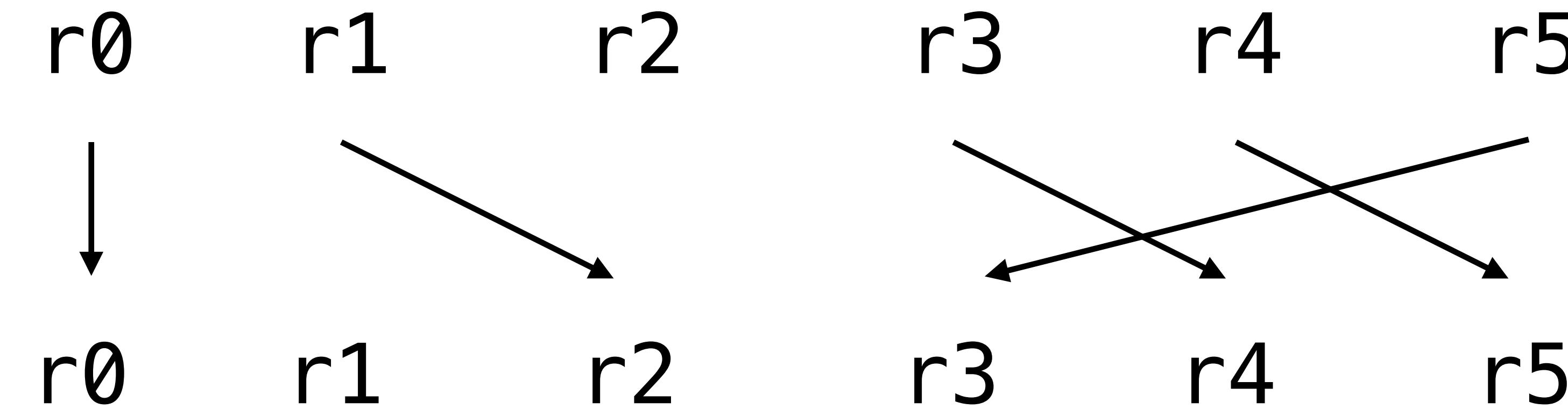
`mov [rsp - off(x)], r`

Implementing Branch with Arguments

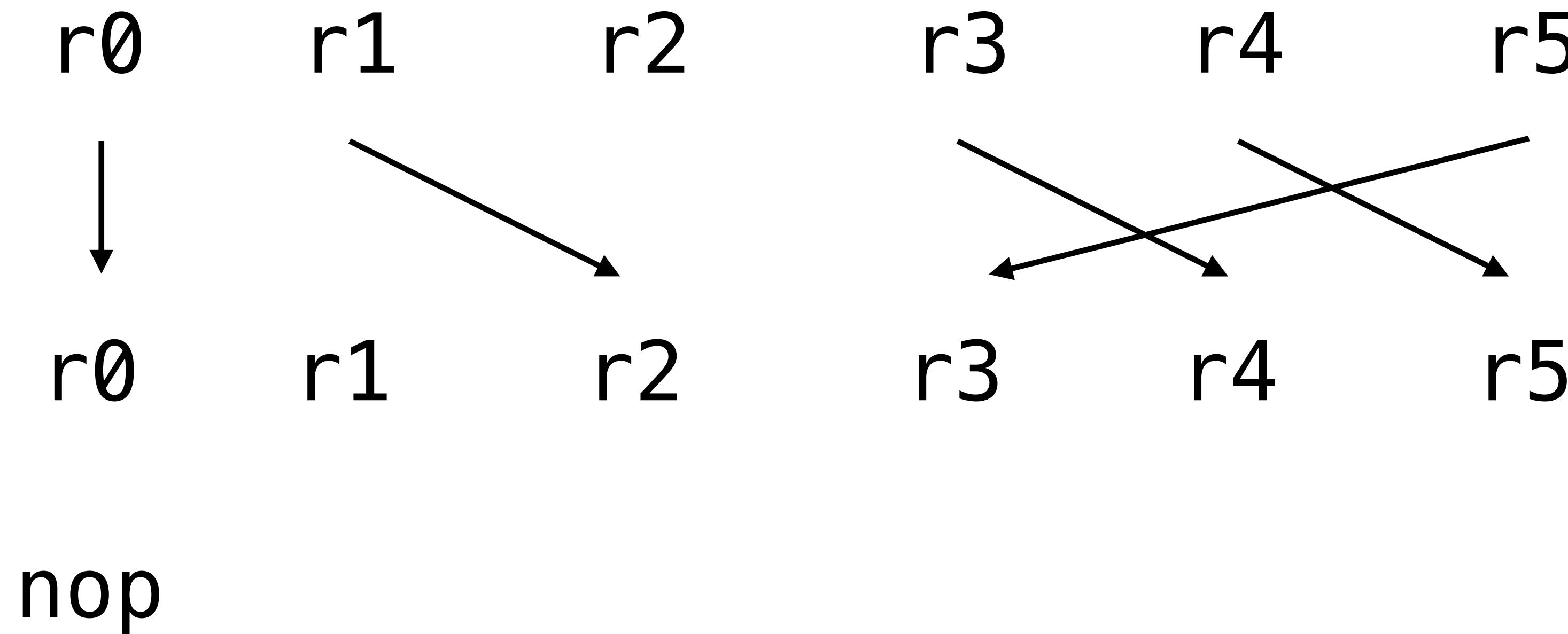
```
f(a,b,c): ...
...
br f(x,y,z)
               mov r_x, r_a
               mov r_y, r_b
               mov r_z, r_c
               jmp f
```

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

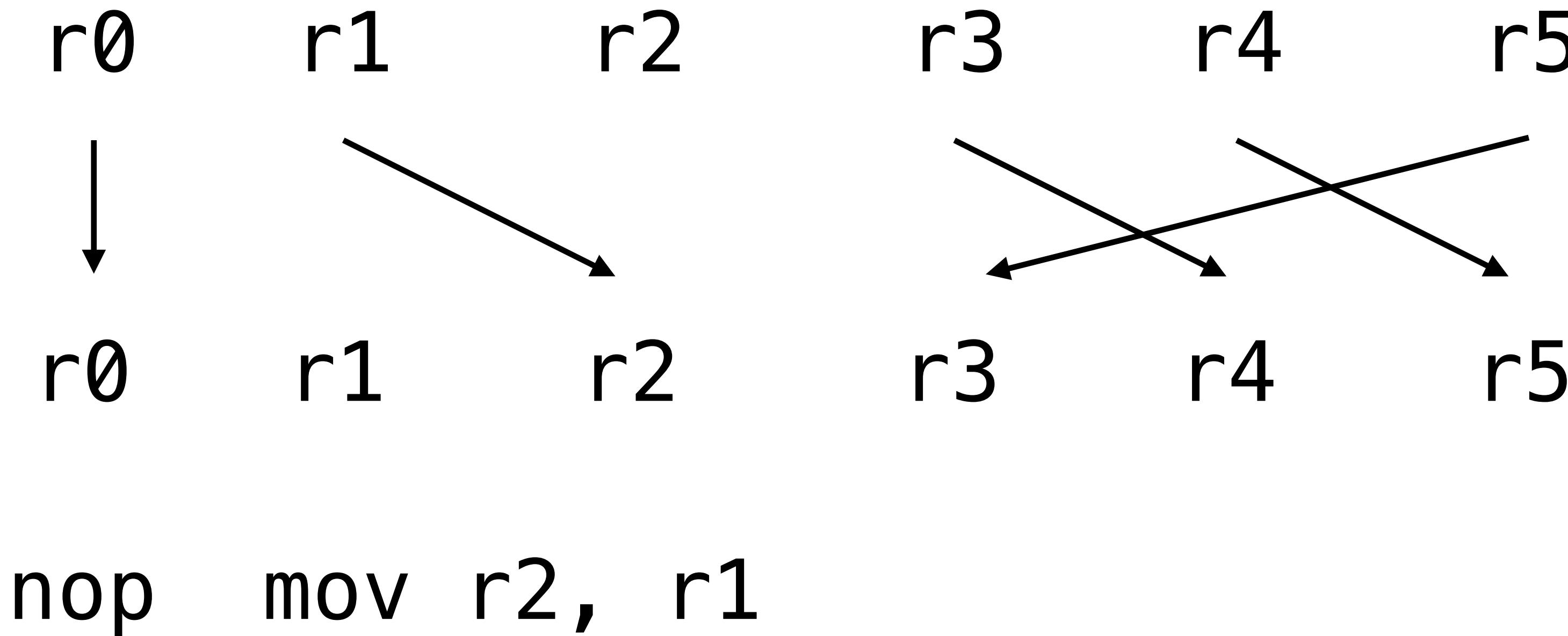
Implementing Branch with Arguments



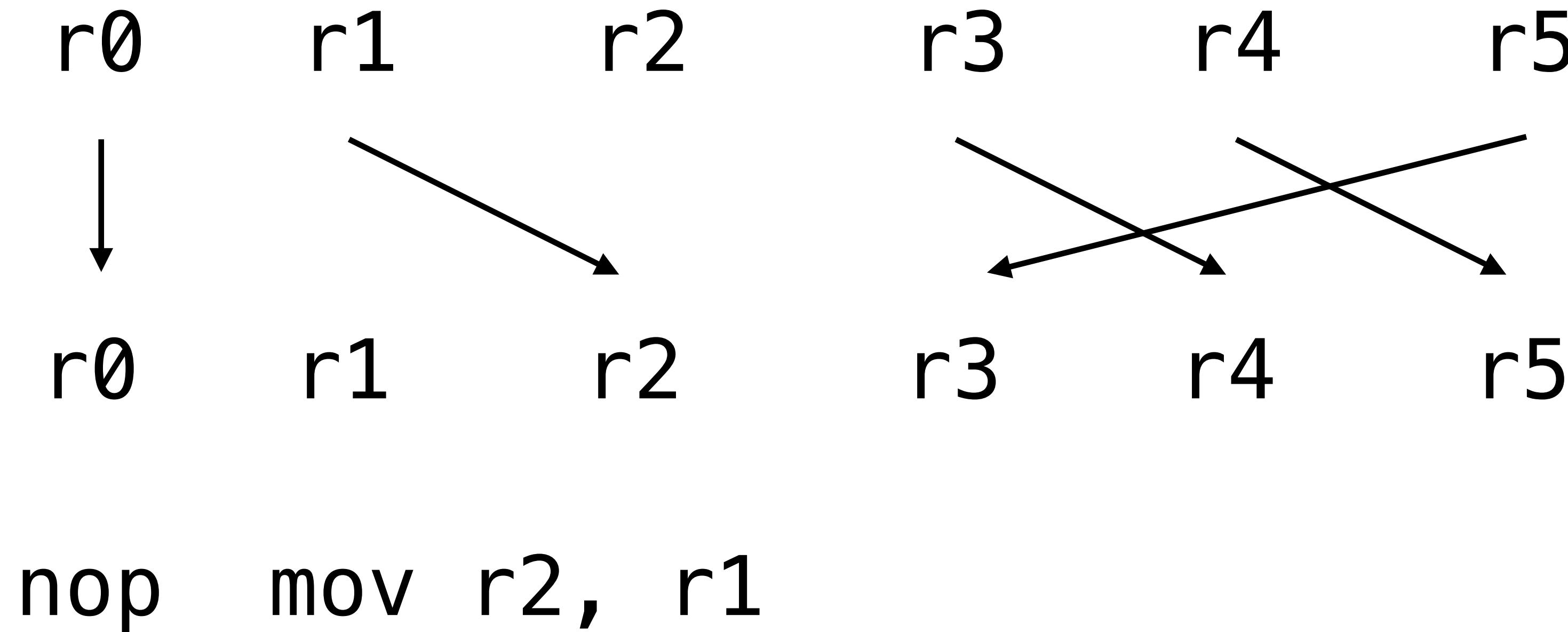
Implementing Branch with Arguments



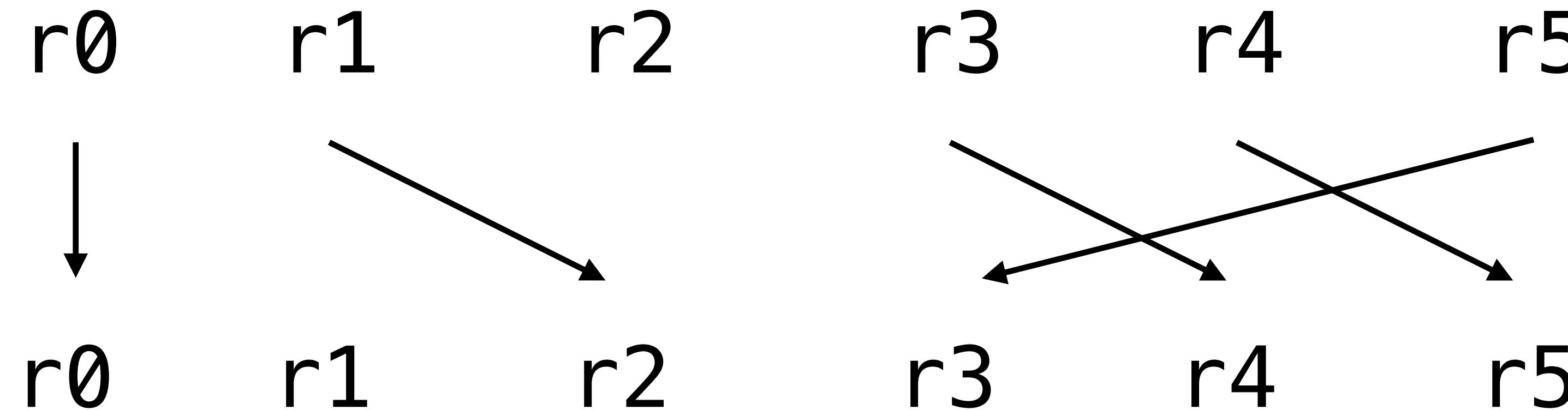
Implementing Branch with Arguments



Implementing Branch with Arguments



Implementing Branch with Arguments



nop mov r2, r1 xchg r3, r4
 xchg r3, r5

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 = ...

y = f(z)

z = x + y

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

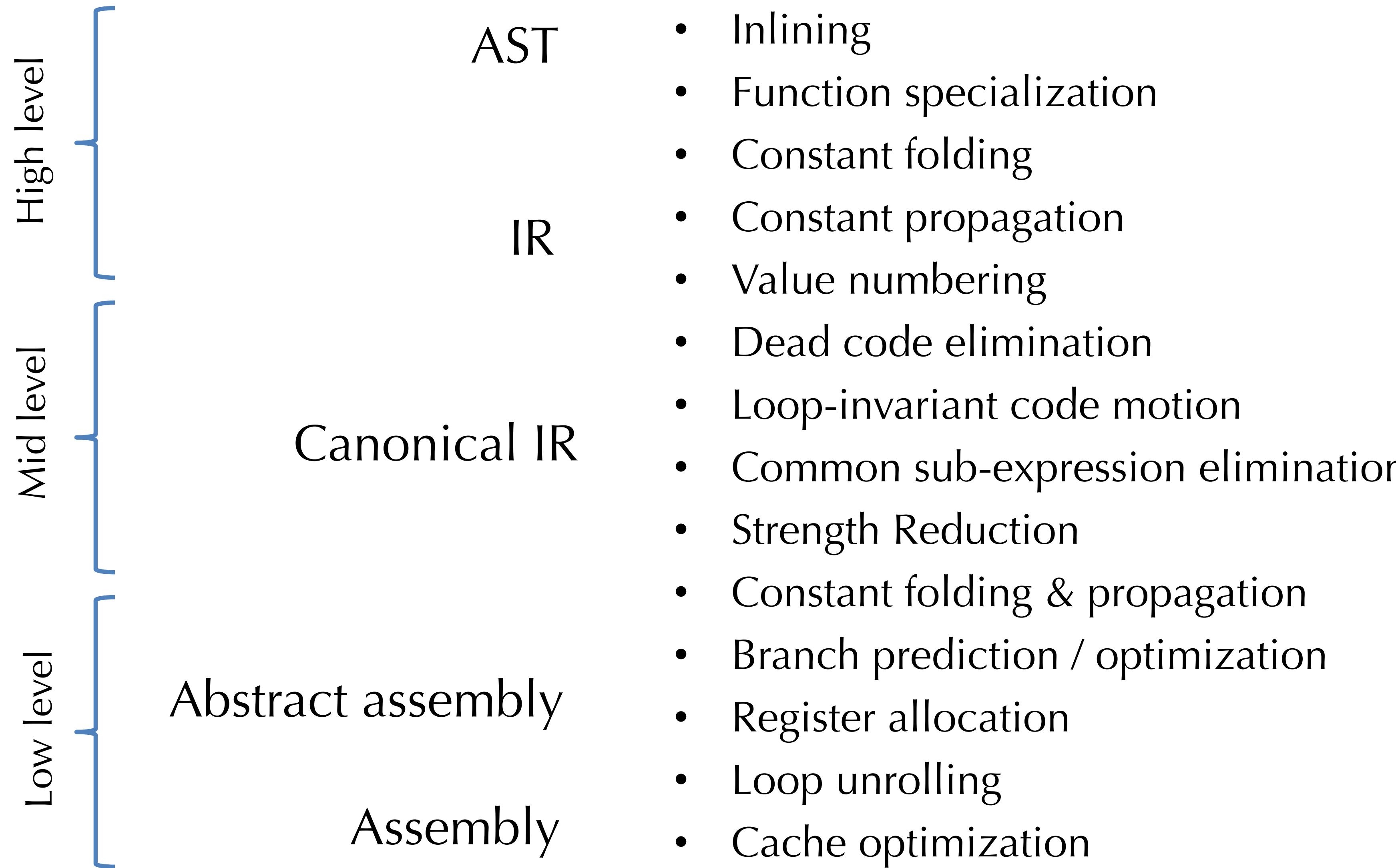
- Simple solution: save all live volatiles to the stack before a call, restore after the call
- Better solution: add nodes to interference graph for volatile registers, add conflicts at every non-tail call



Why optimize?

OPTIMIZATIONS, GENERALLY

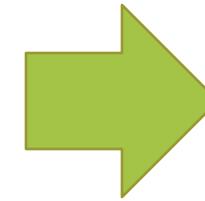
When to apply optimization



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 type, 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]`

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 = 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...

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;
}
```



note: renaming

```
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;
}
```

- 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

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!

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 $2^{(2^{64})}$ bits to represent!

Abstract Interpretation

- To keep space manageable, we need a different representation of sets, one that takes much less space than $2^{(2^{64})}$ 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 ?

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

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

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

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

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

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

- 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

$x = y \ll n \text{ where } n \geq 1$

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

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

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

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

- 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 ?

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

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

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

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

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

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

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)`

$$\text{Poss}(x) = \text{Flow}[\text{assertInt}](\text{Poss}(x))$$

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

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

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

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):
    Update the rest of the internal nodes

    loop(i,a):
        Current Round

        thn():
            3 ret a
            0: {y:Any}

        els():
            4 assertInt(i)
            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}

            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)

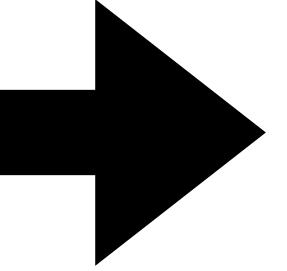
```

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

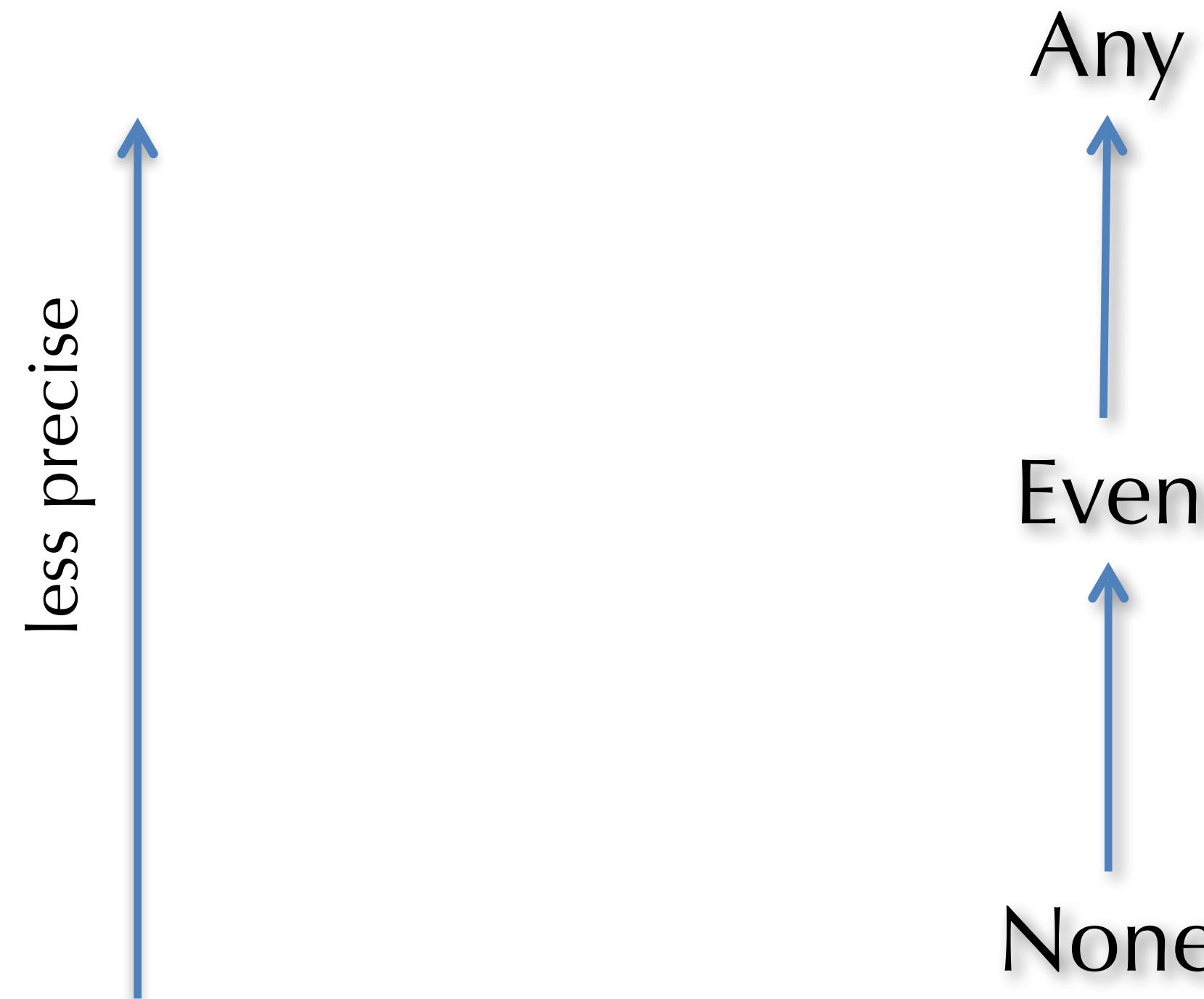


```
extern g
def main(y):
    def loop(i,a):
        if i == 0:
            a
        else:
            loop(i - 1, a + g())
    in
    if y == 0:
        0
    else:
        loop(y - 1, 0 + g())
```

If we re-do the analysis, now 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: the represents the empty set



GENERAL DATAFLOW ANALYSIS

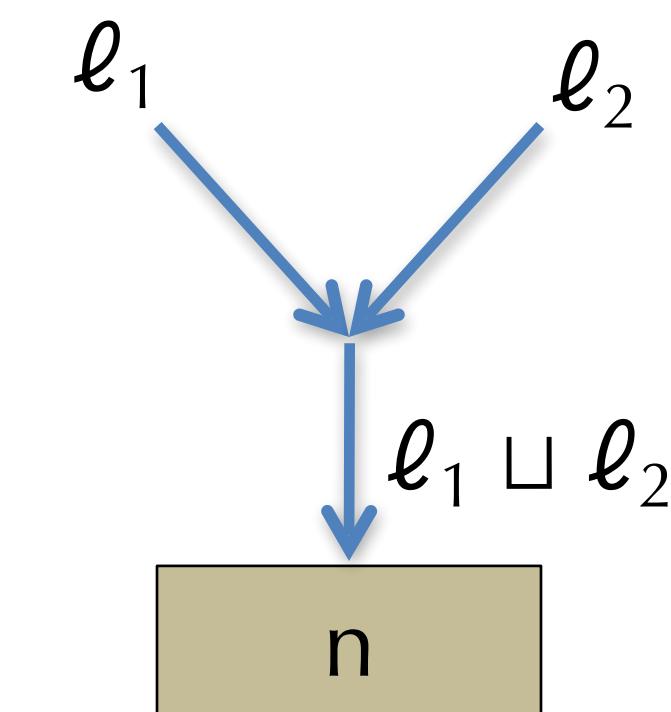
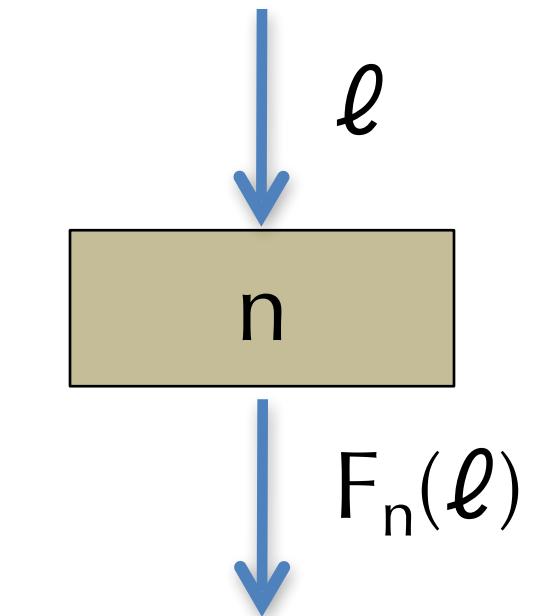
Common Features

- 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. $\mathcal{L} = \text{the powerset of all variables}$
 - 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 ℓ 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 ℓ_1 or ℓ_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

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

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

 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}

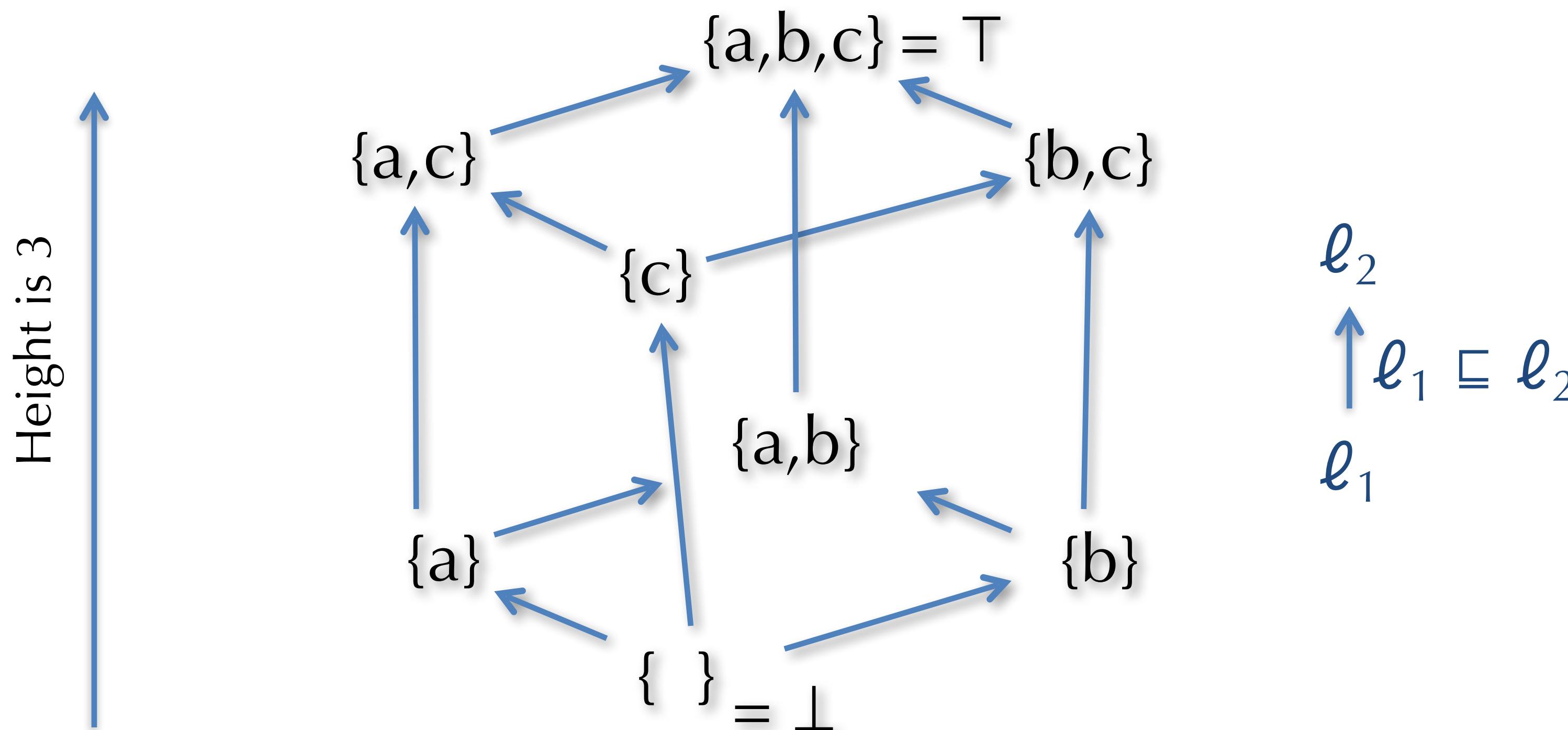
- The domain has structure that reflects the “amount” of information contained in each dataflow value.
- Some dataflow values are more informative than others:
 - Write $\ell_1 \sqsubseteq \ell_2$ whenever ℓ_1 provides at least as much information as ℓ_2 .
 - The dataflow value ℓ_1 is “better” for enabling 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.



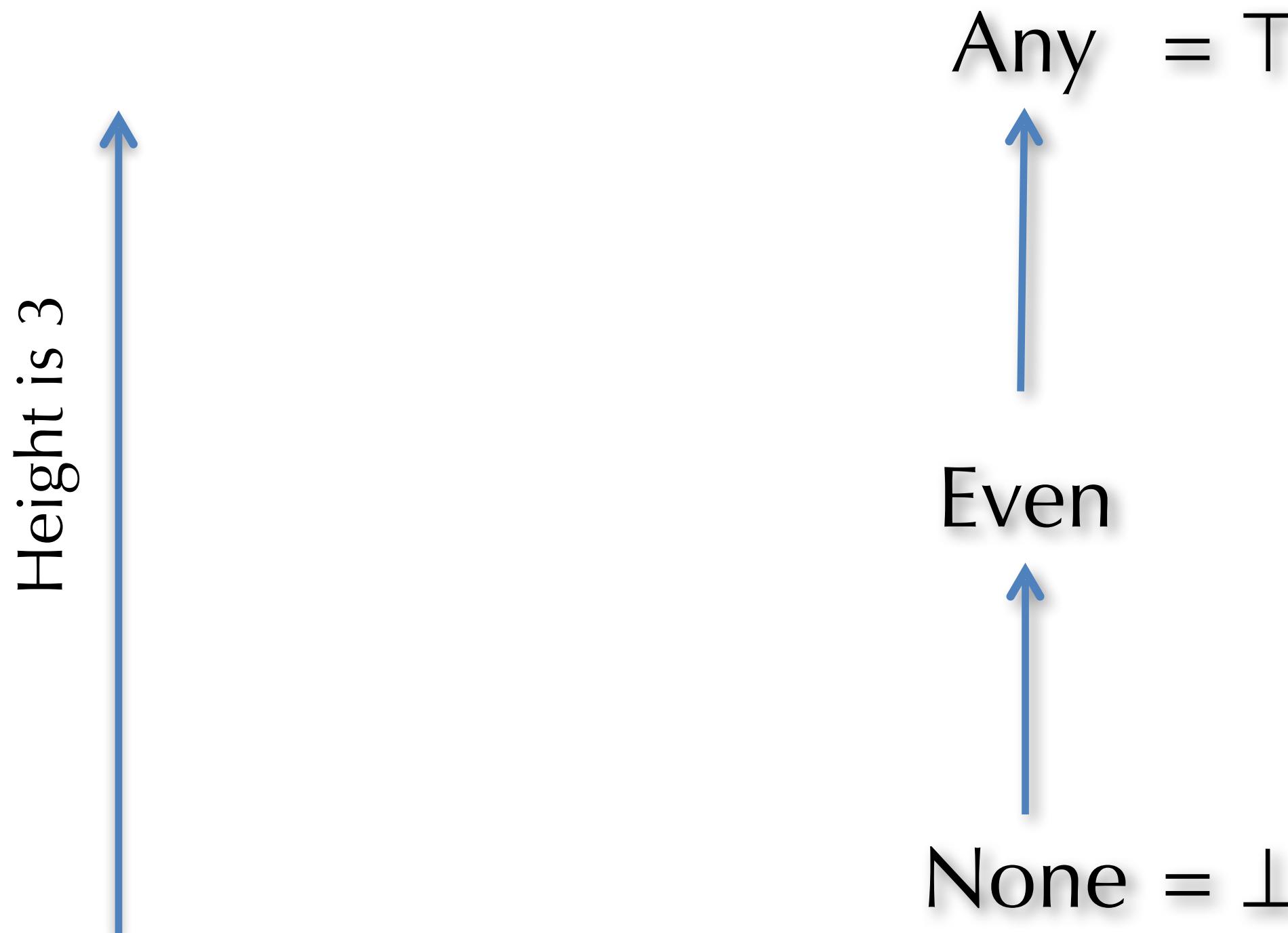
order \sqsubseteq is \subseteq

meet \sqcap is \sqcap

join \sqcup is \sqcup

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 combinig 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:
 - $\text{let } x_n = \text{out}[n]$
 - 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 \mathbf{F} to the old vector:
$$\mathbf{X}_1 = \mathbf{F}(\mathbf{X}_0)$$
$$\mathbf{X}_2 = \mathbf{F}(\mathbf{X}_1)$$

$$\dots$$
- $\mathbf{F}^{k+1}(\mathbf{X}) = \mathbf{F}(\mathbf{F}^k(\mathbf{X}))$
- 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

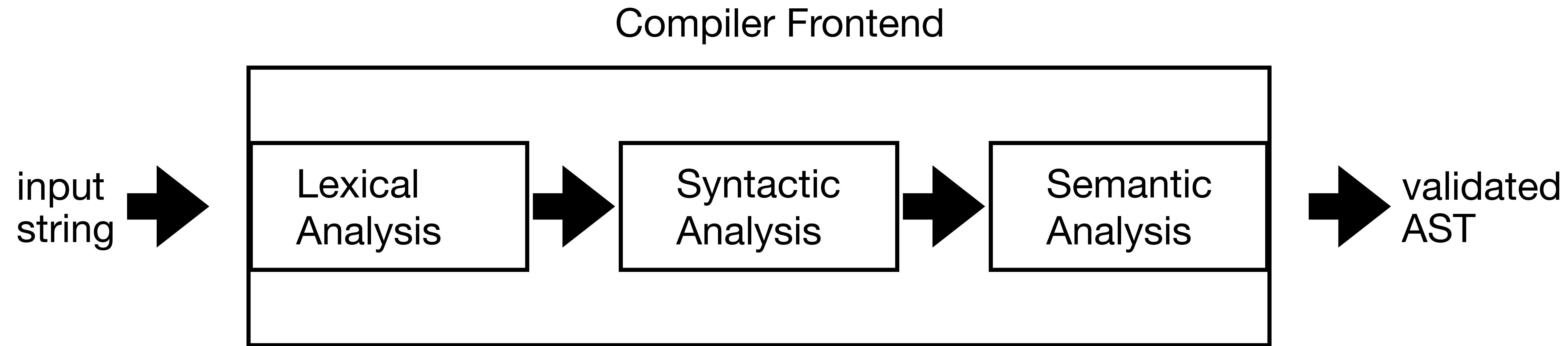
- Each flow function F_n maps lattice elements to lattice elements; to be sensible is should be *monotonic*:
- $F : \mathcal{L} \rightarrow \mathcal{L}$ is *monotonic* iff:
 $\ell_1 \sqsubseteq \ell_2$ implies that $F(\ell_1) \sqsubseteq F(\ell_2)$
 - Intuitively: “If you have more information entering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

Frontend

Compiler Frontends

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

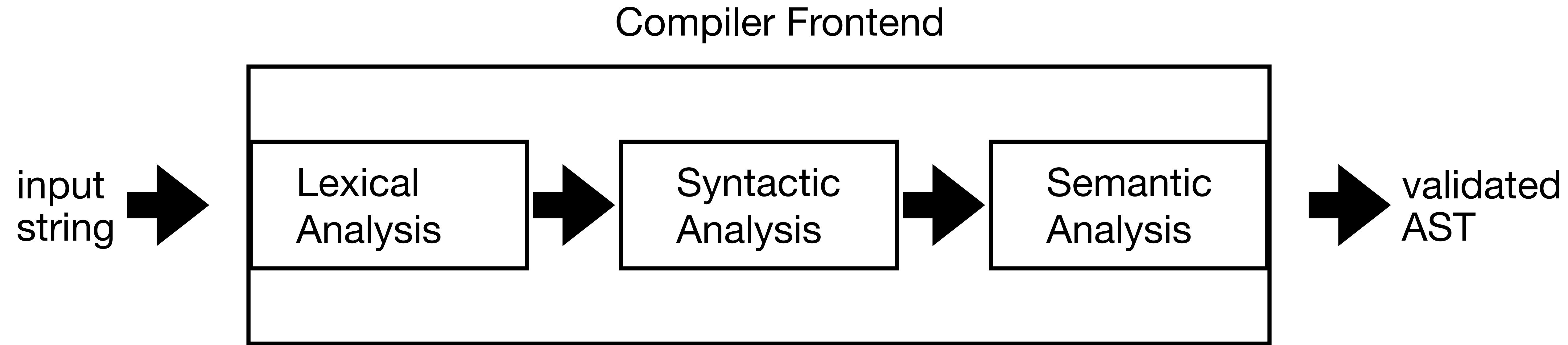
1. Validate that it is a well-formed program
2. Output an **Abstract Syntax Tree** that is more convenient for the rest of the compiler pipeline to use



Compiler 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 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

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

Making Lexing Less Painful

- 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
 - ocamllex
 - 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
 - What strings make up the "tokens" of our language
 - How to turn these abstract tokens into data that our compiler pipeline can use
- Need to make a language for describing sets of strings

Formal Languages

- First we fix the "alphabet" of characters Σ .
 - Common alphabets { 0 , 1 } for bitstrings
 - 0-255 for ASCII characters
 - very large set of Unicode "characters"
- A string (over Σ) is a finite sequence of characters (i.e., elements of Σ)
- 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:
 - ϵ Epsilon stands for the empty string
 - ‘ a ’ An ordinary character stands for itself
 - $R_1 \mid R_2$ Alternatives, stands for choice of R_1 or R_2
 - R_1R_2 Concatenation, stands for R_1 followed by R_2
 - R^* Kleene star, stands for zero or more repetitions of R
- *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 $(\epsilon \mid R)$
 - [‘ a ’ – ‘ z ’] One of a or b or c or ... z , equivalent to $(a \mid b \mid \dots \mid z)$
 - [^ ‘0’ – ‘9’] Any character except 0 through 9
 - R as x Name the string matched by R as x

Example Regular Expressions

- Recognize the keyword “if”: “if”
- Recognize a digit: ['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
```

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 `String -> 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

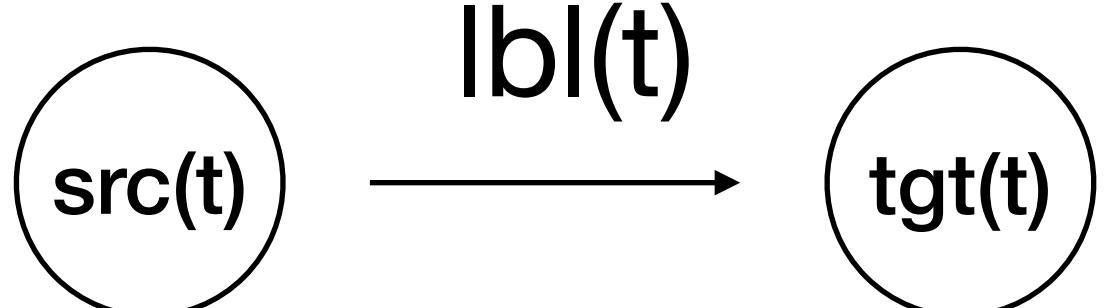
How can we efficiently implement a recognizer for a regular language?

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

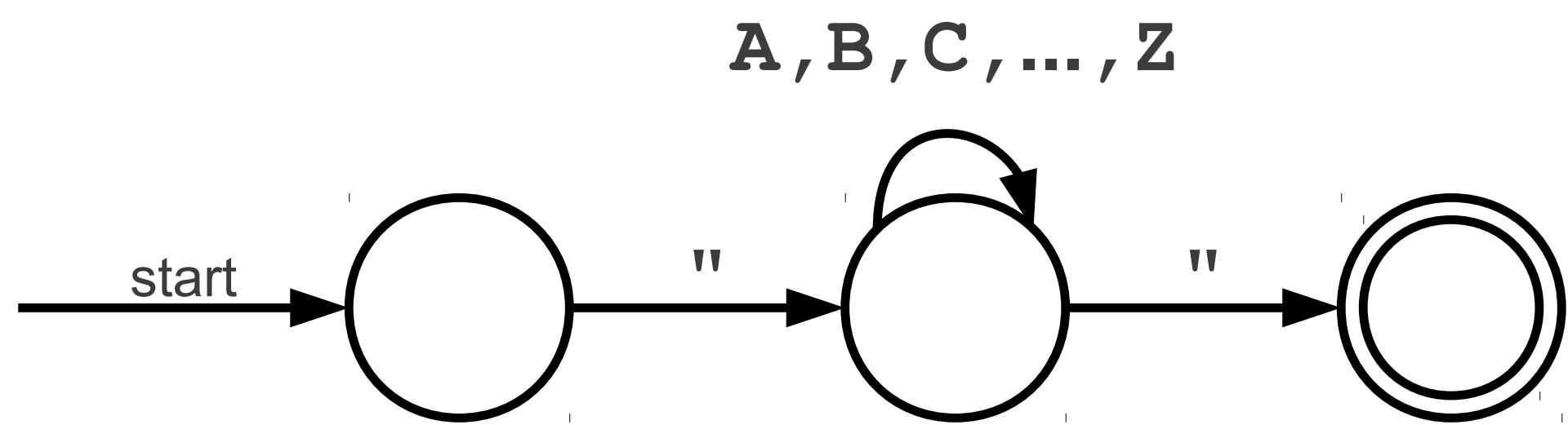
Finite State Automata

A (non-deterministic) finite state automaton over an alphabet Σ consists of

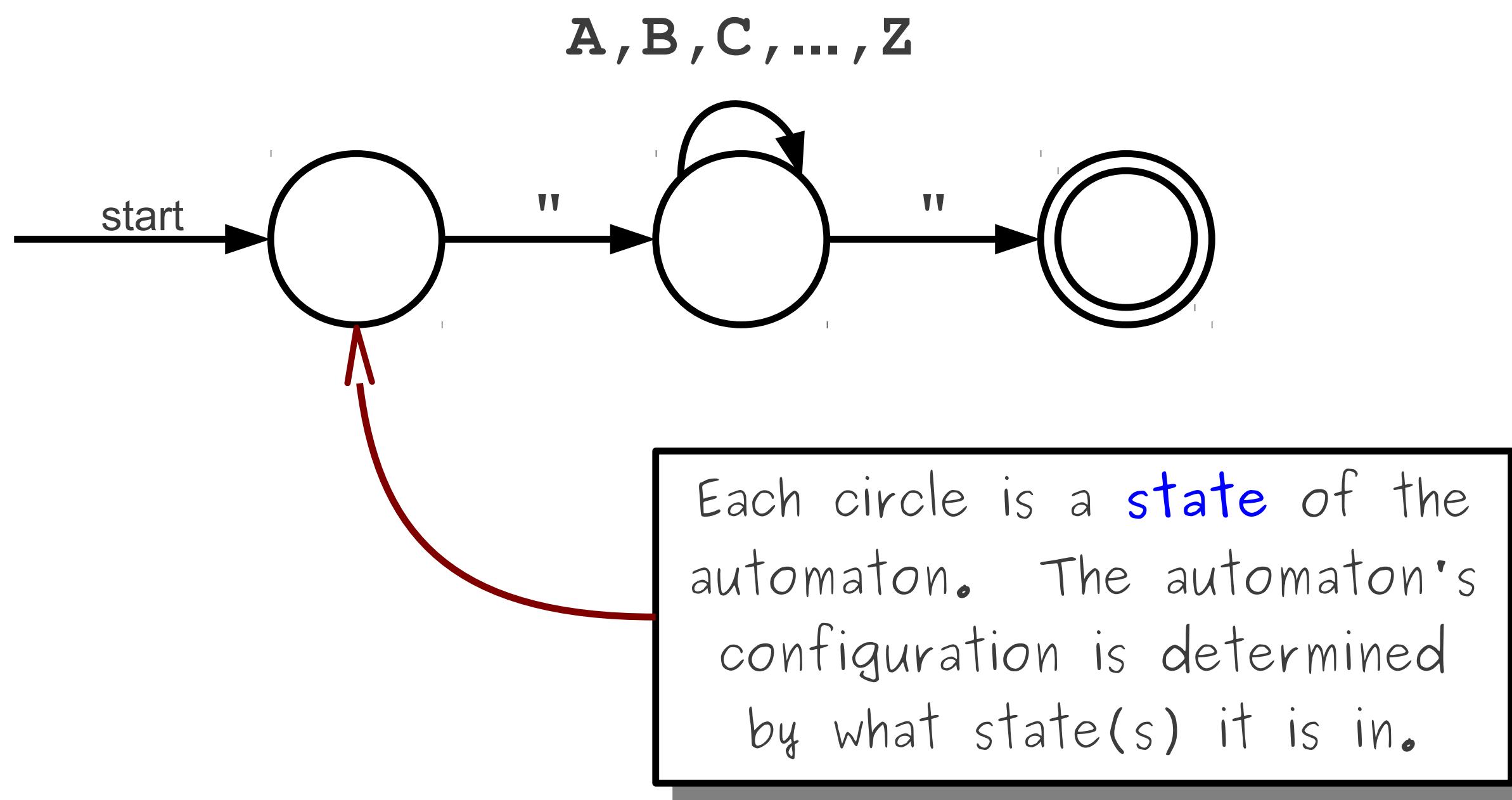
- A finite set of states \mathbf{S}
- A distinguished start state $s_0 \in \mathbf{S}$
- A subset of accepting states $\mathbf{Acc} \subseteq \mathbf{S}$
- A set of transitions δ , where each transition $t \in \delta$ has
 - a source state $\mathbf{src}(t)$
 - a target state $\mathbf{tgt}(t)$
 - a label $\mathbf{lbl}(t)$, which is either a character $c \in \Sigma$ or ϵ



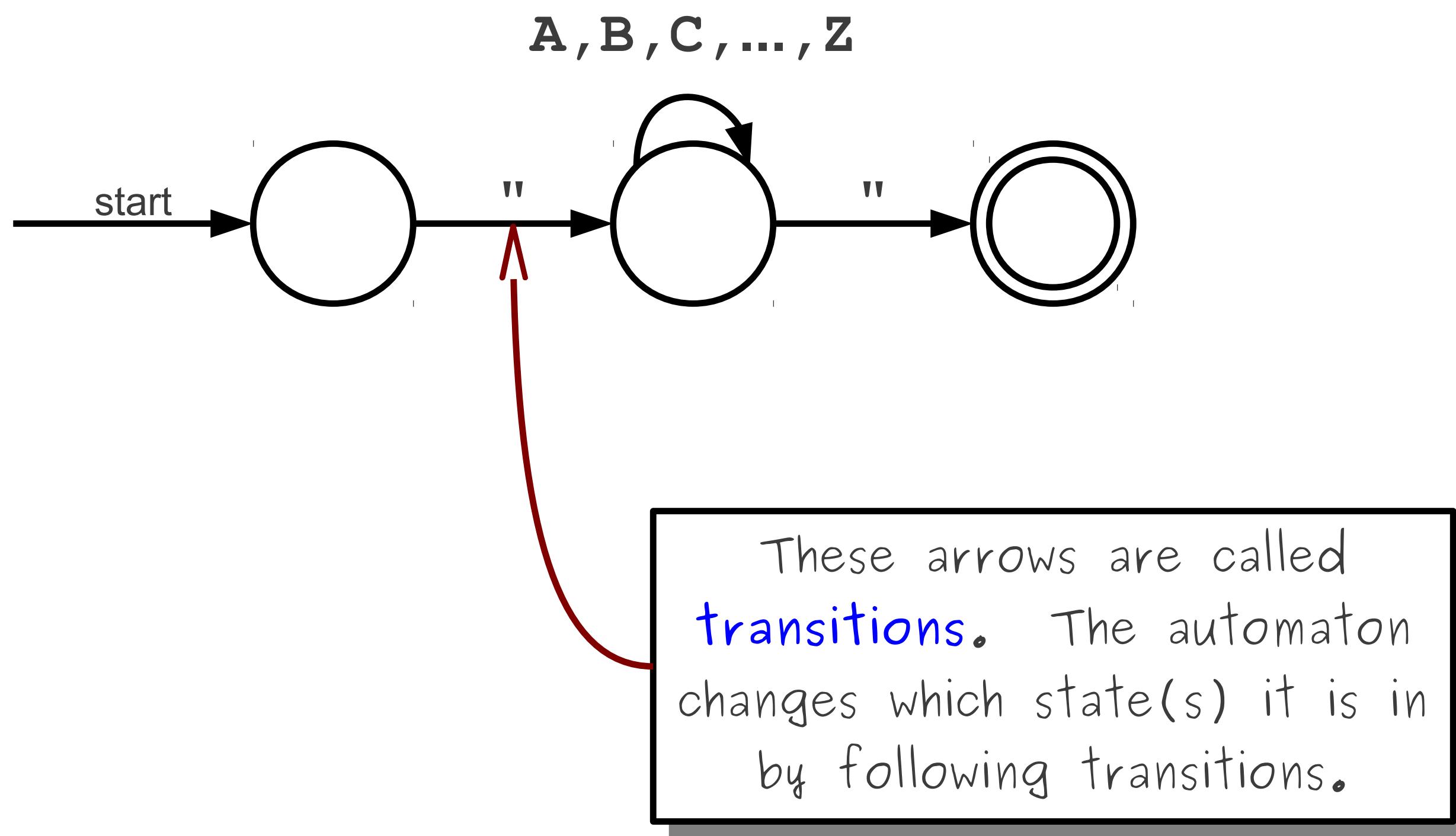
A Simple Automaton



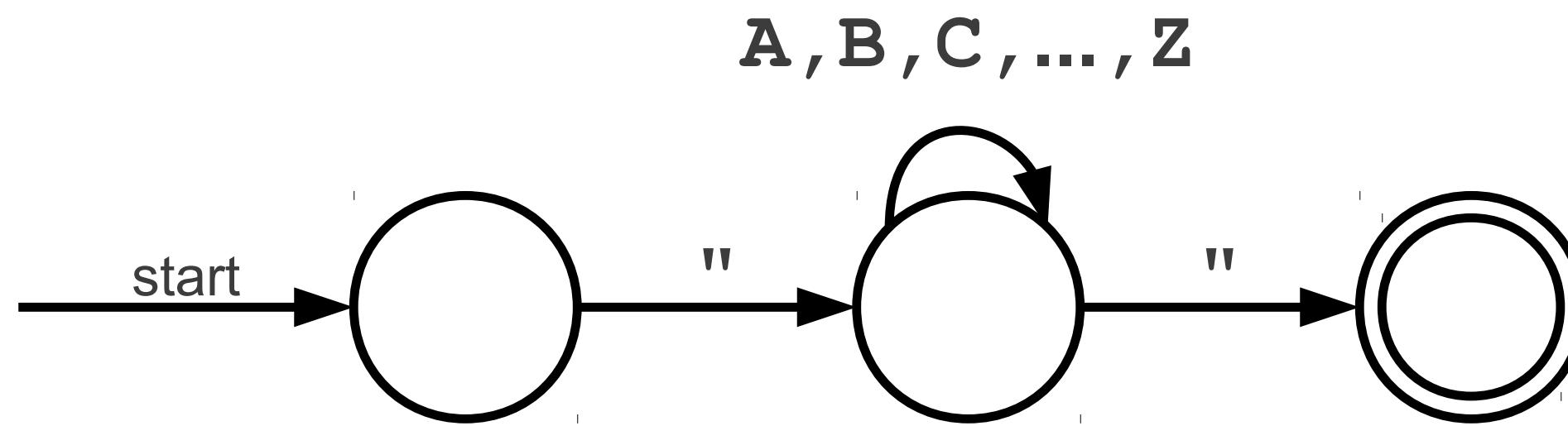
A Simple Automaton



A Simple Automaton



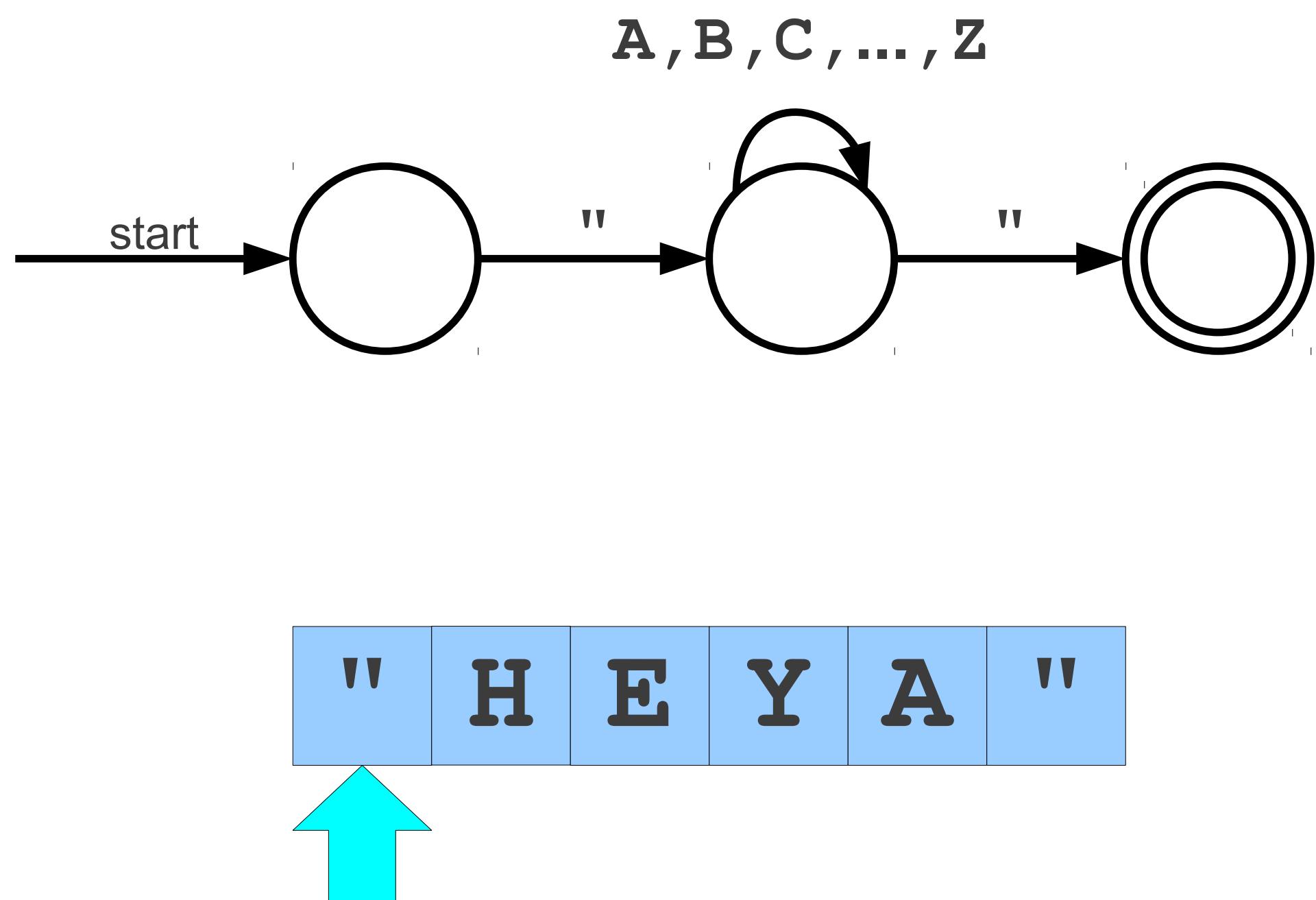
A Simple Automaton



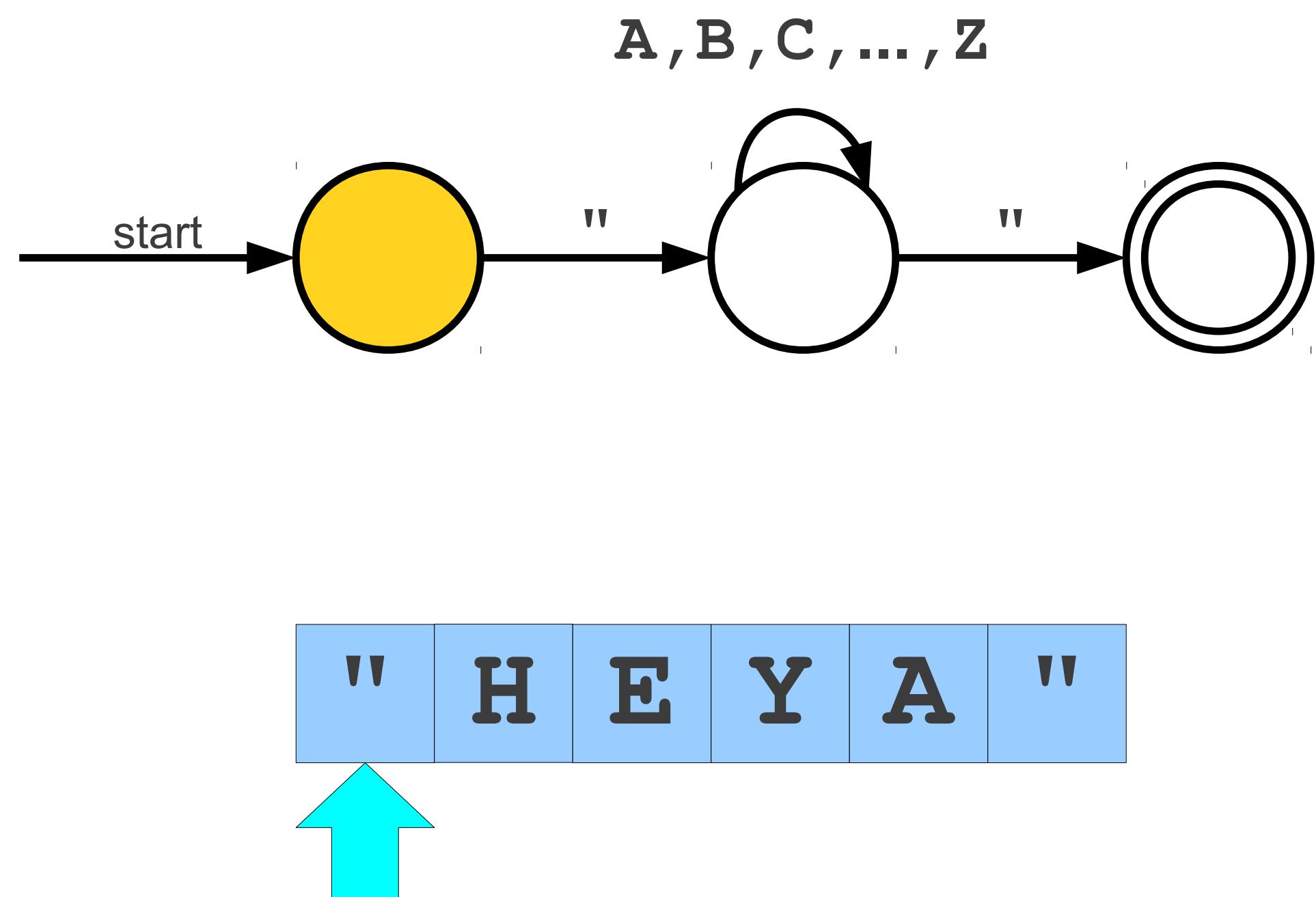
"	H	E	Y	A	"
---	---	---	---	---	---

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

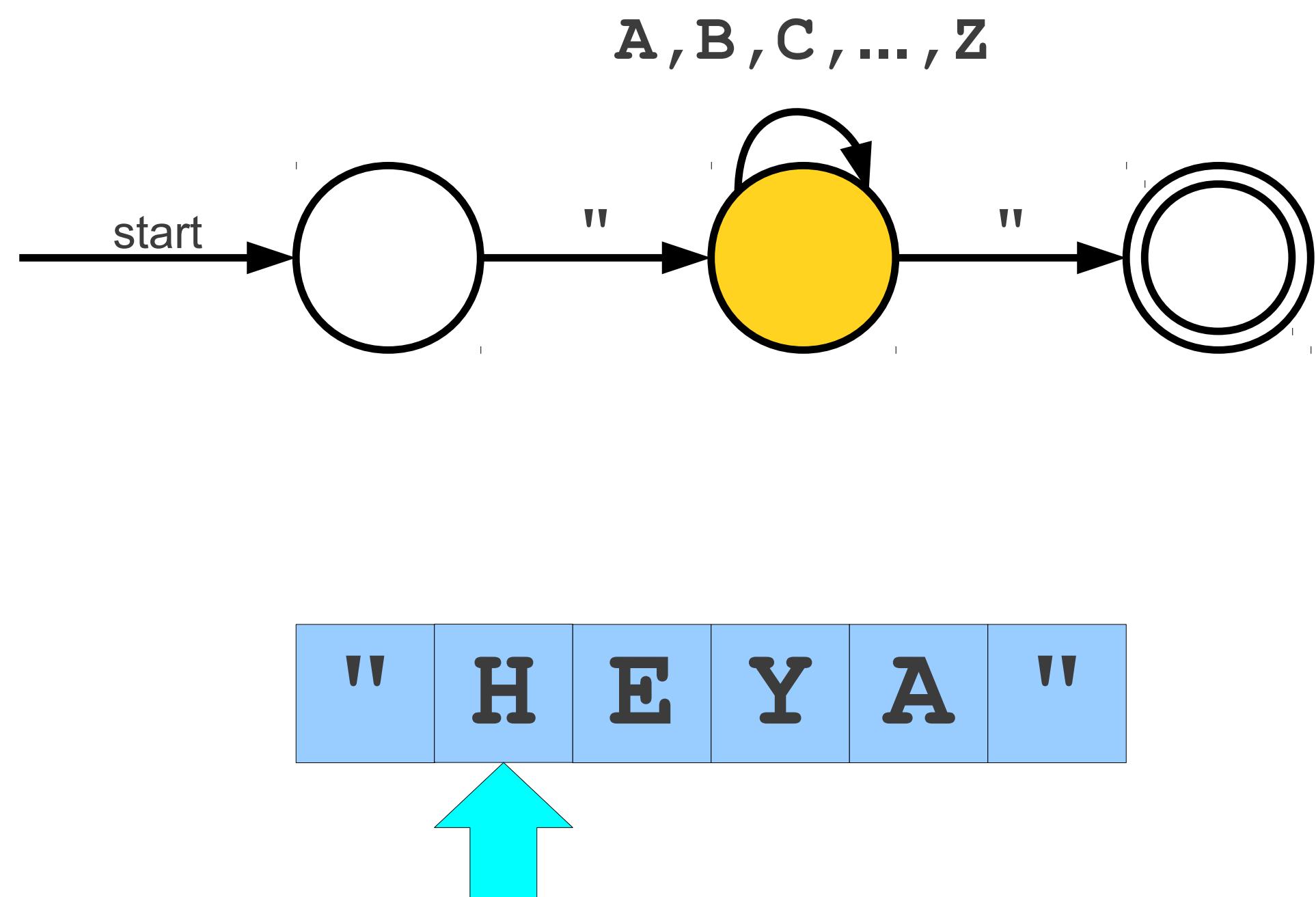
A Simple Automaton



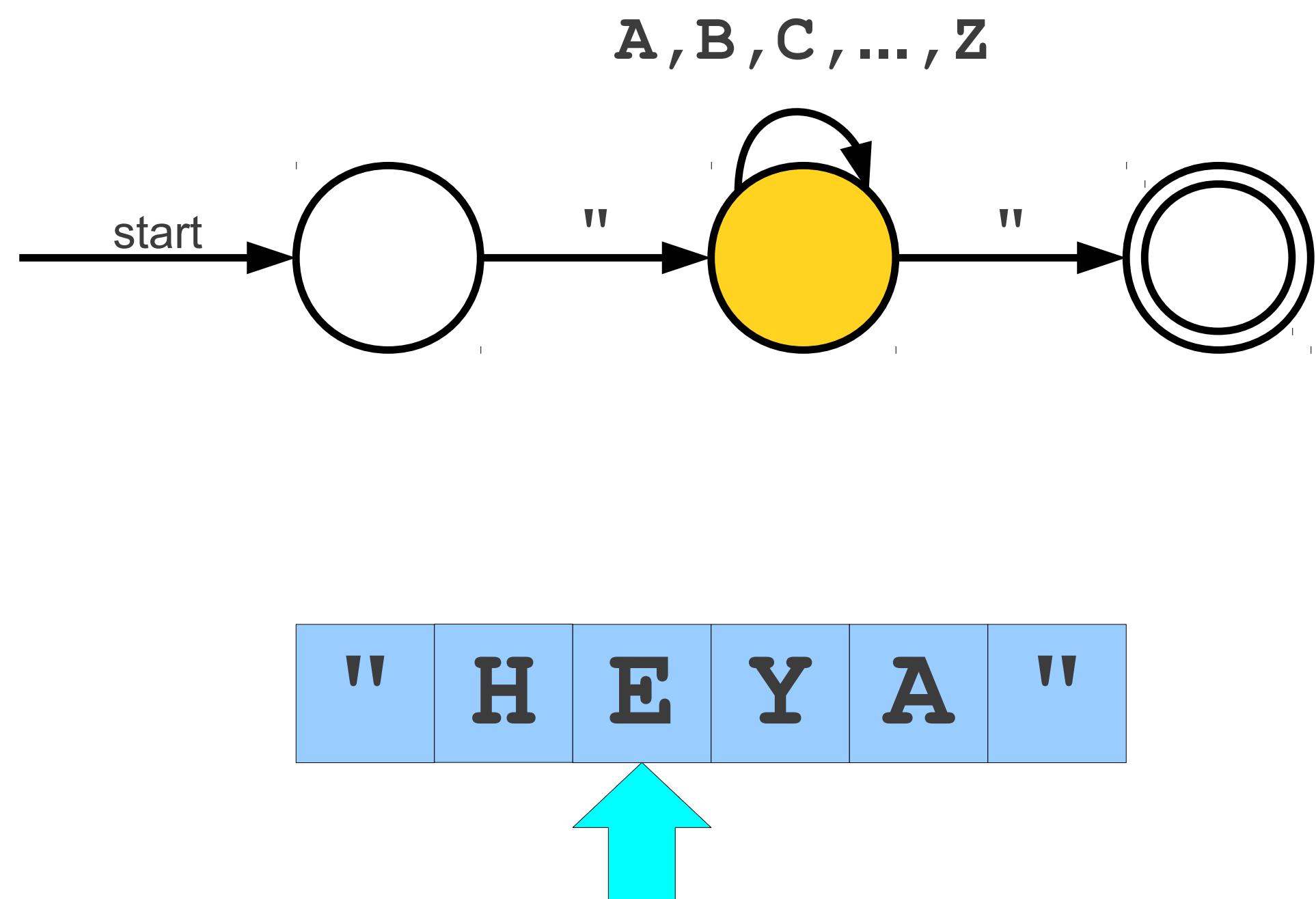
A Simple Automaton



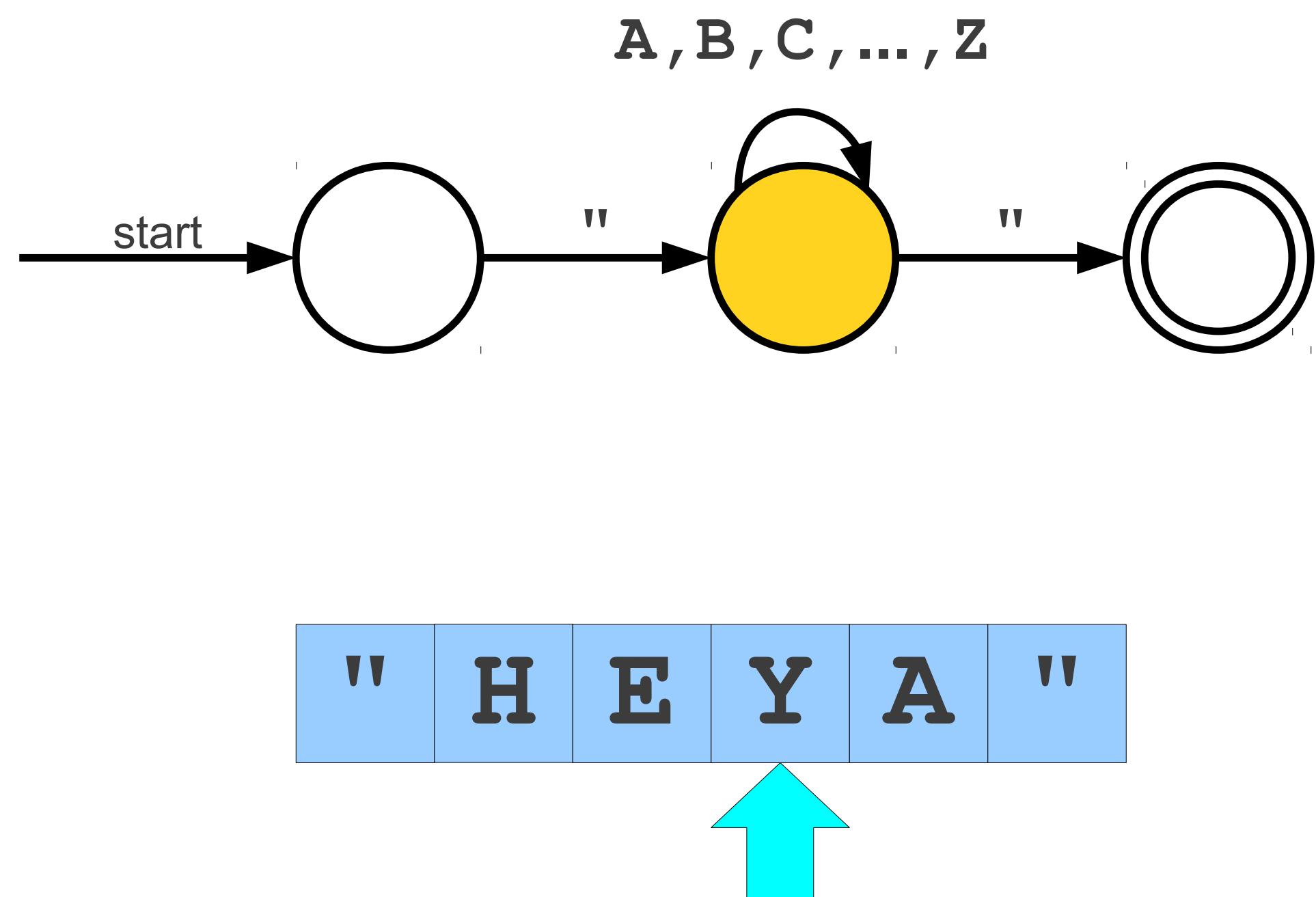
A Simple Automaton



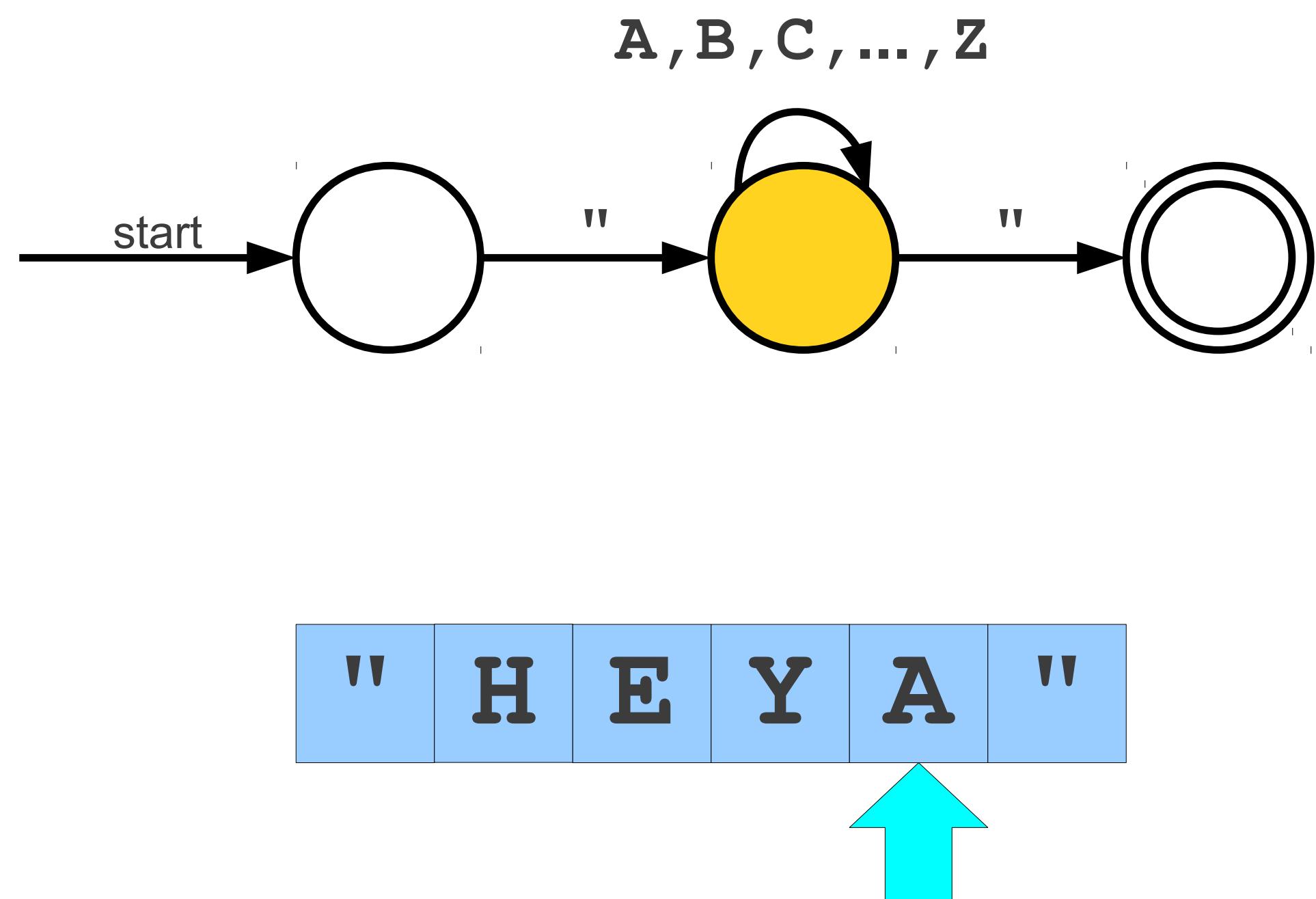
A Simple Automaton



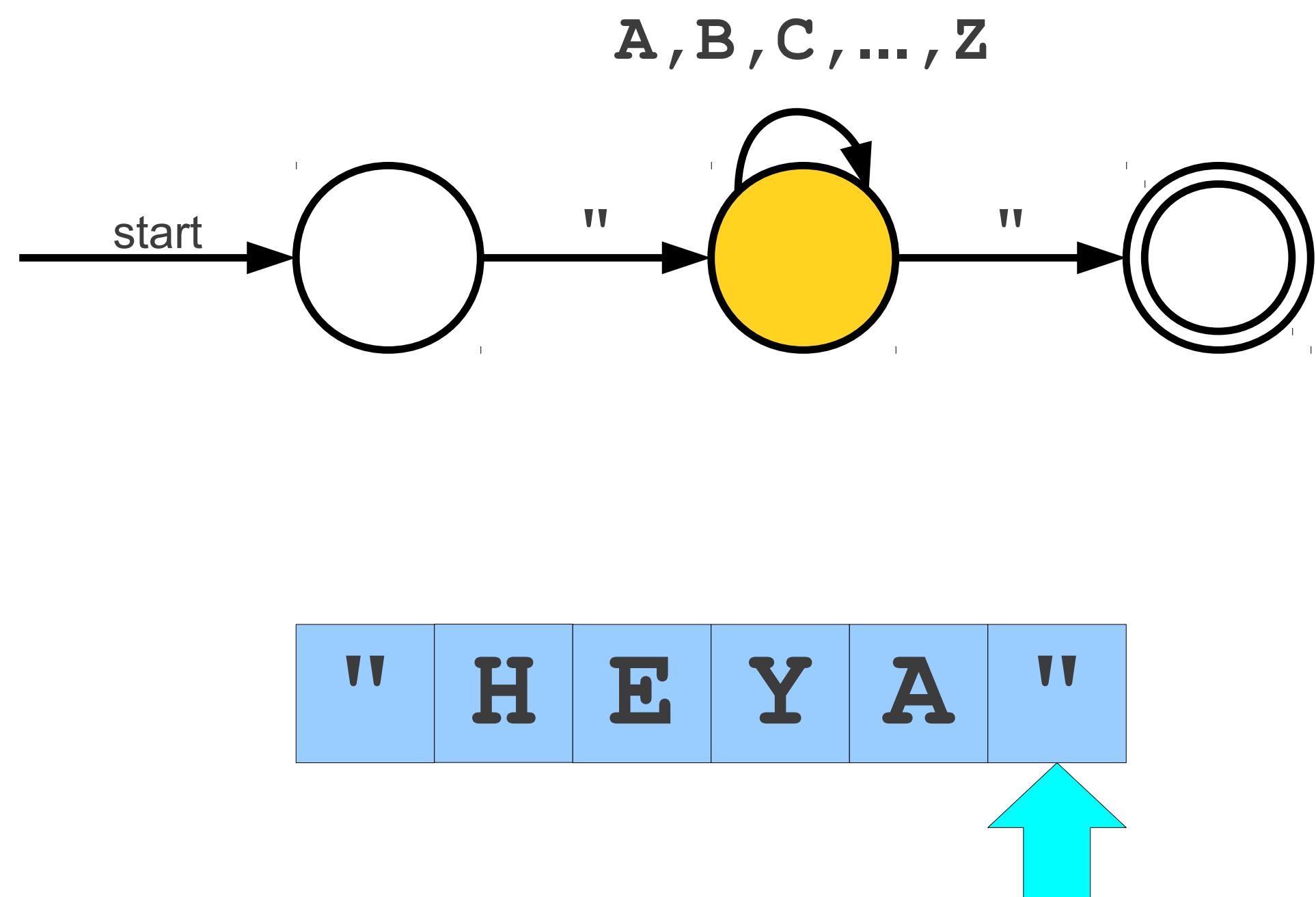
A Simple Automaton



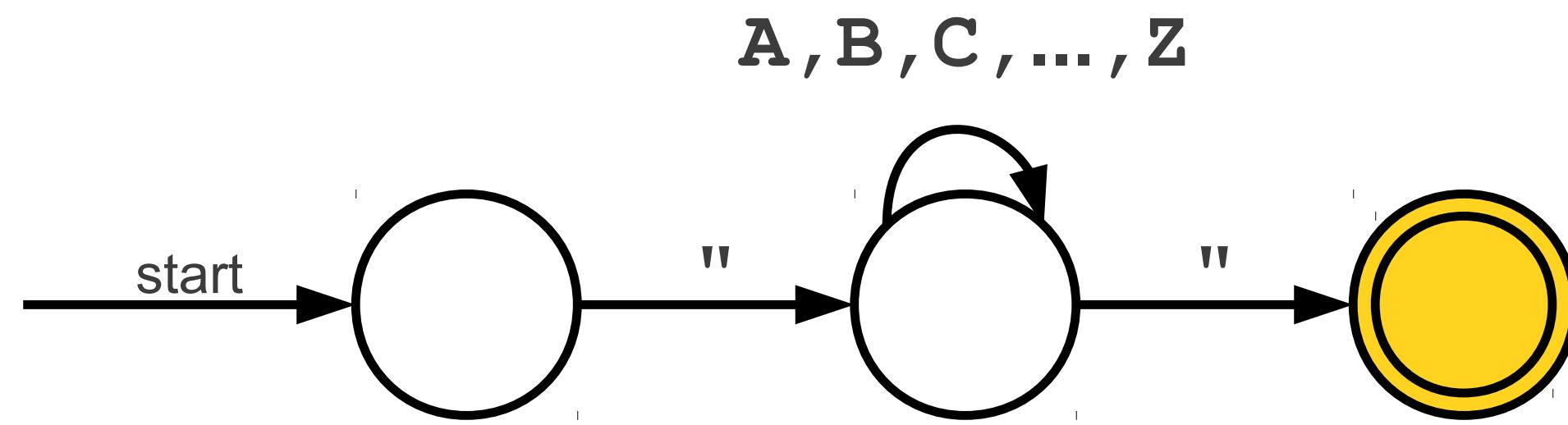
A Simple Automaton



A Simple Automaton

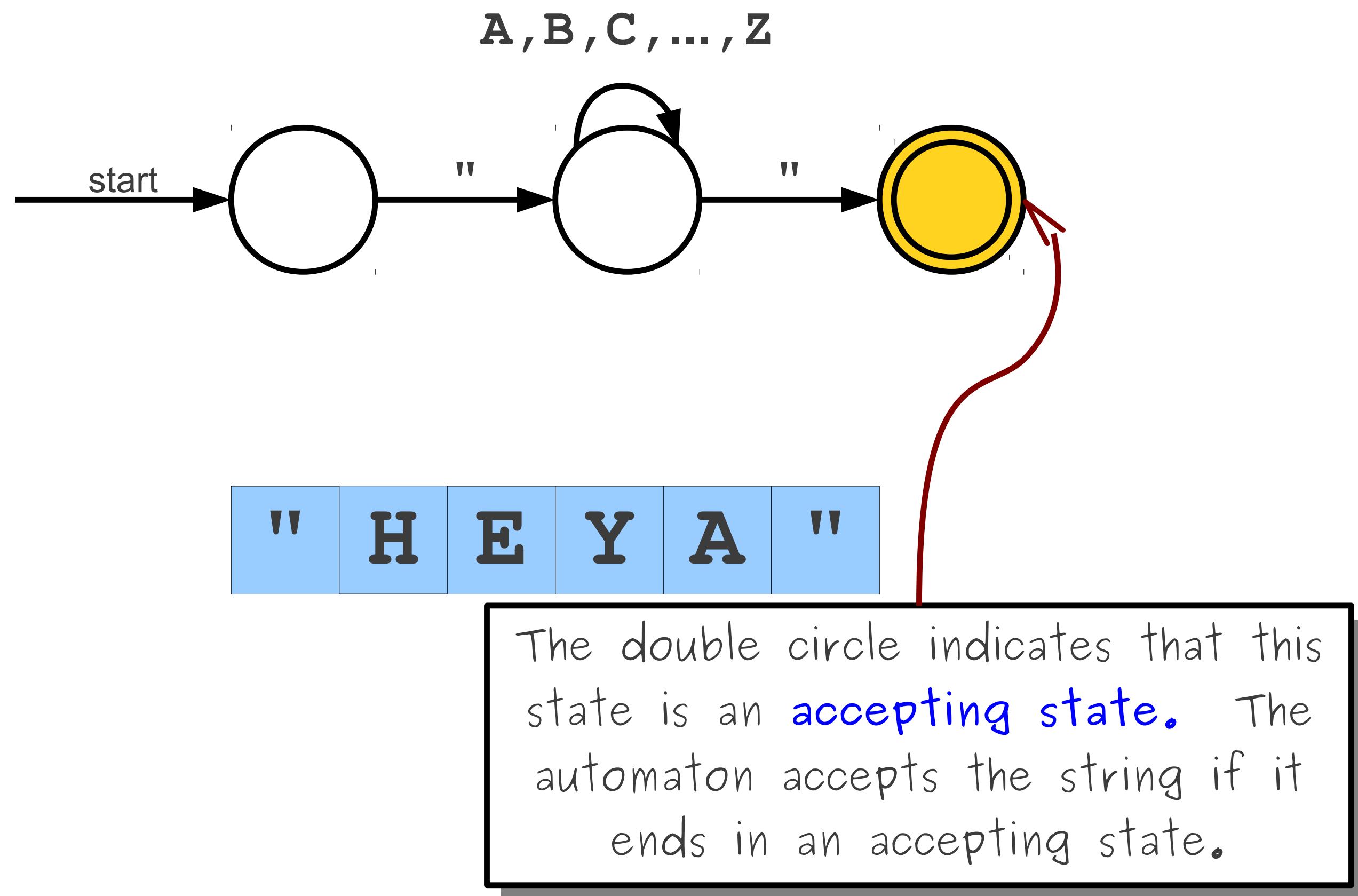


A Simple Automaton

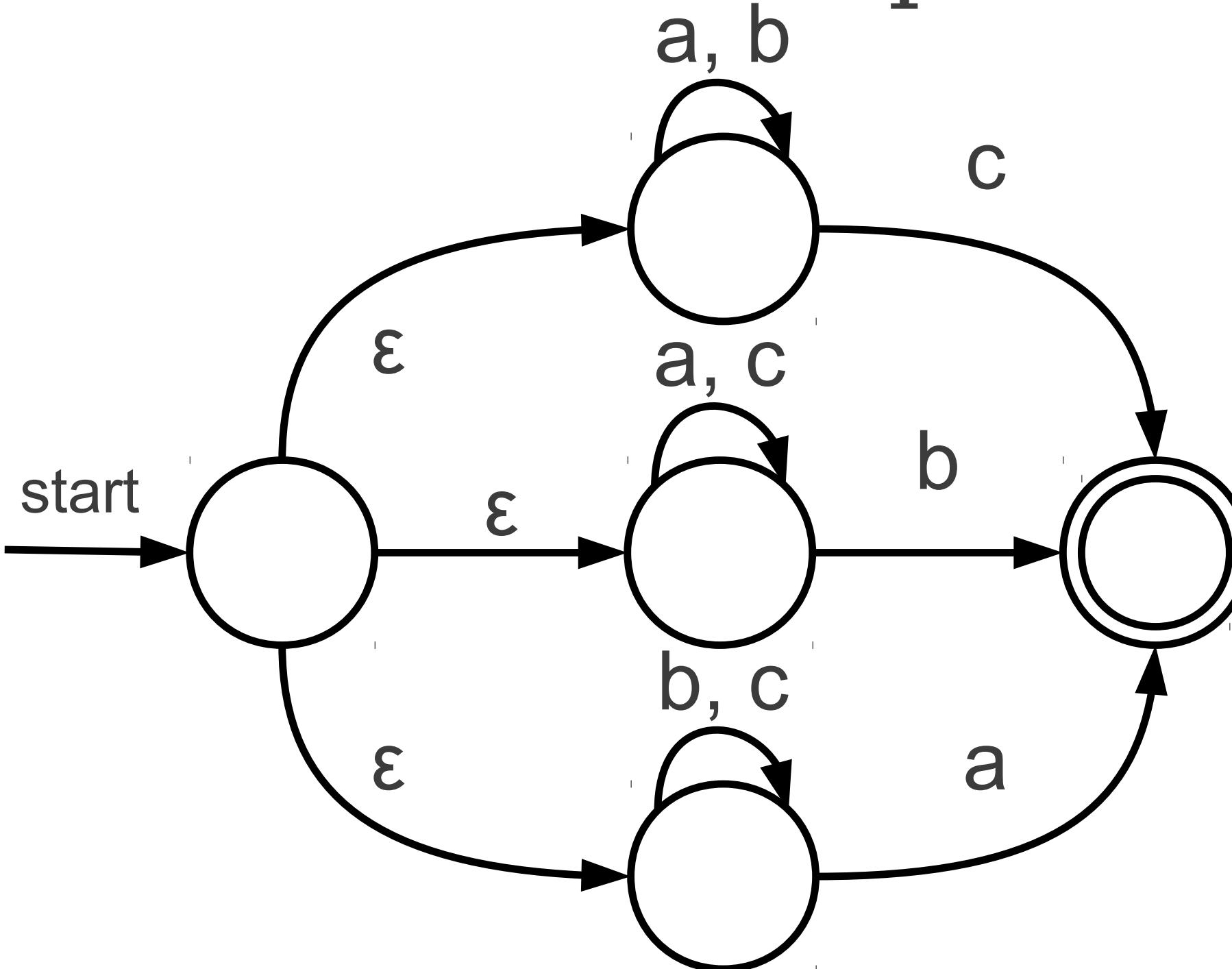


"	H	E	Y	A	"
---	---	---	---	---	---

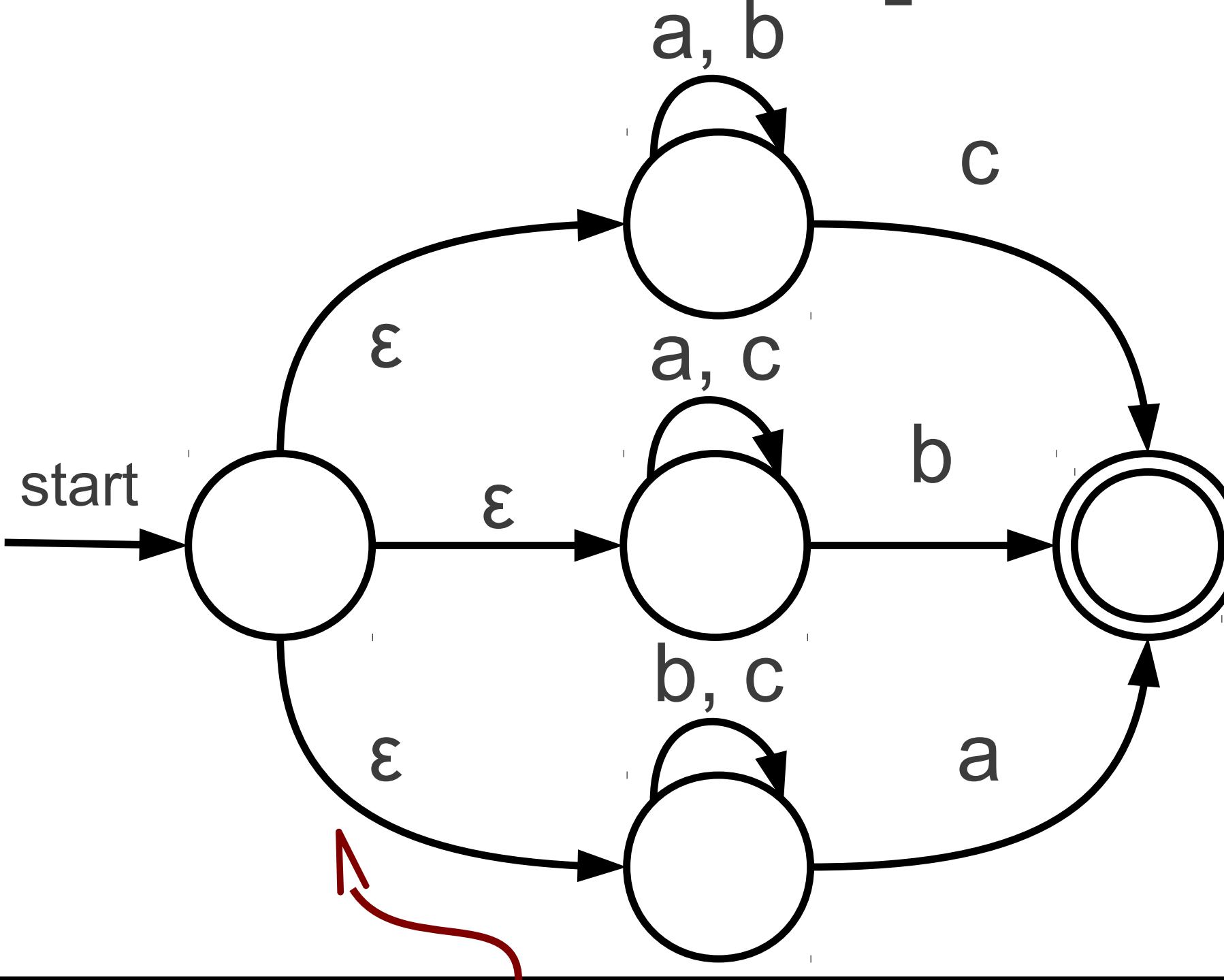
A Simple Automaton



An Even More Complex Automaton

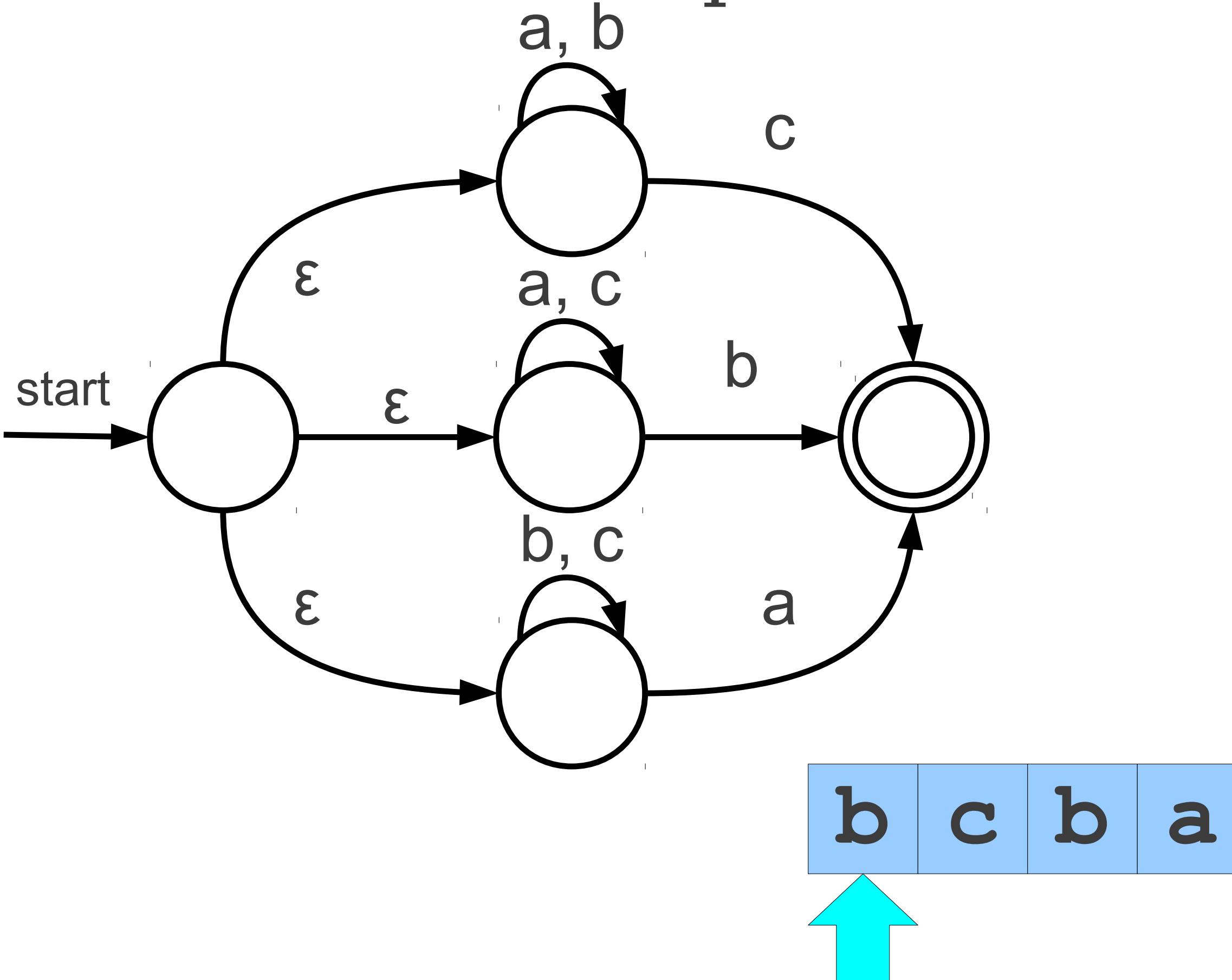


An Even More Complex Automaton

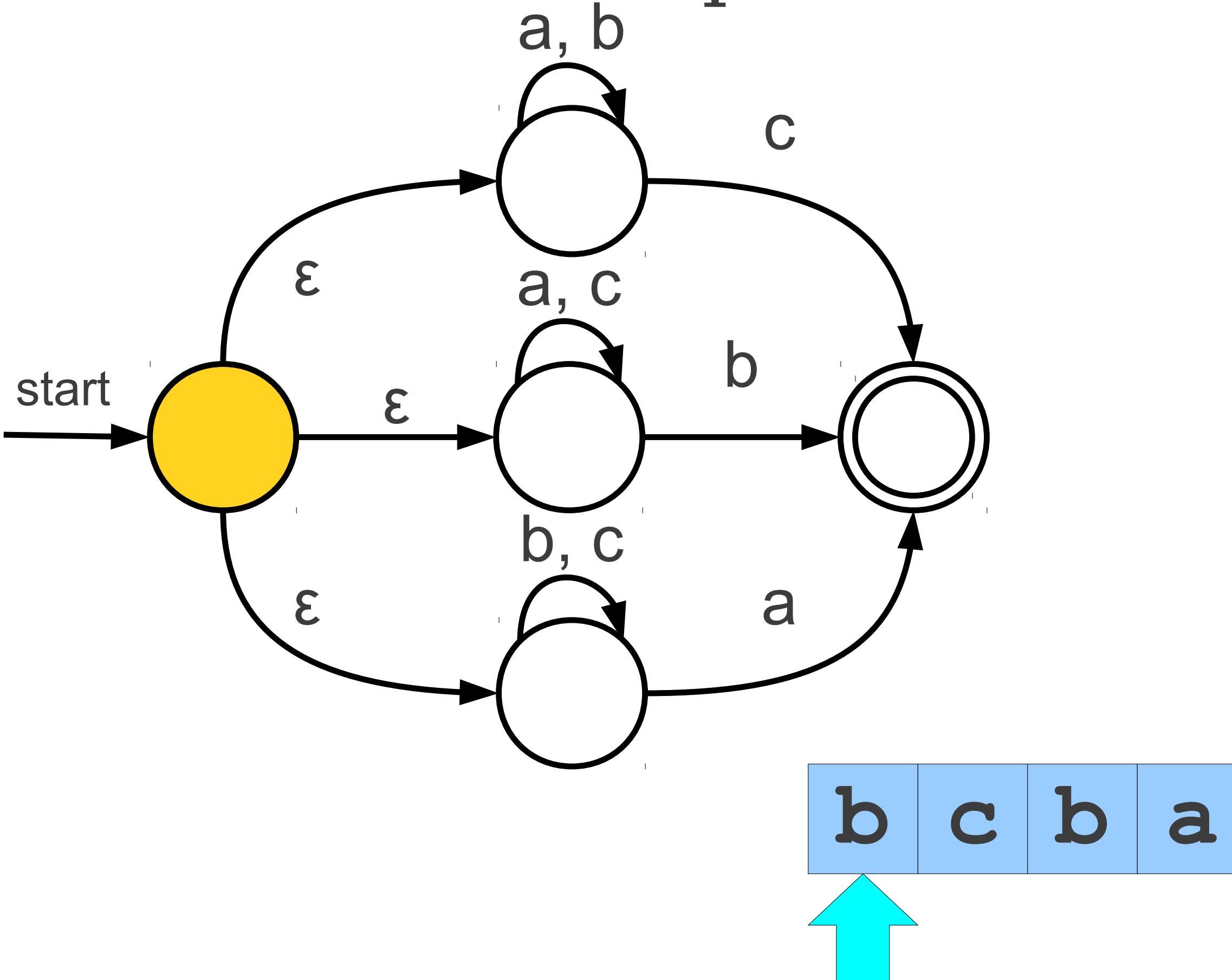


These are called **ϵ -transitions**. These transitions are followed automatically and without consuming any input.

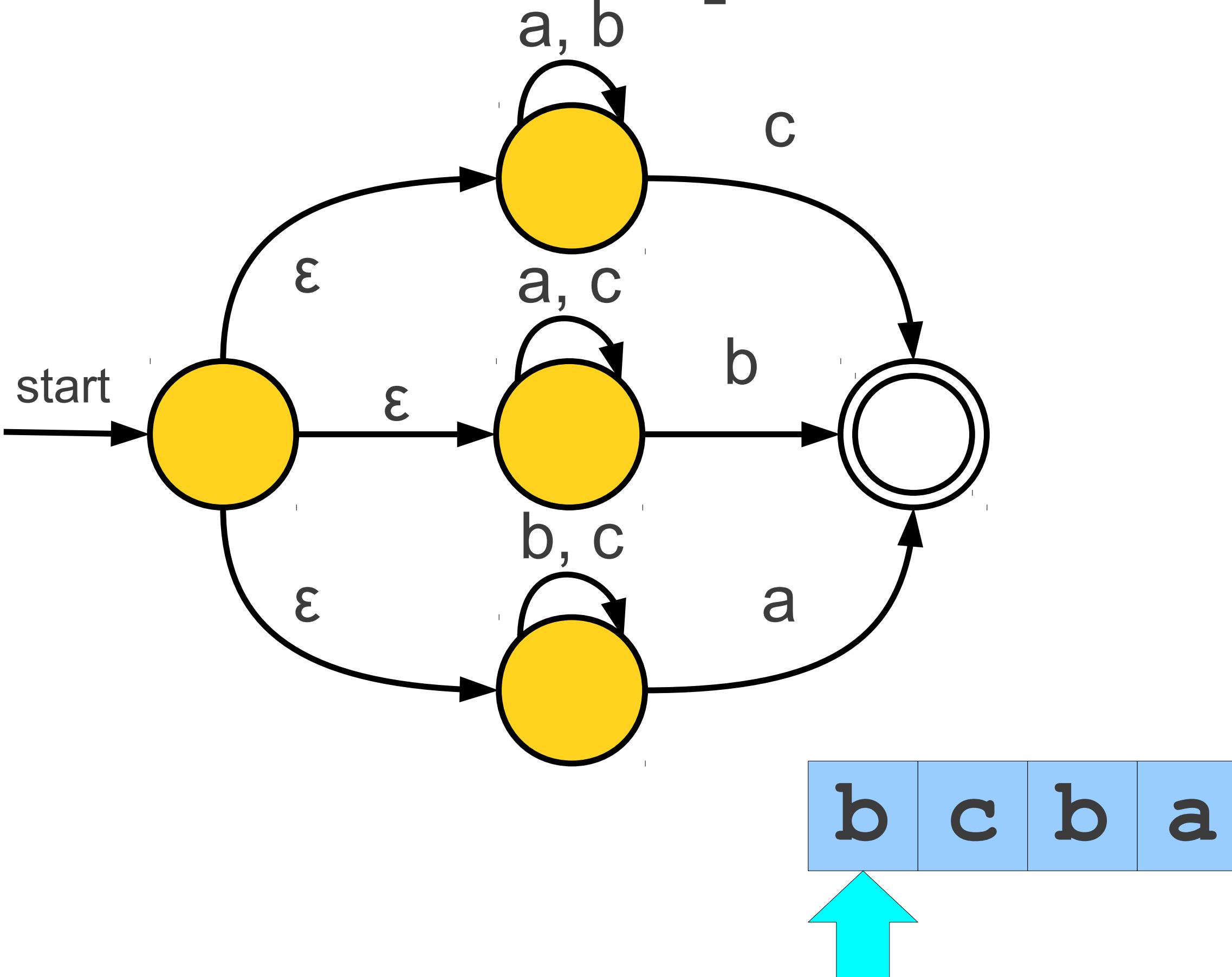
An Even More Complex Automaton



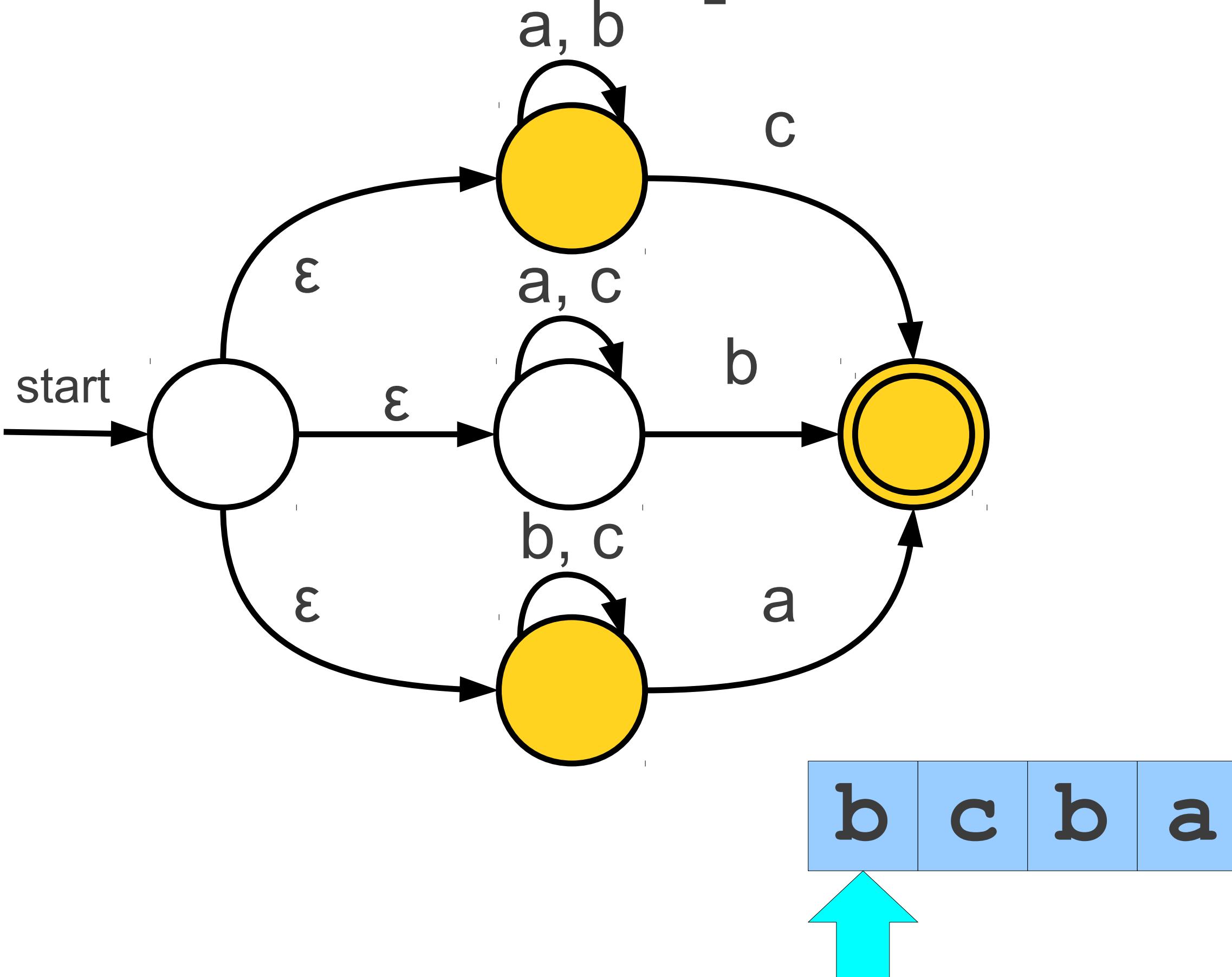
An Even More Complex Automaton



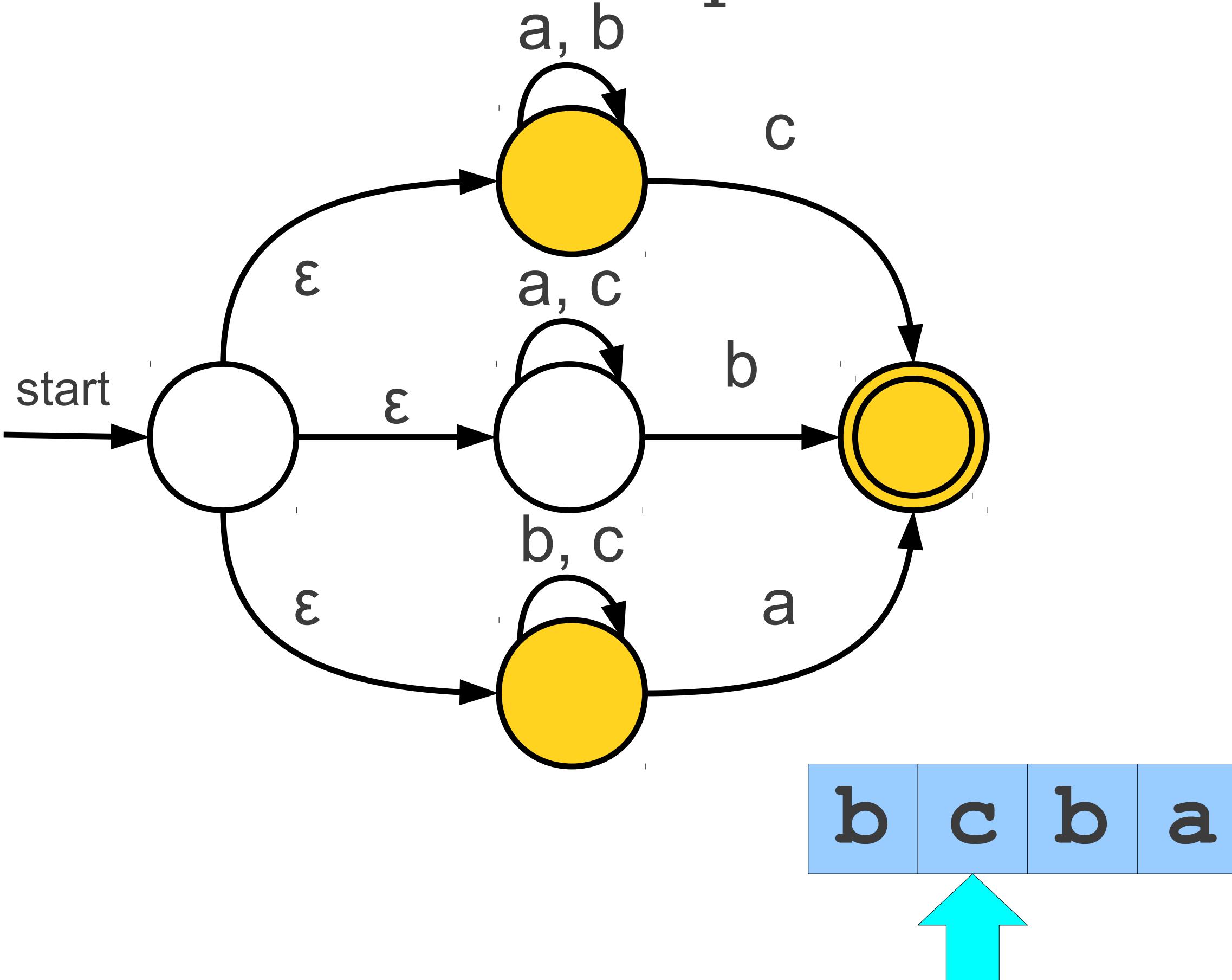
An Even More Complex Automaton



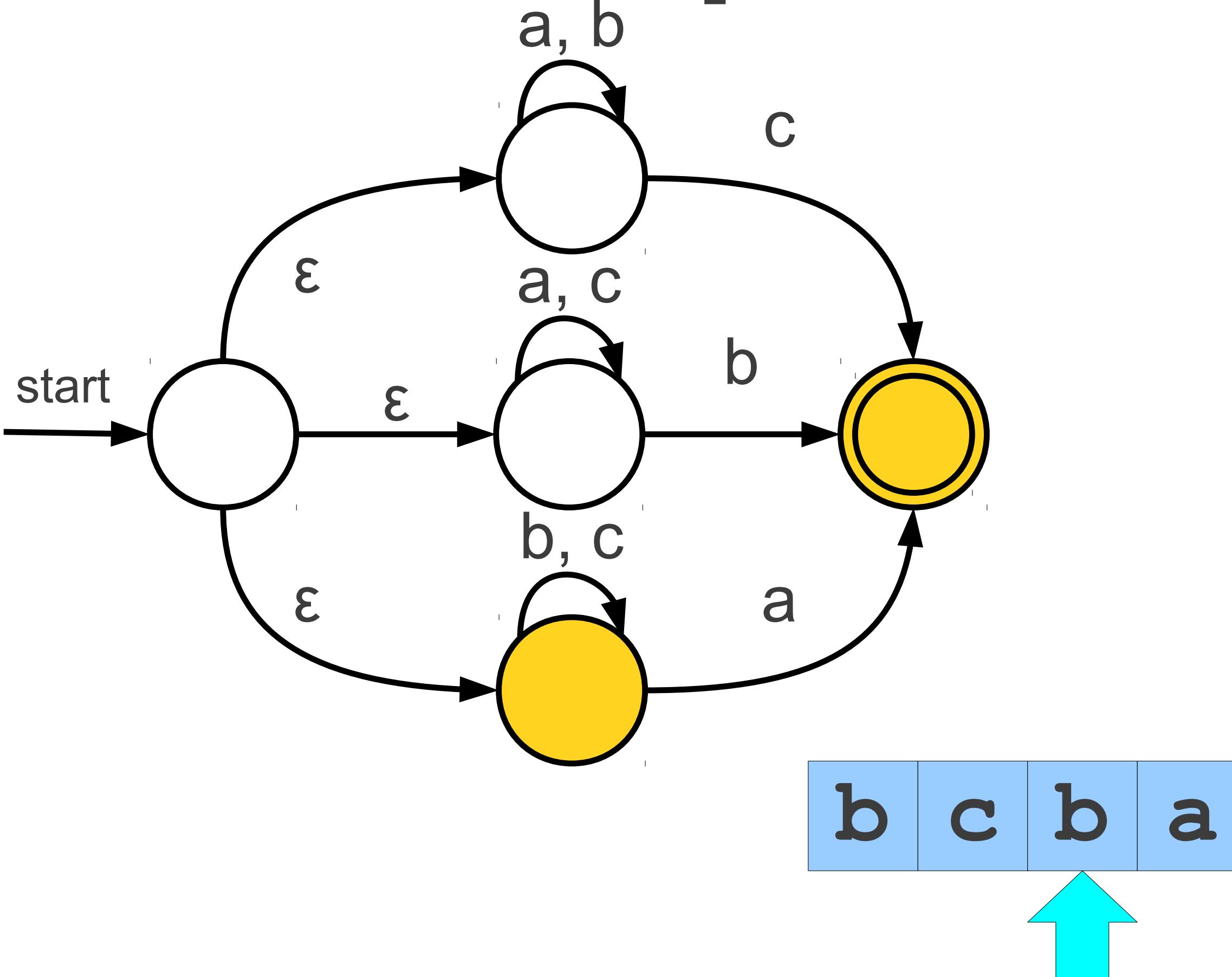
An Even More Complex Automaton



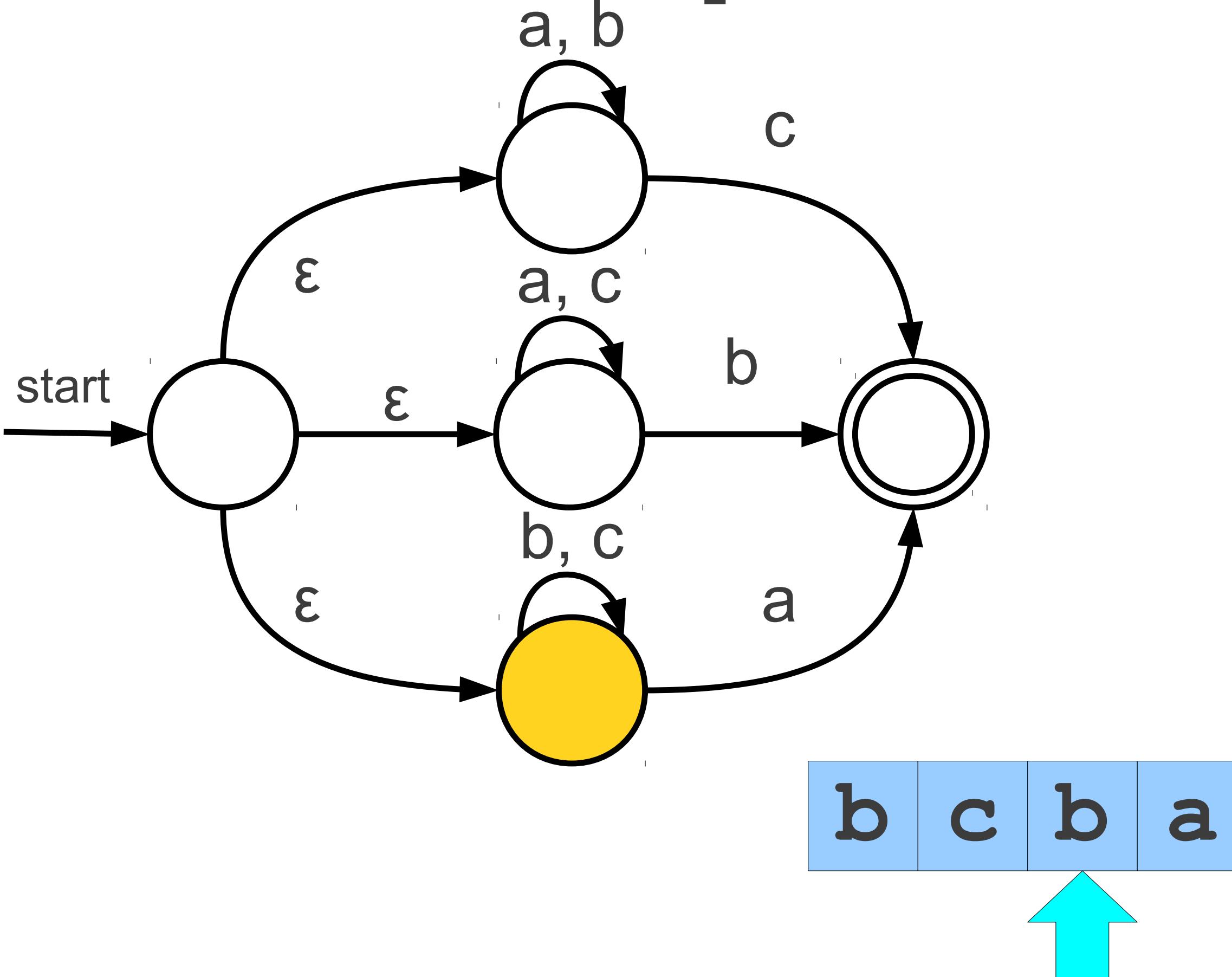
An Even More Complex Automaton



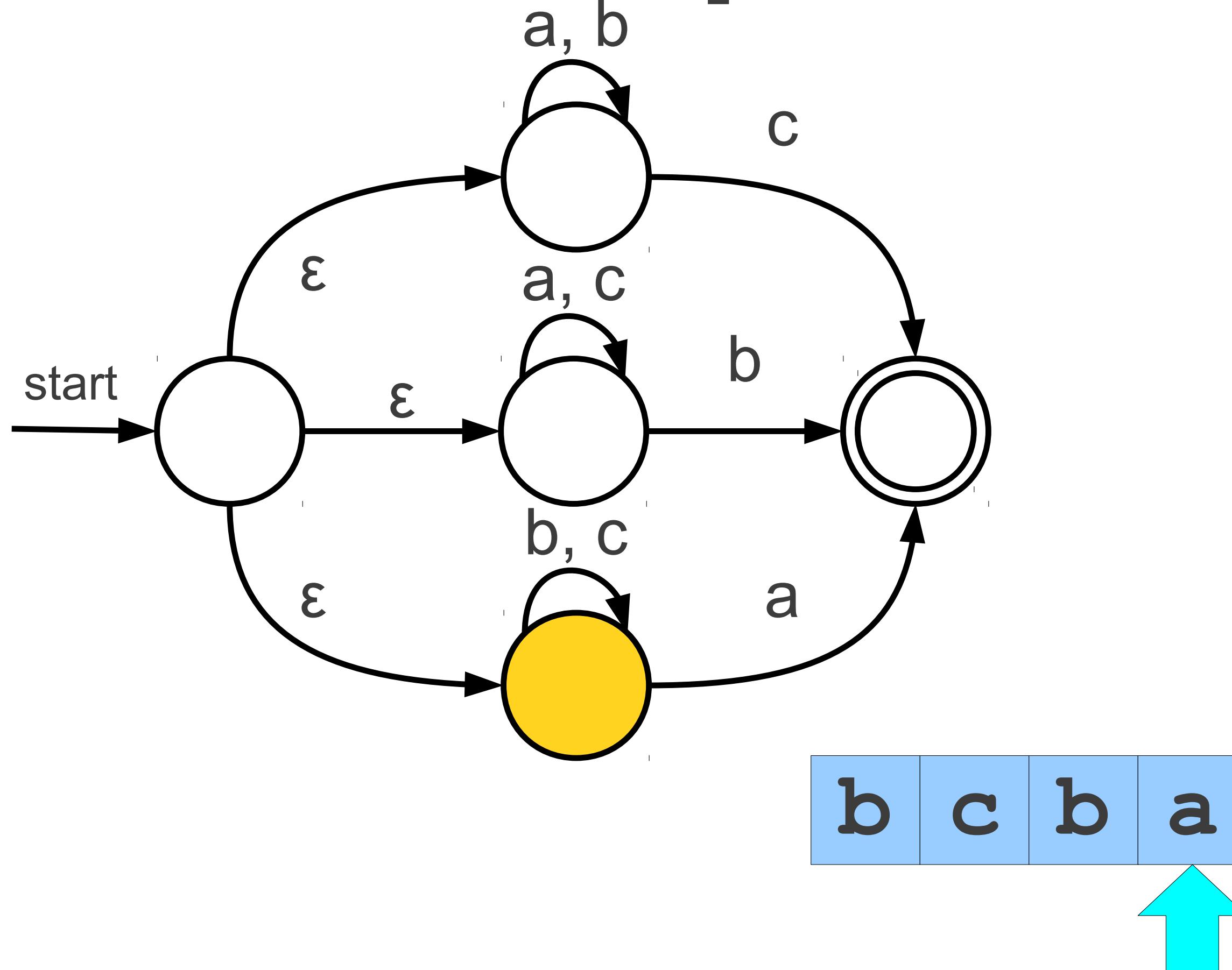
An Even More Complex Automaton



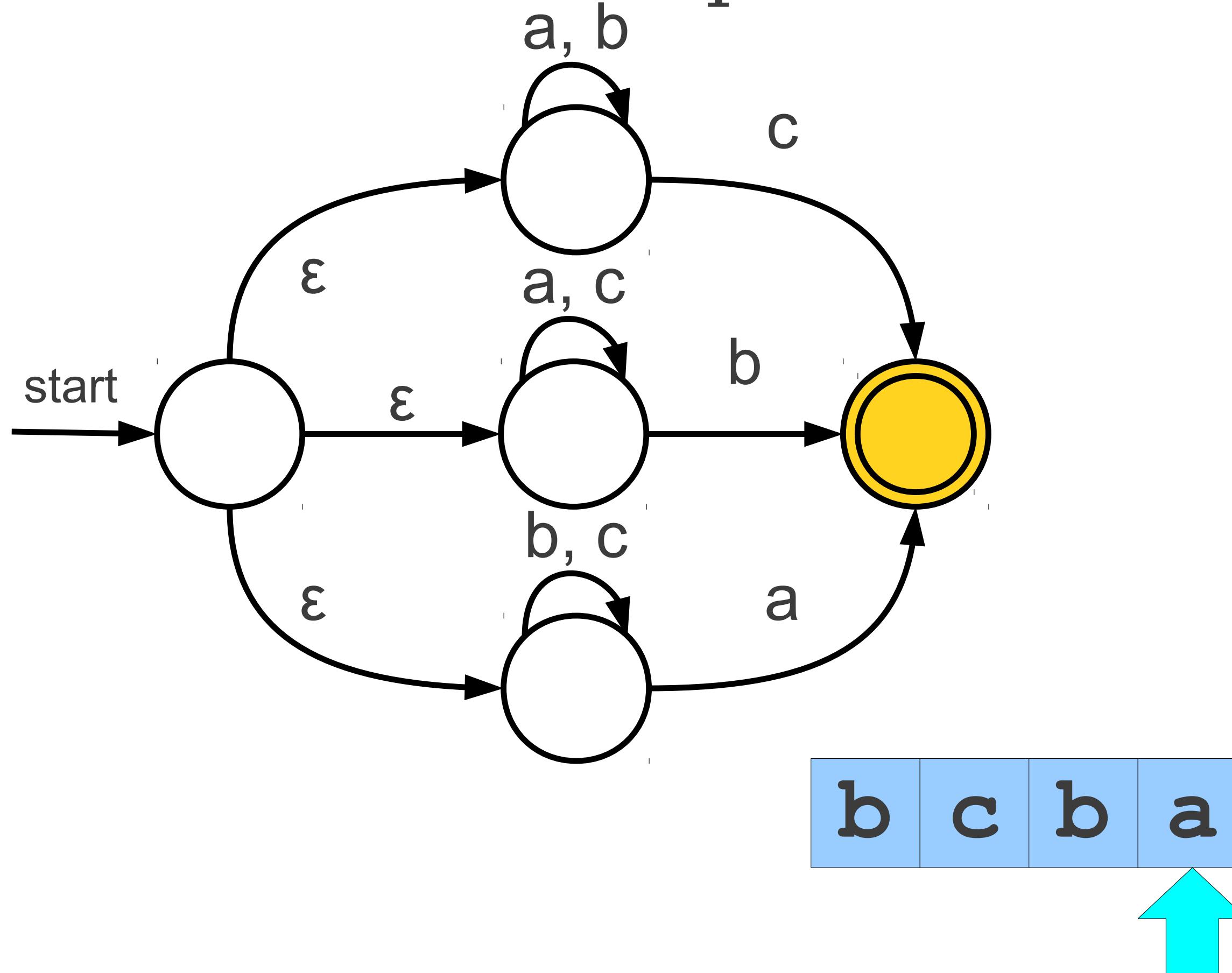
An Even More Complex Automaton



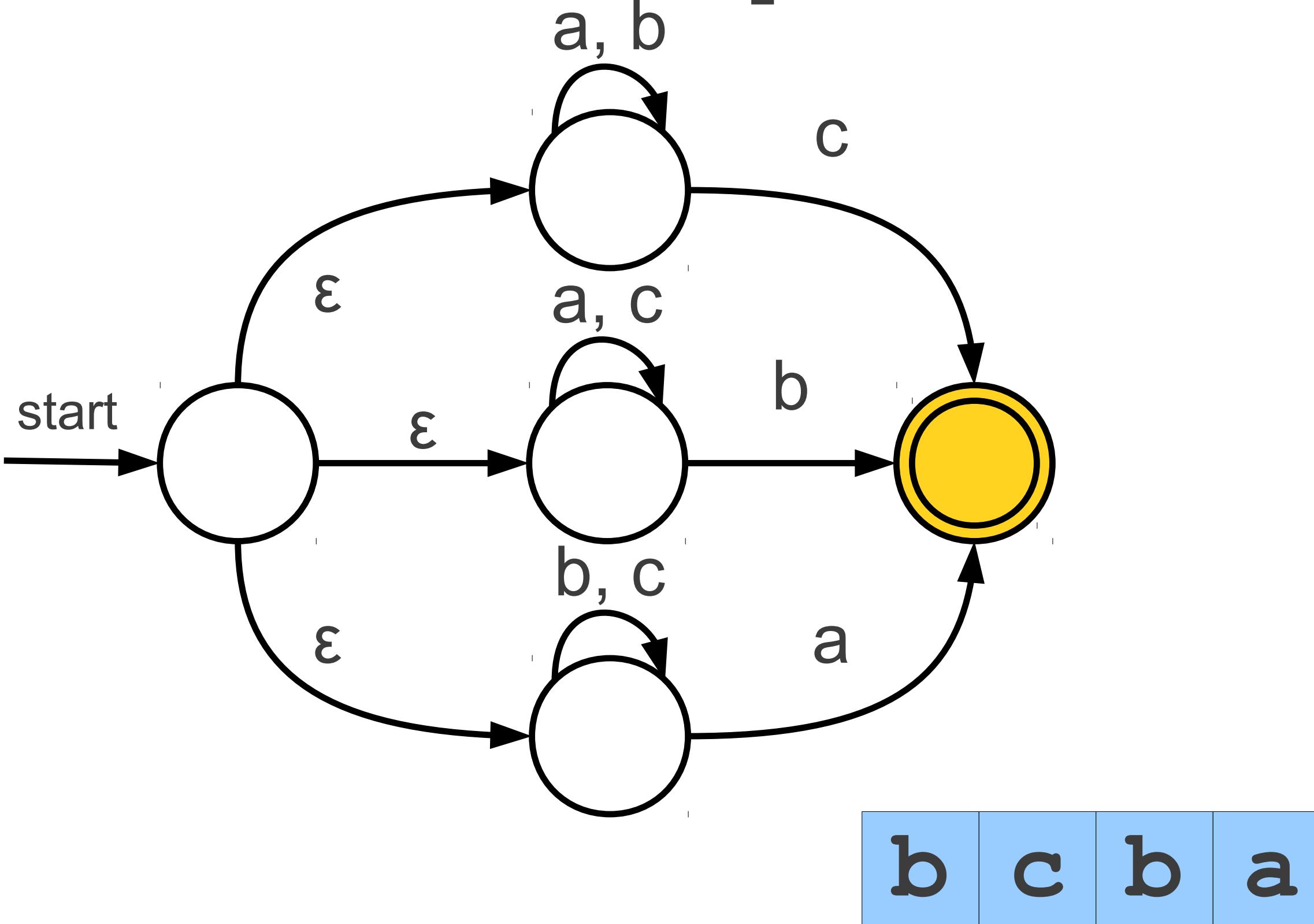
An Even More Complex Automaton



An Even More Complex Automaton



An Even More Complex Automaton



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 supported by current hardware. Need to **simulate** this instead.

DFA vs. NFA

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

Automating Lexical Analyzer (scanner) Construction

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

- Build an NFA for each term
- Combine them with ϵ -moves

NFA \rightarrow DFA (*subset construction*)

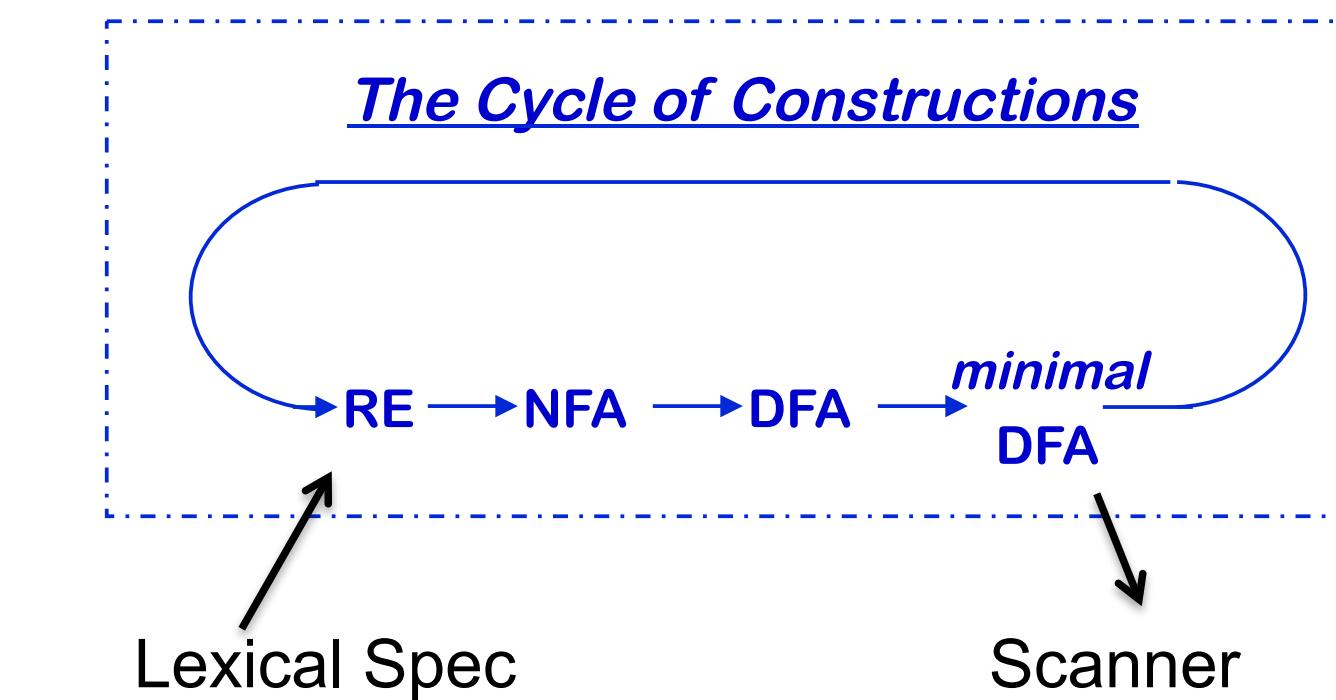
- Build the simulation

DFA \rightarrow Minimal DFA

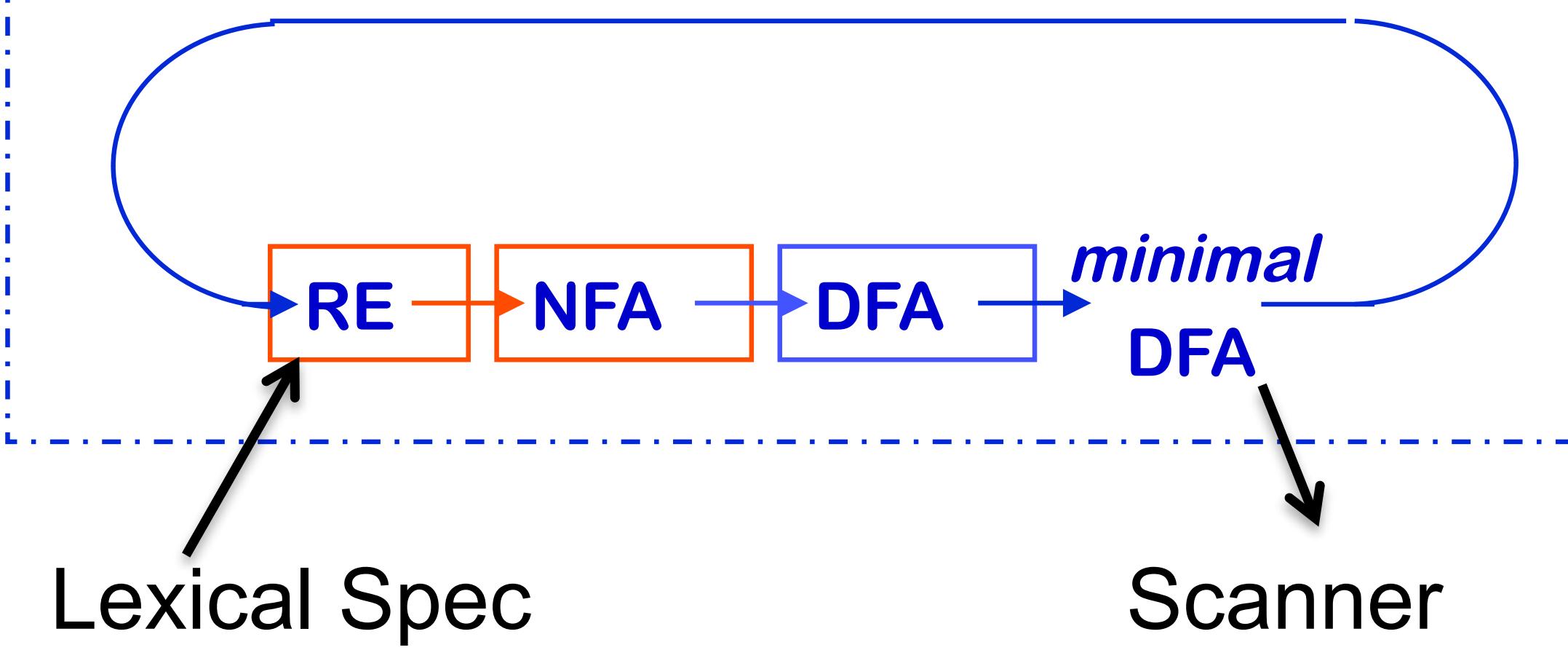
- Hopcroft's algorithm

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



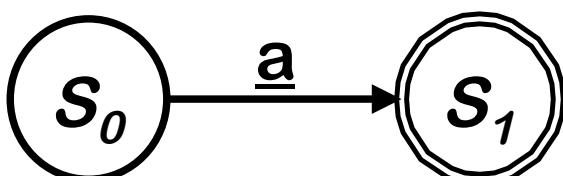
The Cycle of Constructions



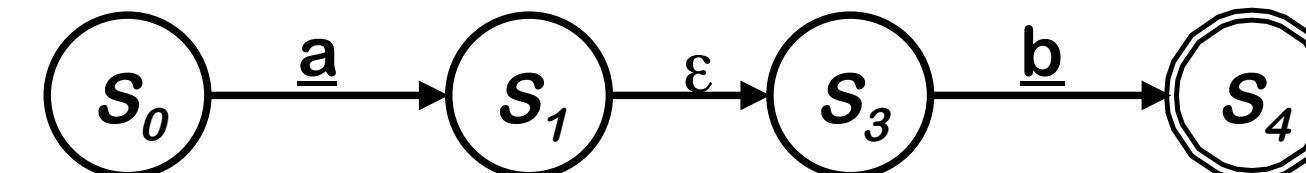
RE \rightarrow NFA using Thompson's Construction

Key idea

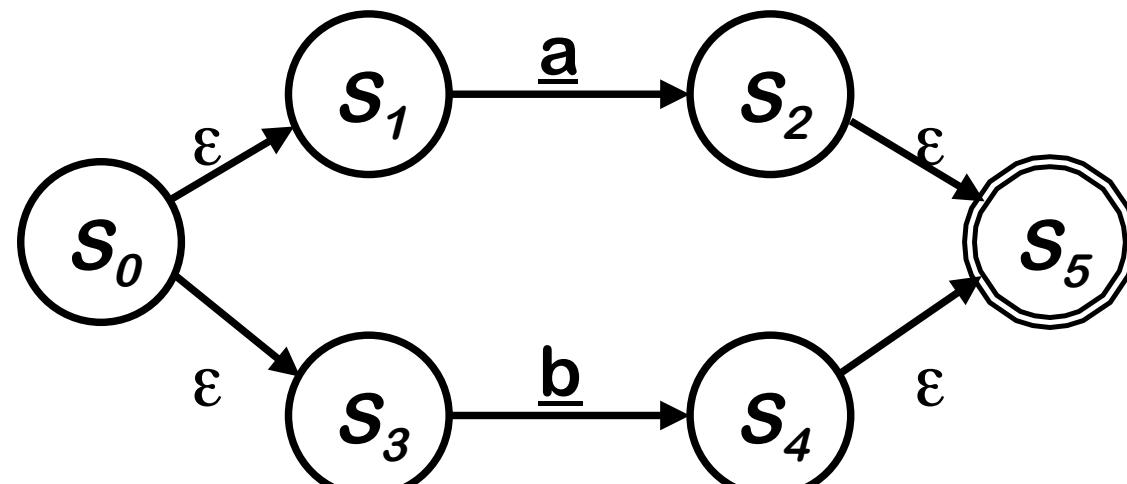
- NFA pattern for each symbol & each operator
- Join them with ϵ moves in precedence order



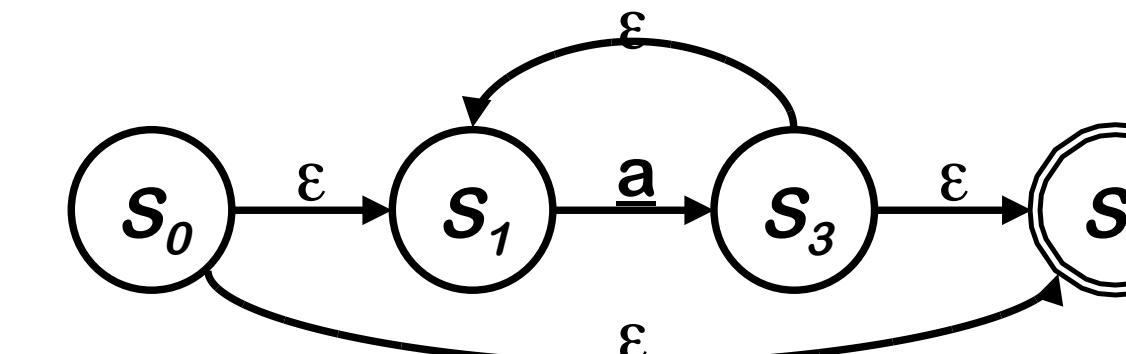
NFA for a



NFA for ab



NFA for a | b



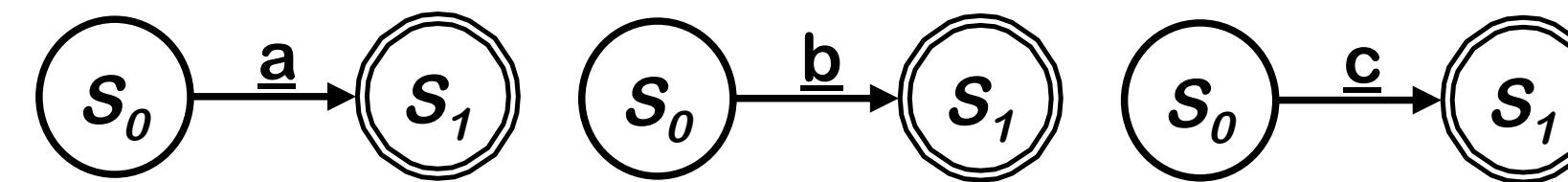
NFA for a^{*}

Ken Thompson, CACM, 1968

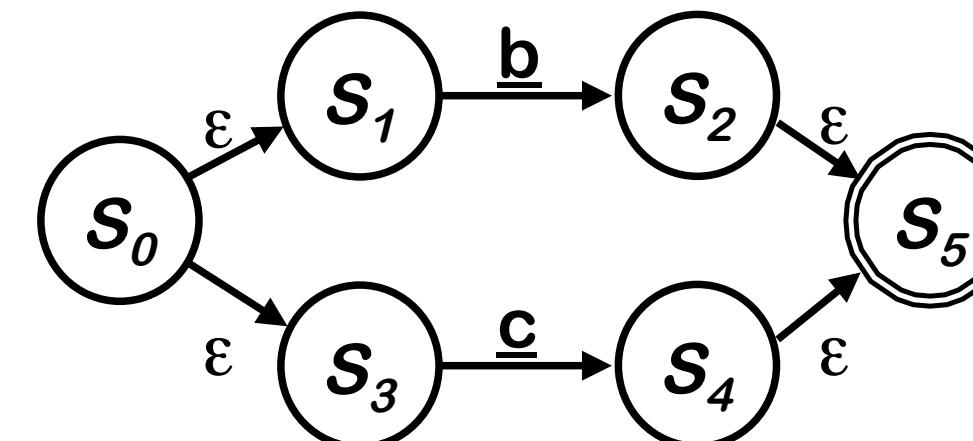
Example of Thompson's Construction

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

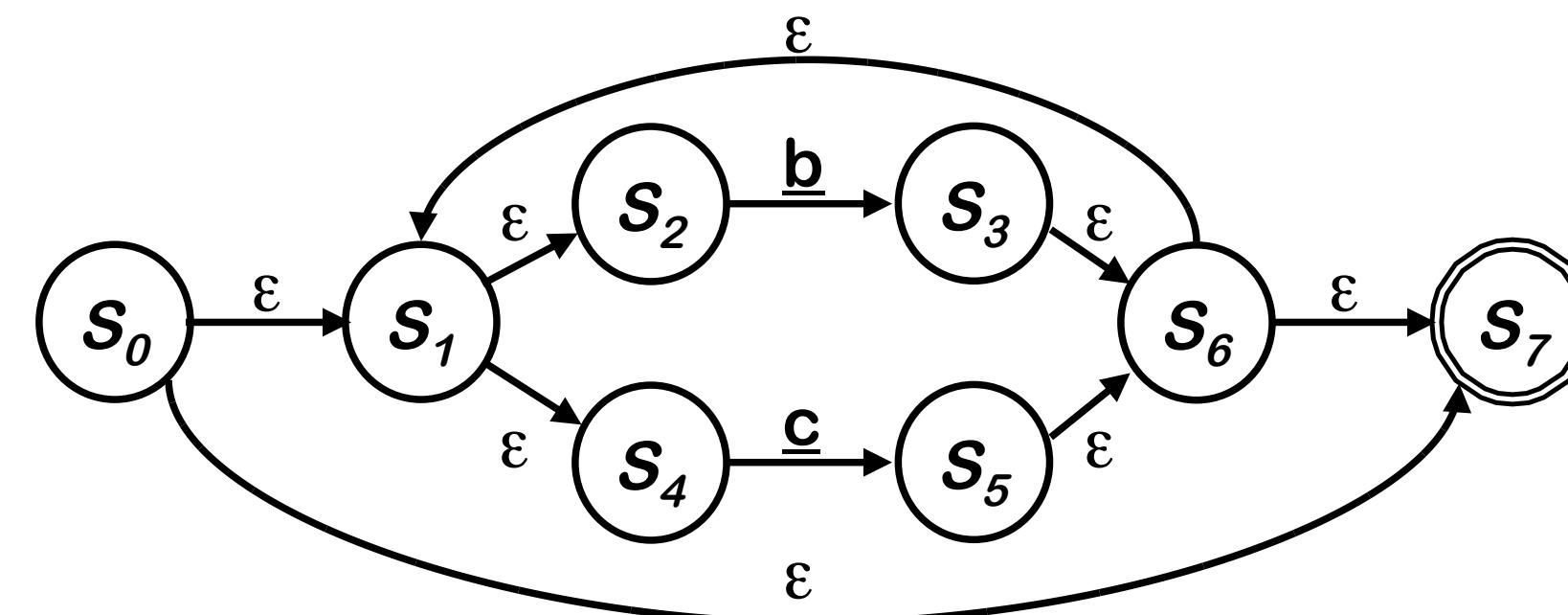
1. \underline{a} , \underline{b} , & \underline{c}



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

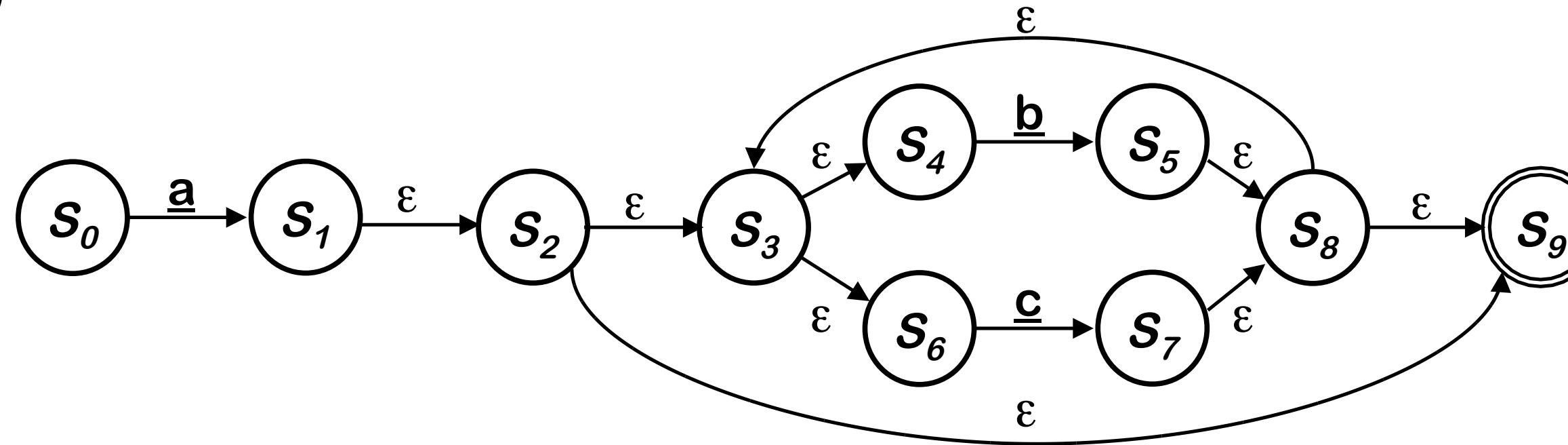


3. $(\underline{b} \mid \underline{c})^*$

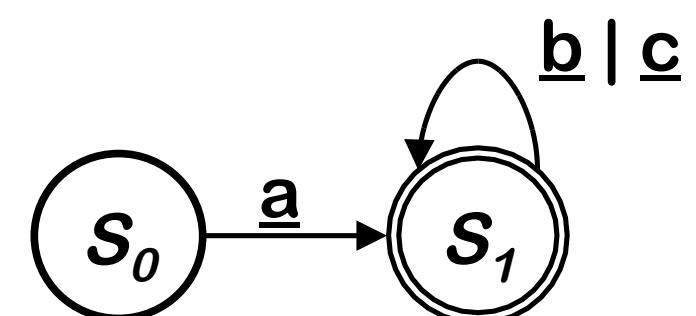


Example of Thompson's Construction (con't)

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

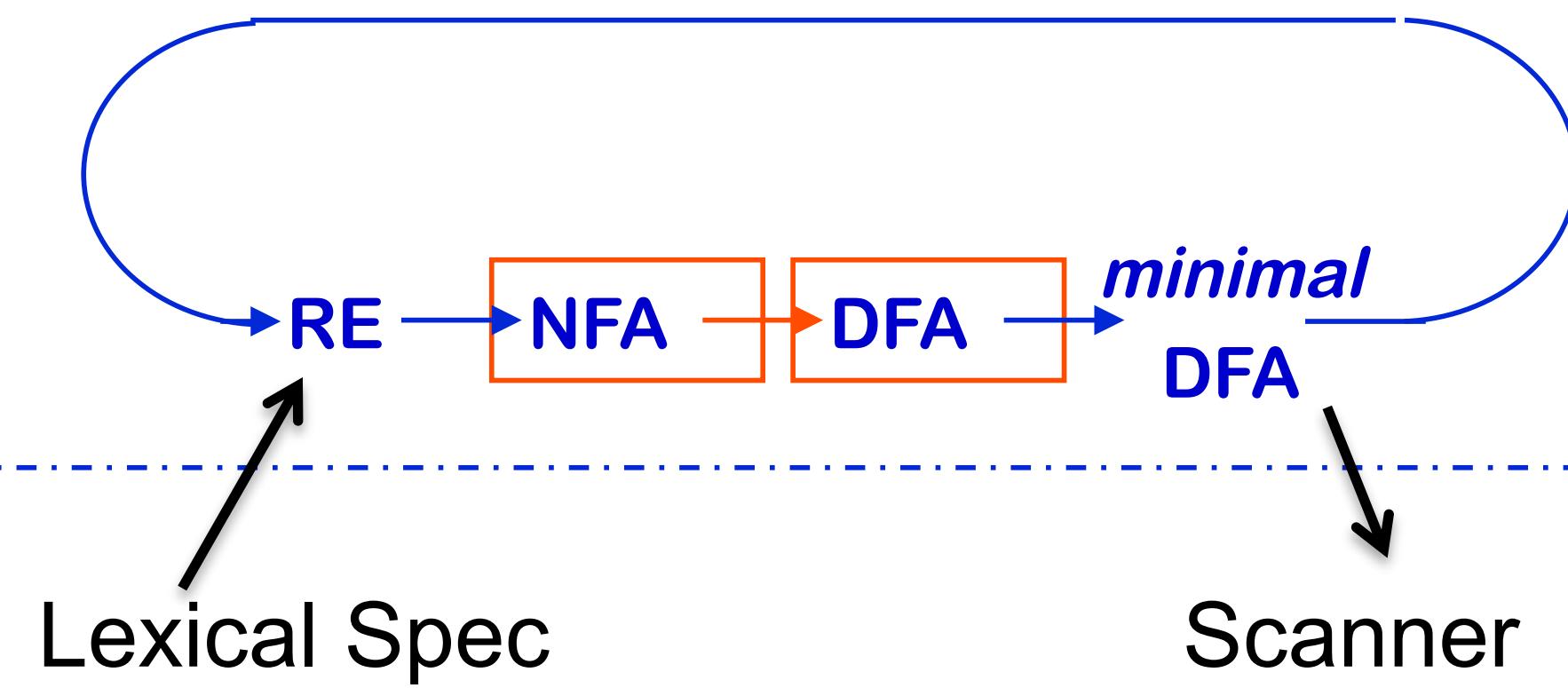


Of course, a human would design something simpler ...



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

The Cycle of Constructions

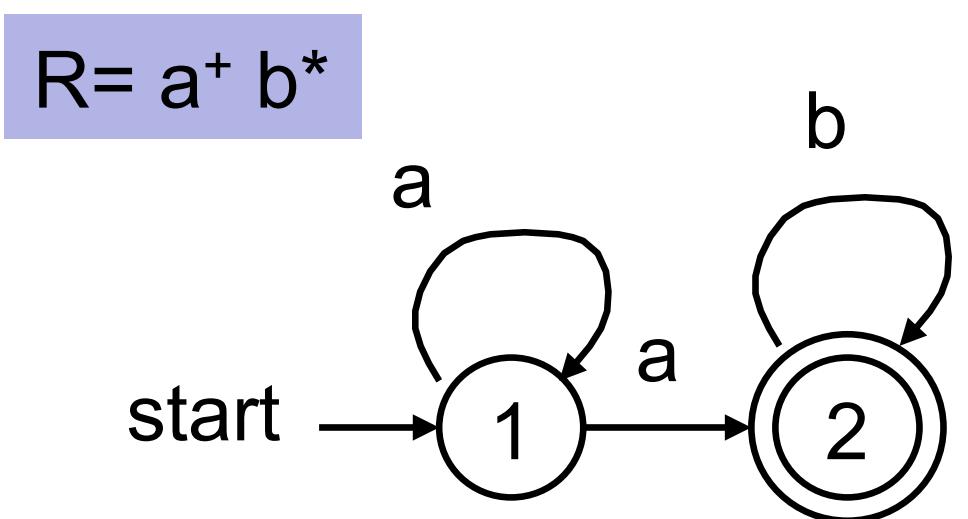


NFA to DFA : Trick

- 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 ϵ -moves from NFA start state
- Add a transition $S \rightarrow^a S'$ to DFA iff
 - S' is the set of NFA states reachable from any state in S after seeing the input a , considering ϵ -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



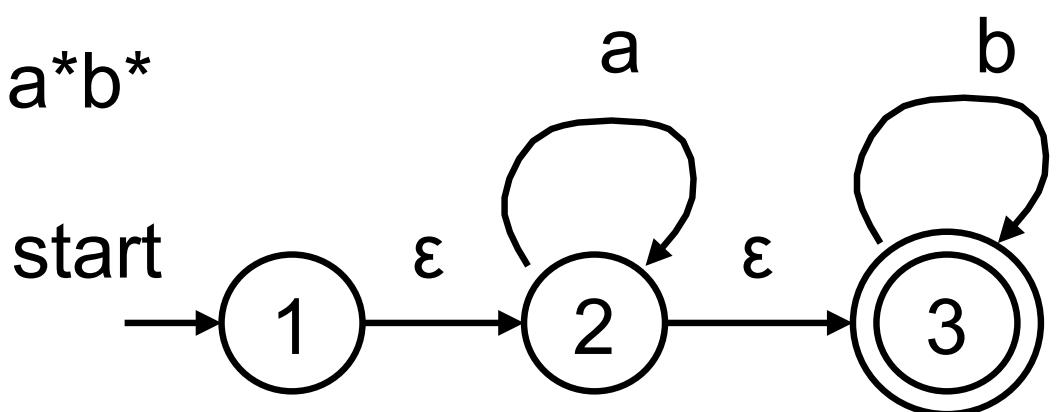
ϵ -closure(1) = {1} include state “1”

(1,a) \rightarrow {1,2} include state “1/2”

(1,b) \rightarrow ERROR

NFA to DFA (3)

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

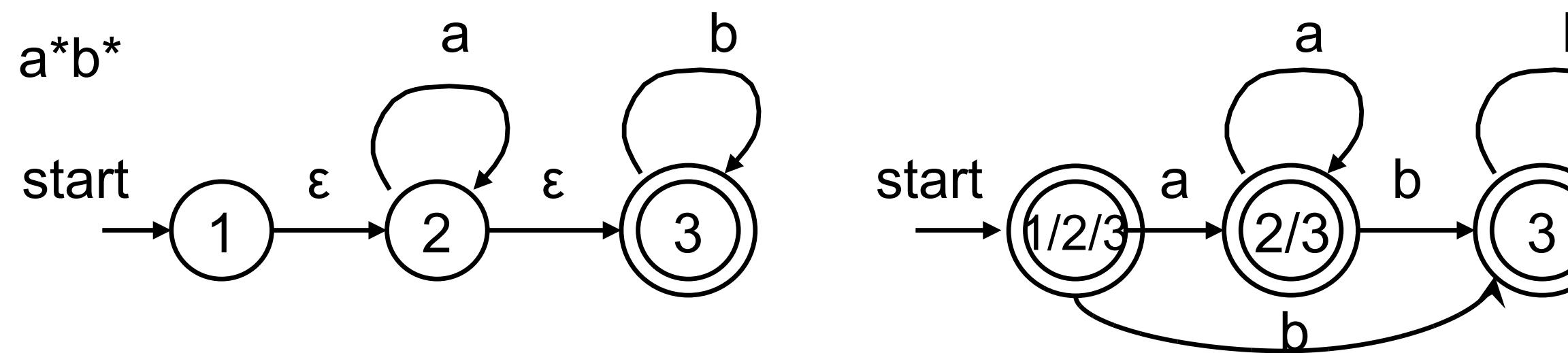


1. $\epsilon\text{-closure}(1) = \{1,2,3\}$; include 1/2/3
2. $\text{Move}(1/2/3, a) = \{2, 3\} + \epsilon\text{-closure}(2,3) = \{2,3\}$; include 2/3
3. $\text{Move}(1/2/3, b) = \{3\} + \epsilon\text{-closure}(3) = \{3\}$; include state 3
4. $\text{Move}(2/3, a) = \{2\} + \epsilon\text{-closure}(2)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

NFA to DFA (3)

- ϵ transitions

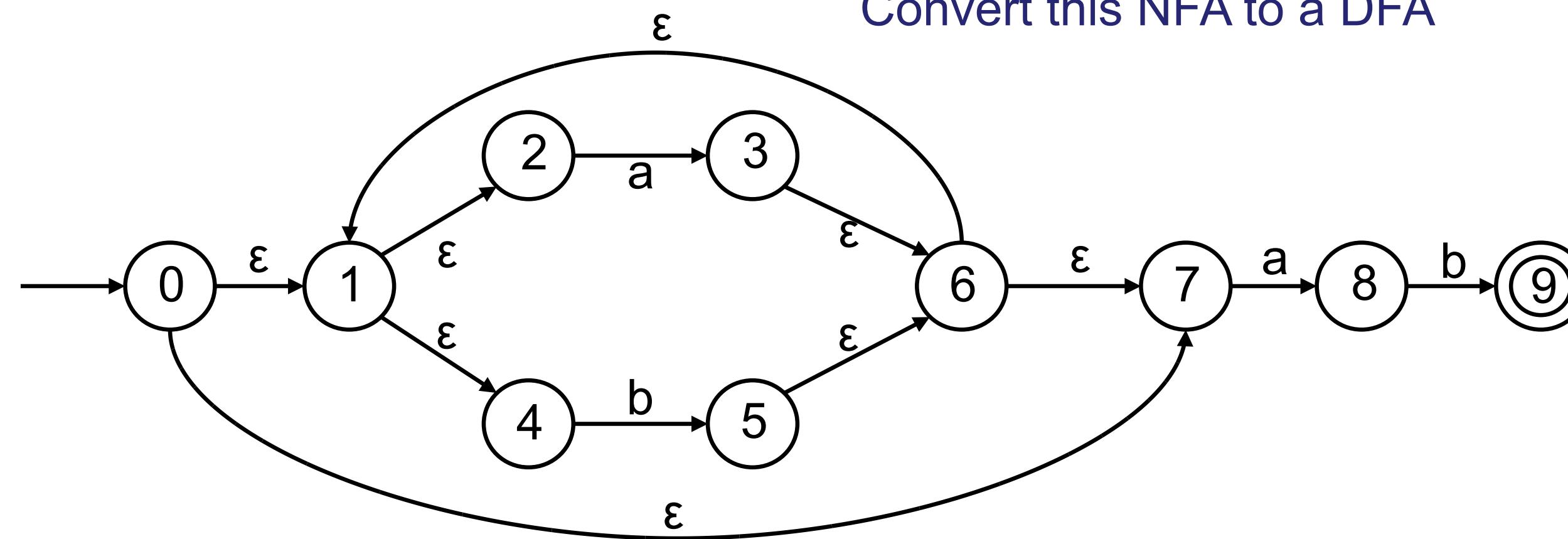
- Any state reachable by an ϵ transition is “part of the state”
- ϵ -closure - Any state reachable from S by ϵ transitions is in the ϵ -closure; treat ϵ -closure as 1 big state, always include ϵ -closure as part of the state



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

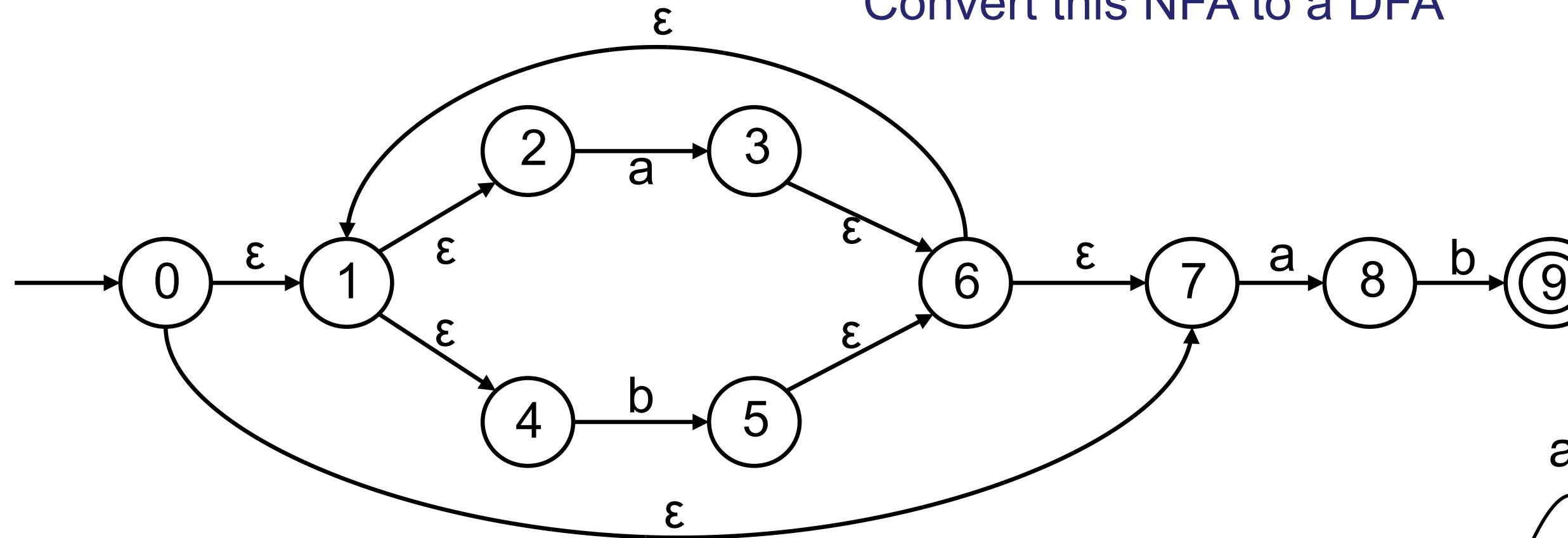
Class Problem

Convert this NFA to a DFA



Class Problem

Convert this NFA to a DFA

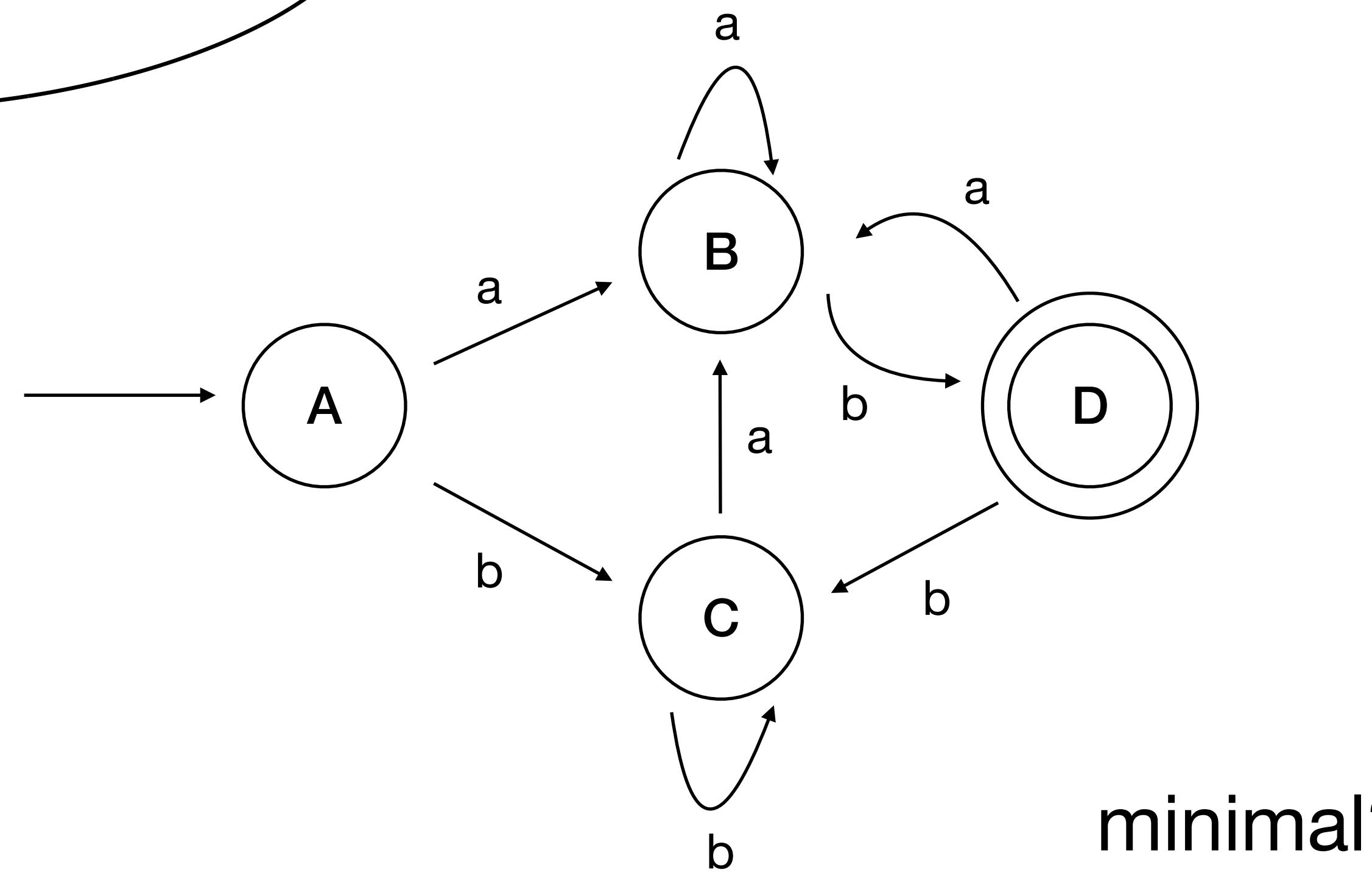


$$A = \{ 0, 1, 2, 4, 7 \}$$

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

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

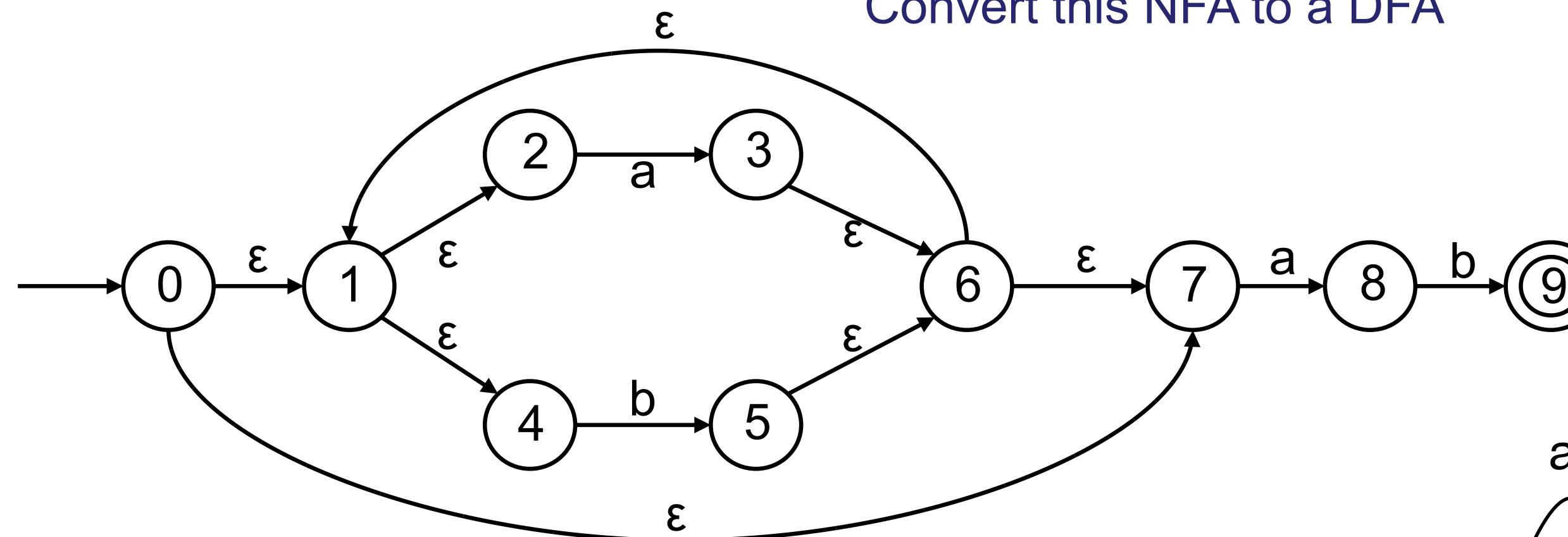
$$D = \{ 1, 2, 4, 5, 6, 8, 9 \}$$



minimal?

Class Problem

Convert this NFA to a DFA

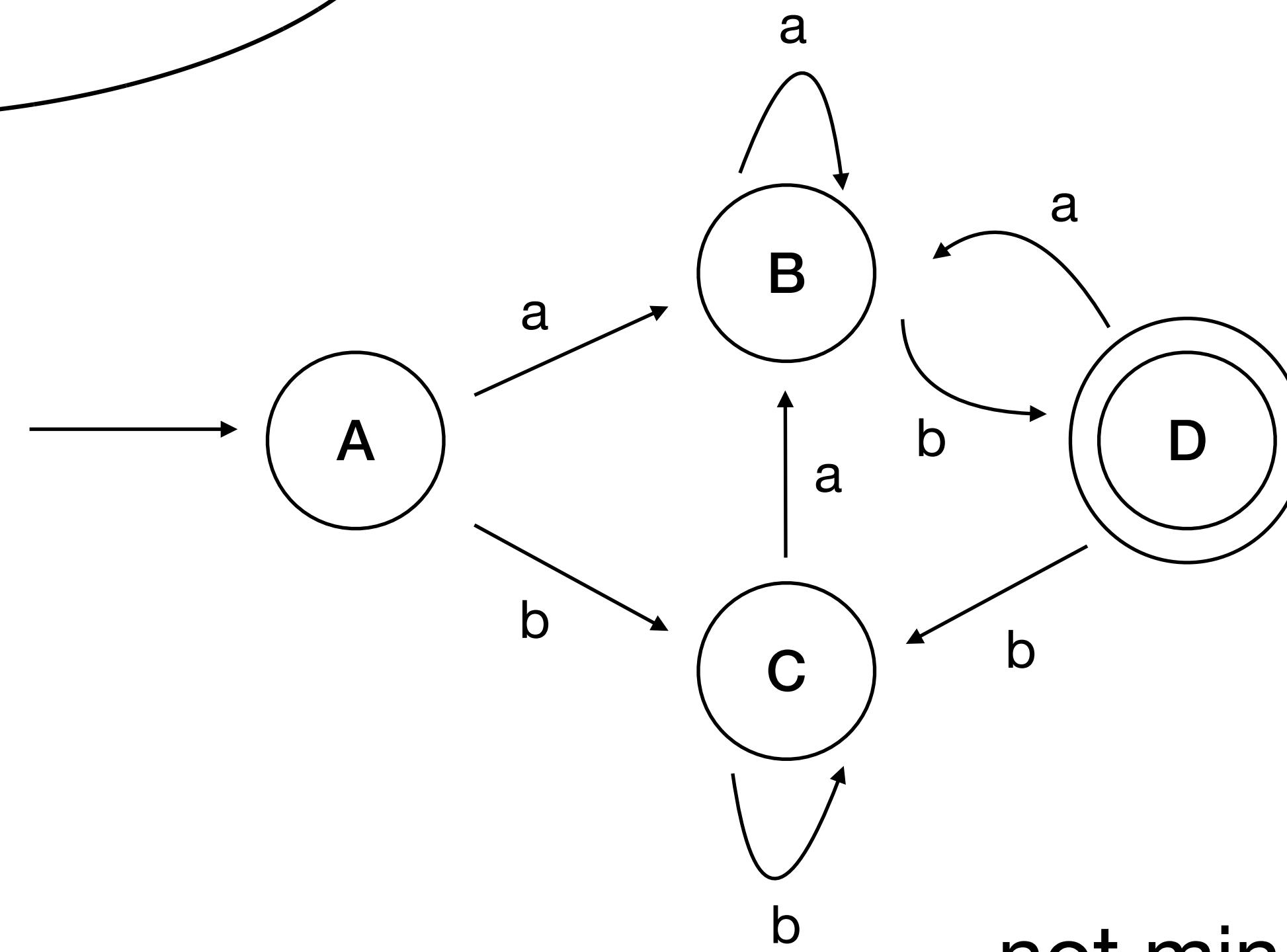


$$A = \{ 0, 1, 2, 4, 7 \} == (a|b)^*ab$$

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

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

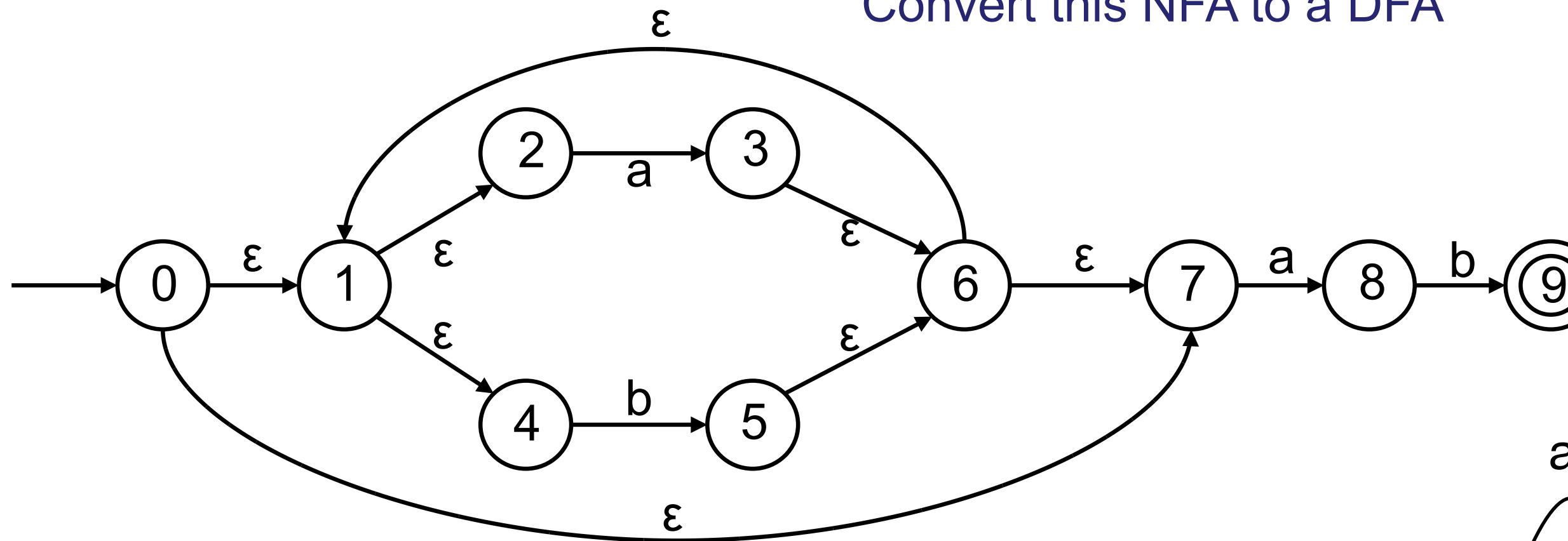
$$D = \{ 1, 2, 4, 5, 6, 8, 8, 9 \} == \epsilon|A$$



not minimal, A == C

Class Problem

Convert this NFA to a DFA

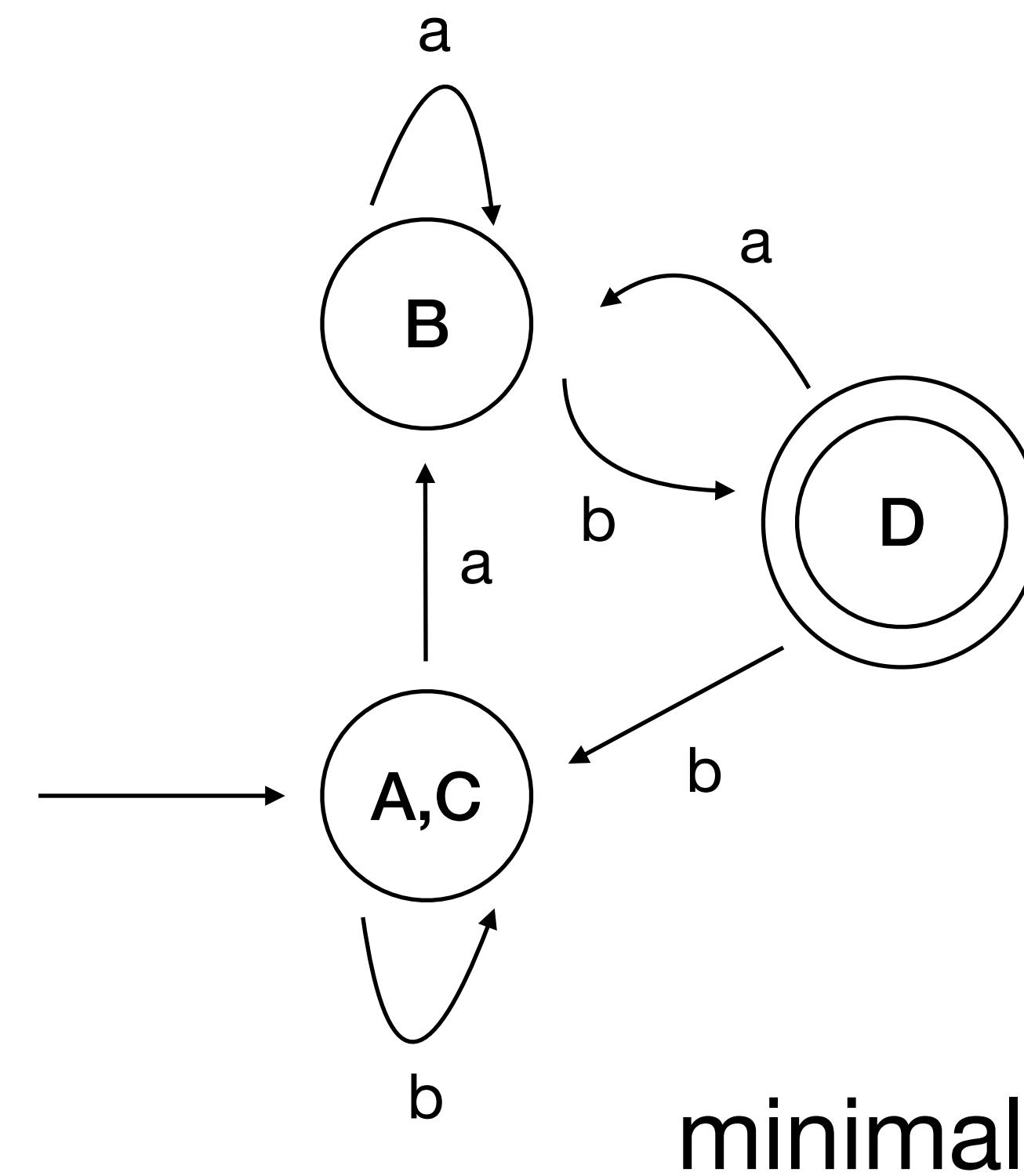


$$A = \{ 0, 1, 2, 4, 7 \} == (a|b)^*ab$$

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

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

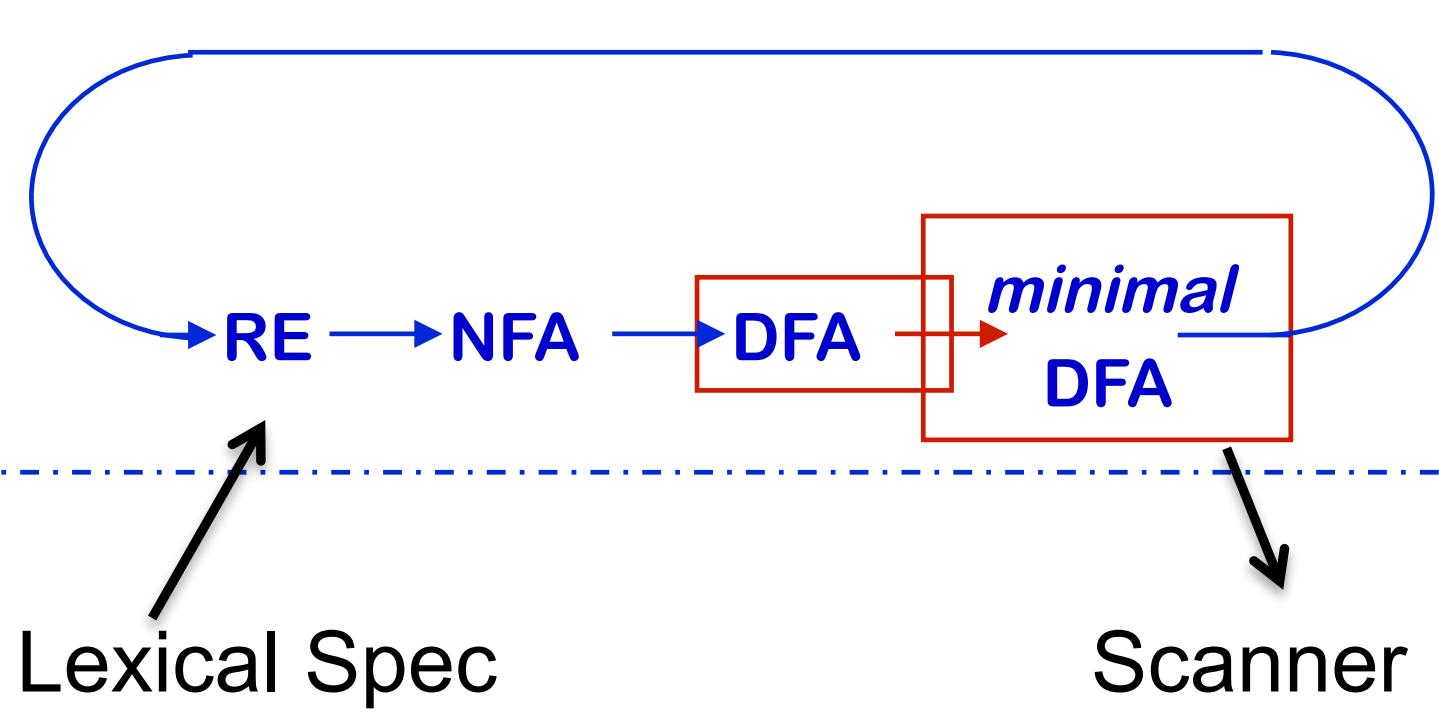
$$D = \{ 1, 2, 4, 5, 6, 8, 9 \} == \epsilon|A$$



NFA to DFA : cont..

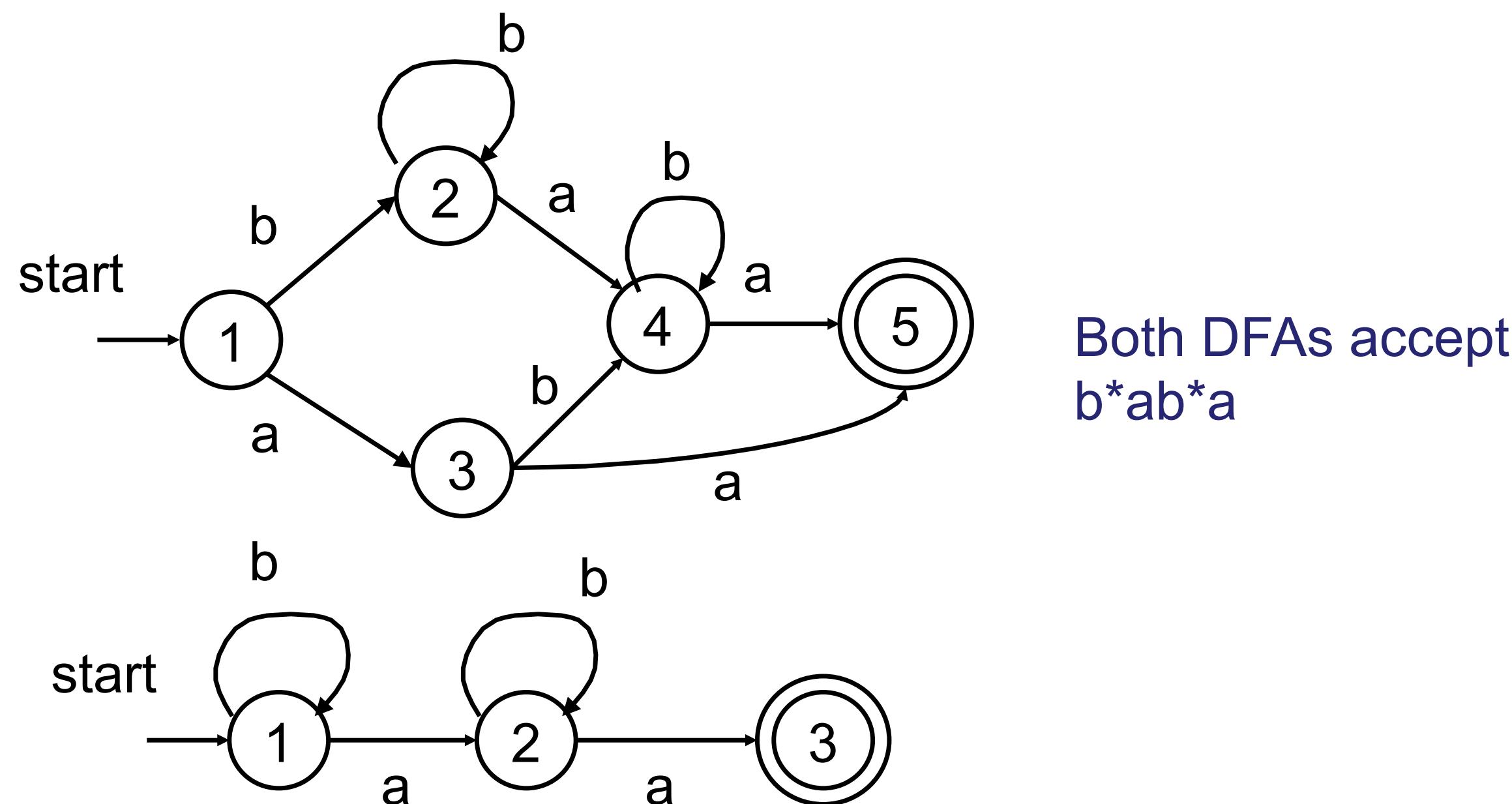
- An NFA may be in many states at any time
- How many different states ?
- If there are N states, the NFA must be in some subset of those N states
- How many subsets are there?
 $2^N - 1 = \text{finitely many}$

The Cycle of Constructions



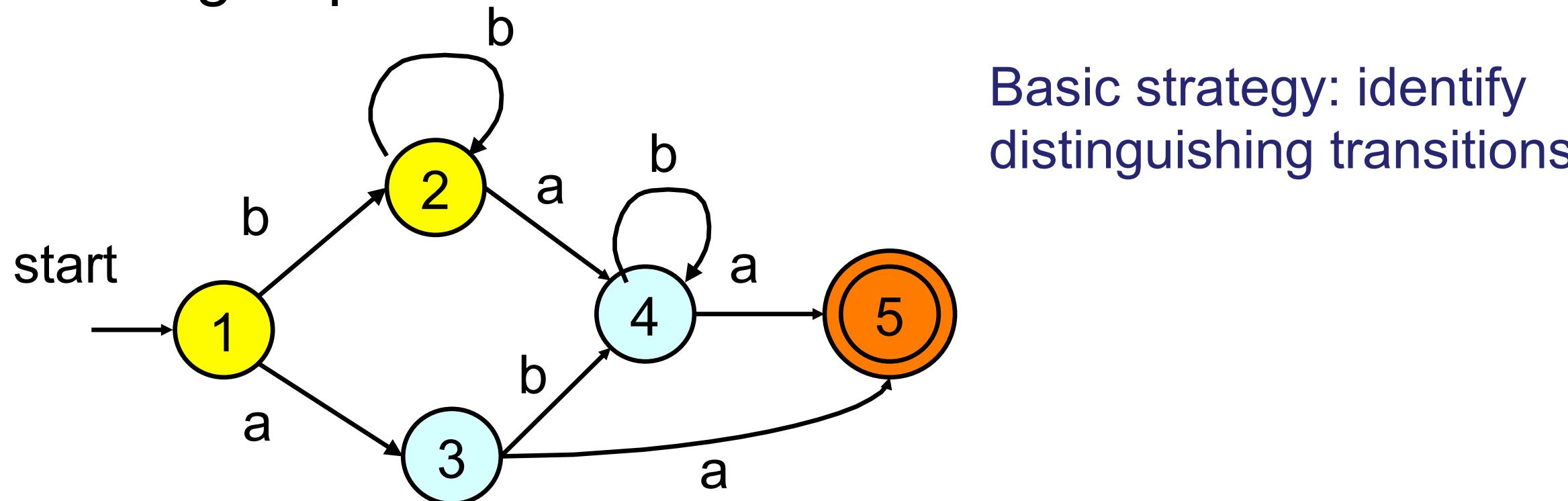
State Minimization

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



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



DFA Minimization

Overview of algorithm:

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

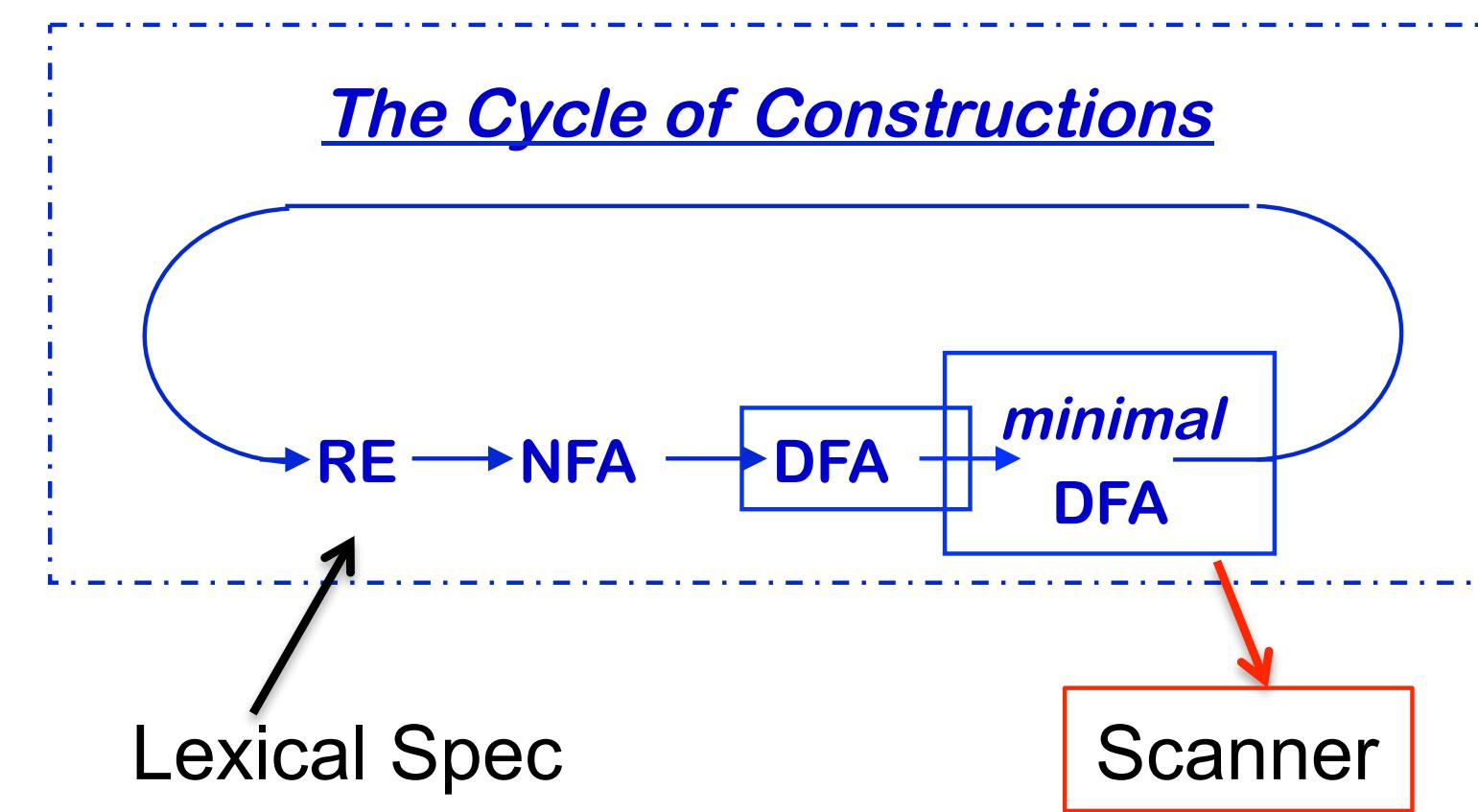
Initialize: two sets, accepting and rejecting

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.

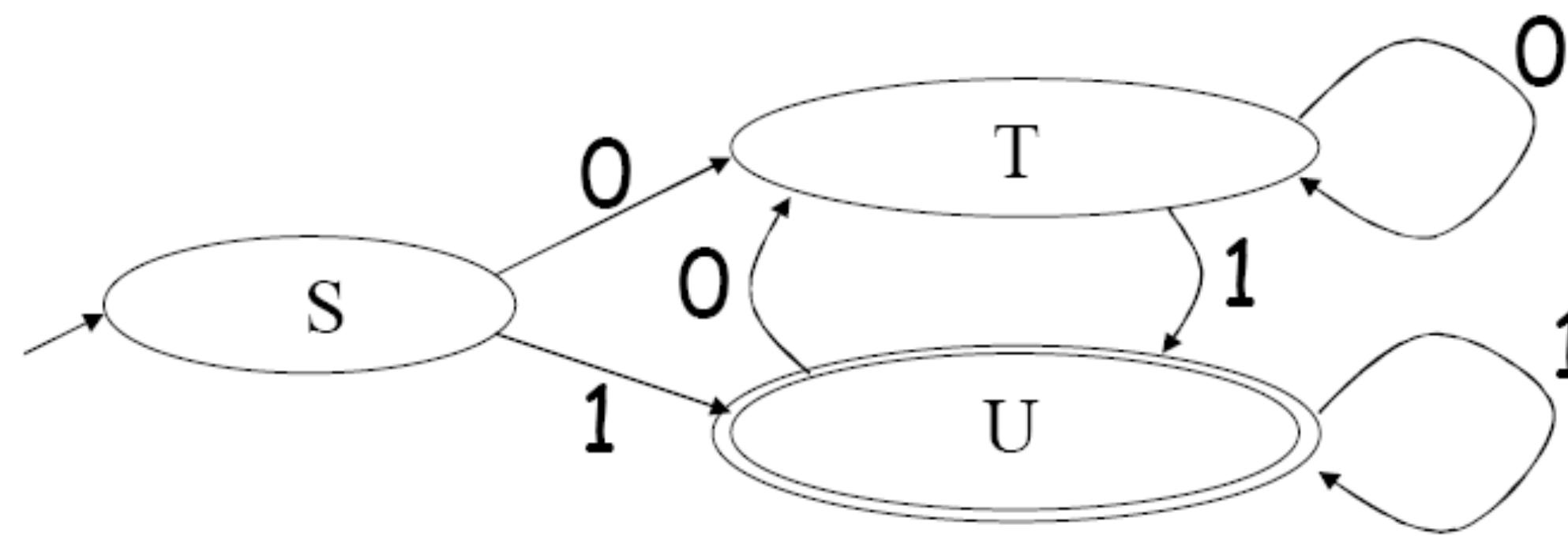
Similar to iterative fixpoint algorithms for dataflow analysis!



DFA Implementation

- A DFA can be implemented by a 2D table T
 - One dimension is “states”
 - Other dimension is “input symbol”
 - For every transition $S_i \rightarrow^a S_k$ 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

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)**

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?

- Example

Implementing Maximal Munch

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

Implementing Maximal Munch

T_Do

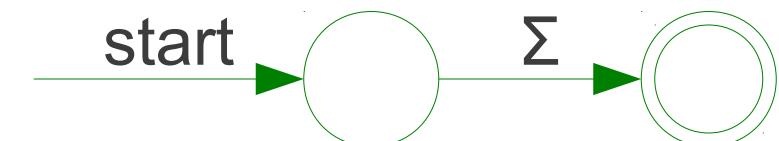
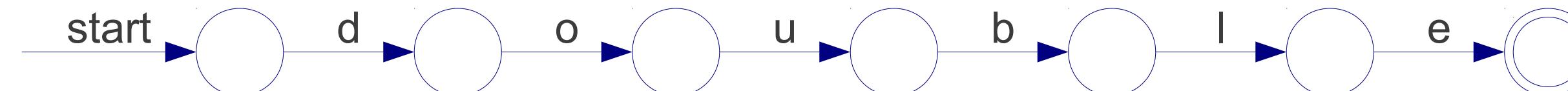
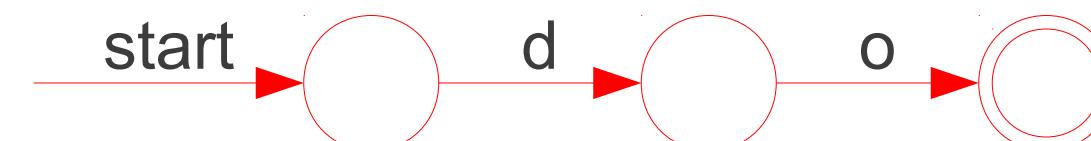
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

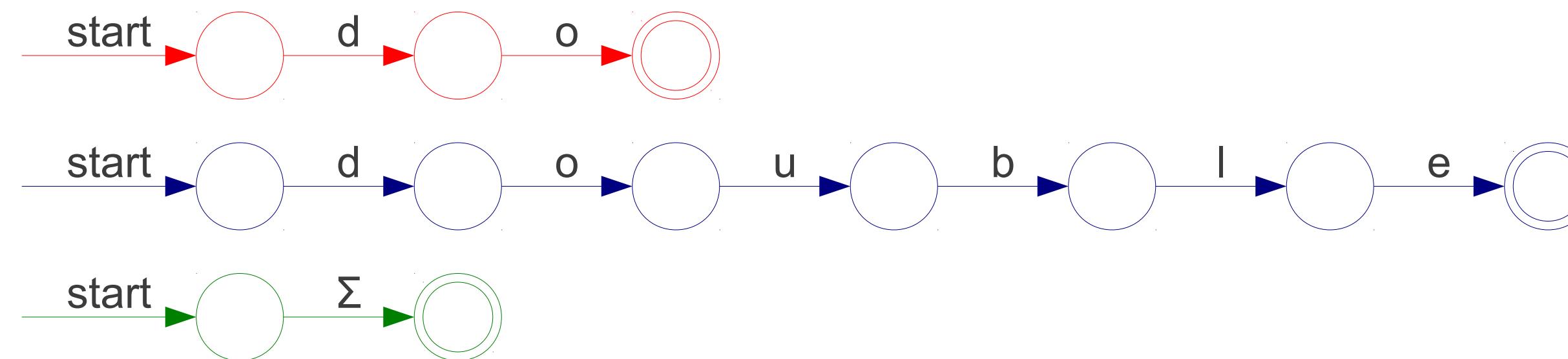
T_Double

T_Mystery

do

double

[A-Za-z]



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

Implementing Maximal Munch

T_Do

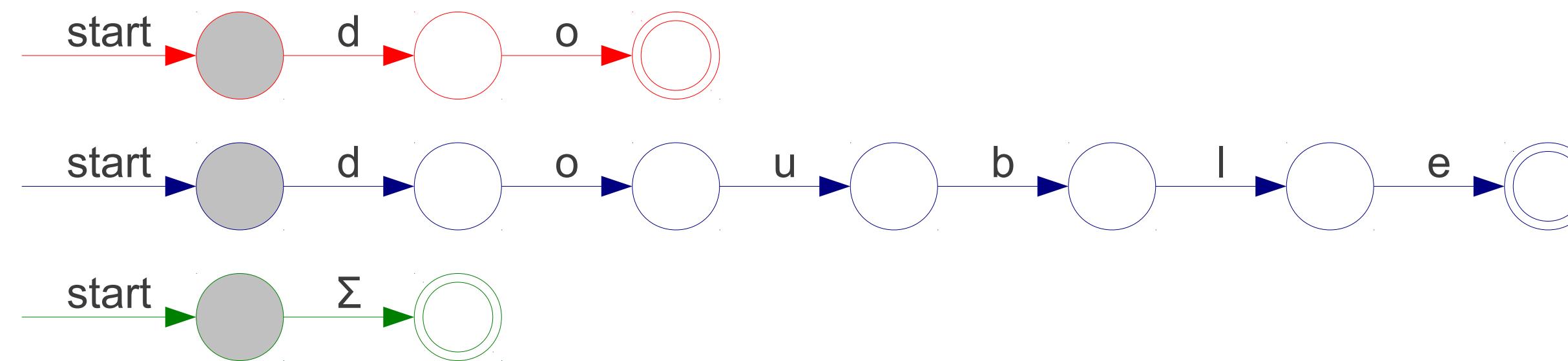
T_Double

T_Mystery

do

double

[A-Za-z]



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

Implementing Maximal Munch

T_Do

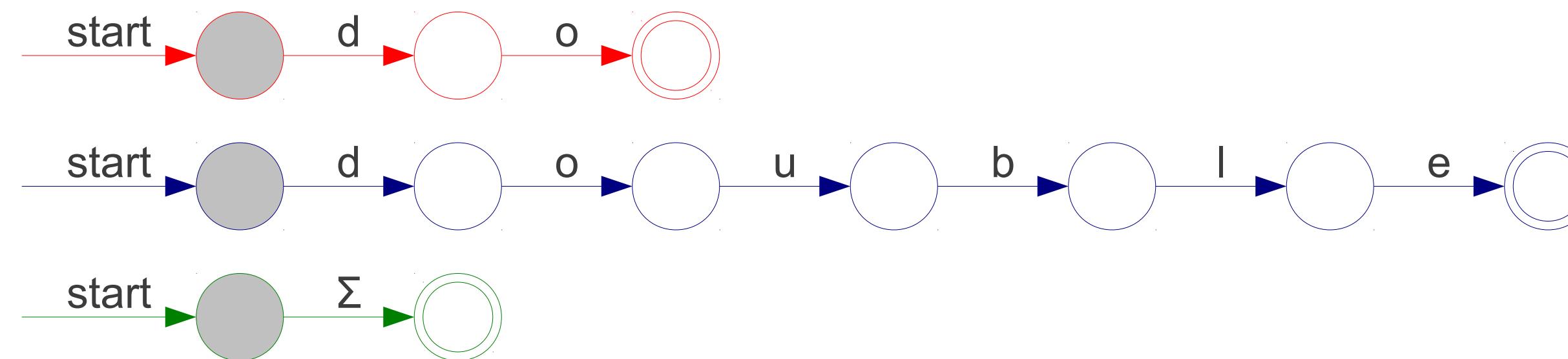
T_Double

T_Mystery

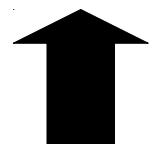
do

double

[A-Za-z]



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



Implementing Maximal Munch

T_Do

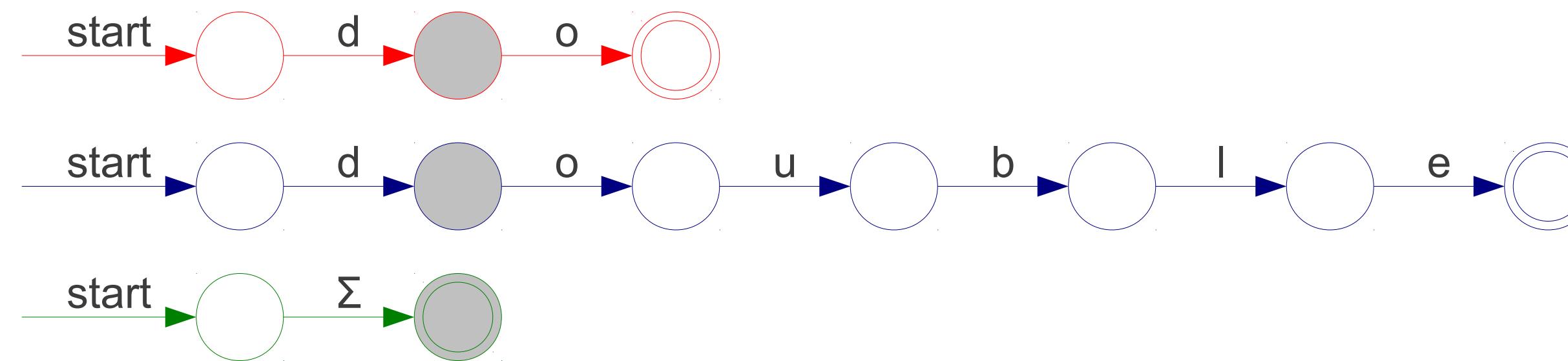
T_Double

T_Mystery

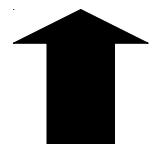
do

double

[A-Za-z]



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



Implementing Maximal Munch

T_Do

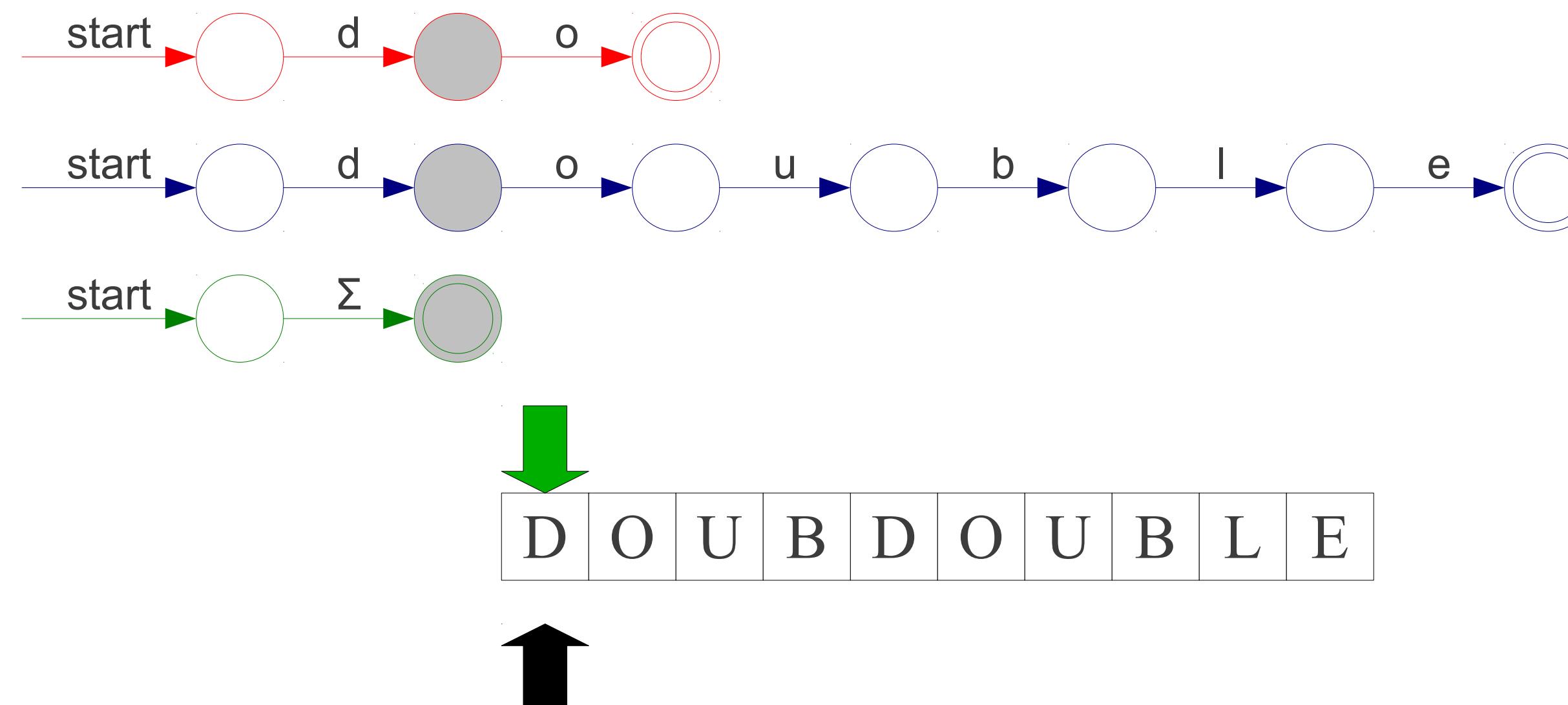
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

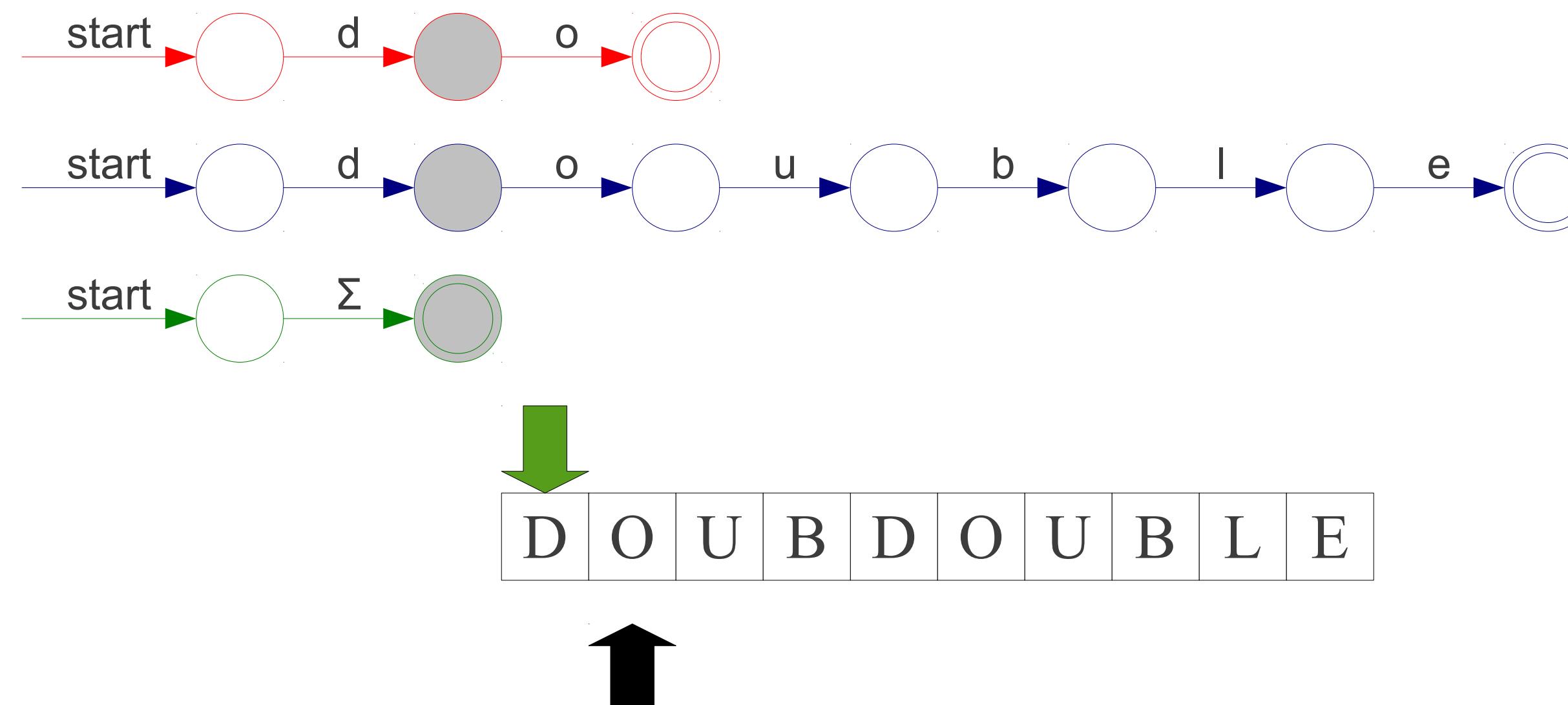
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

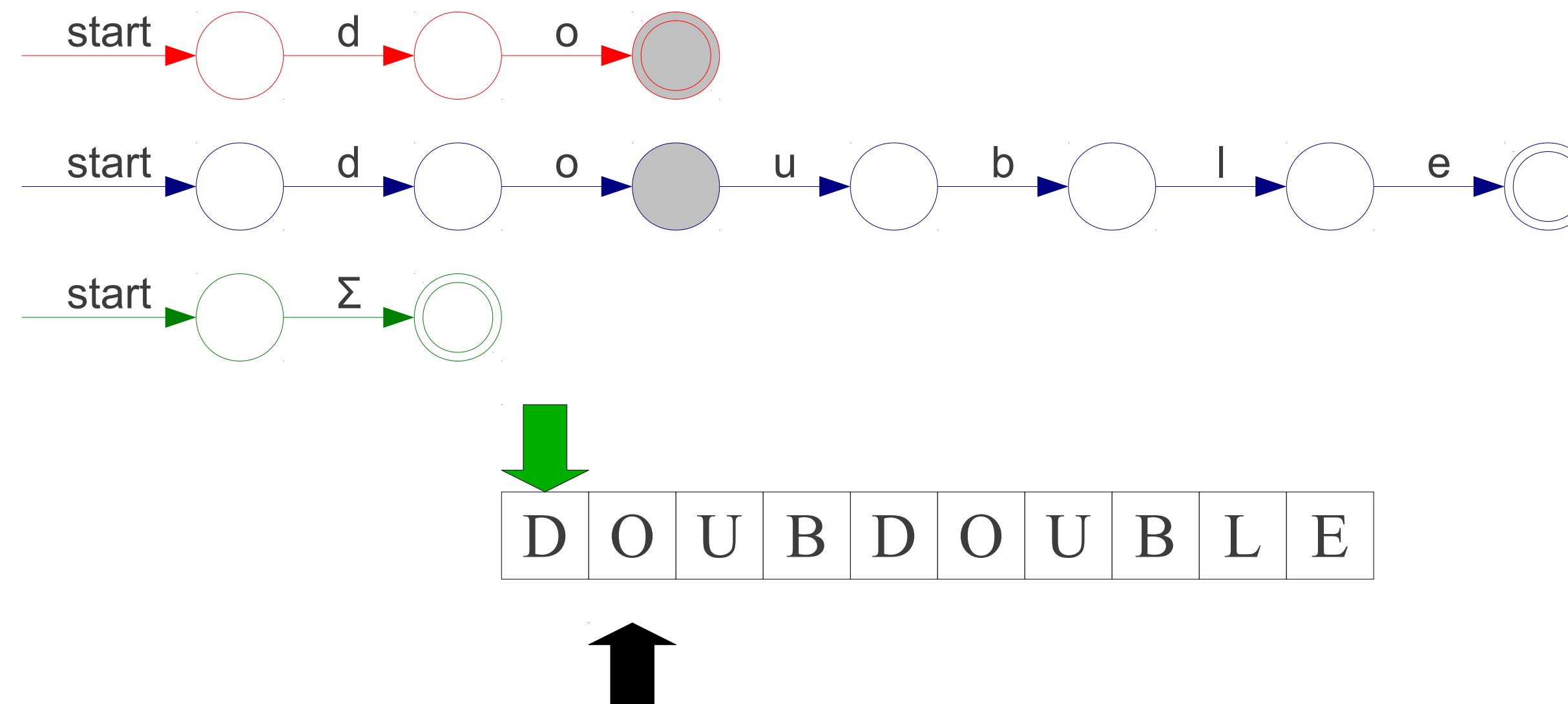
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

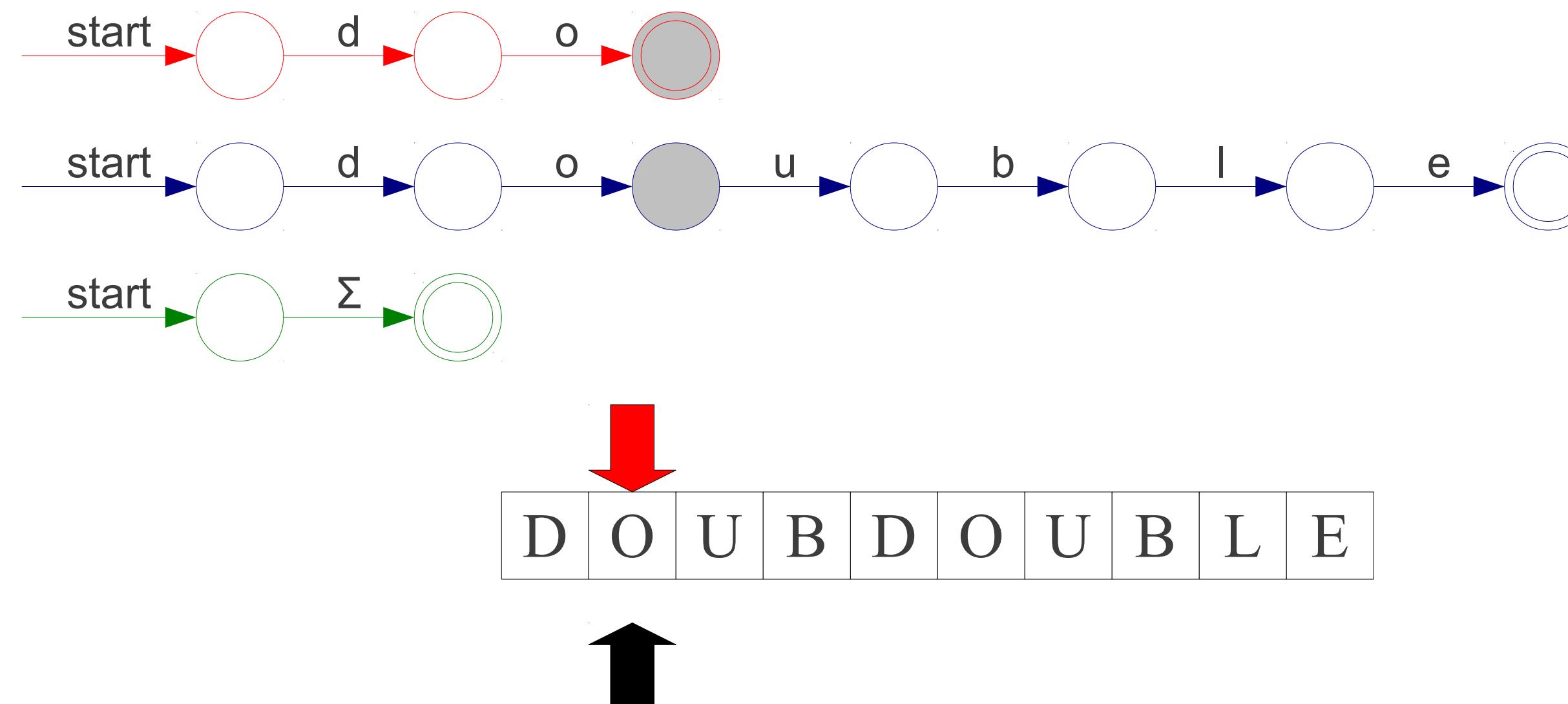
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

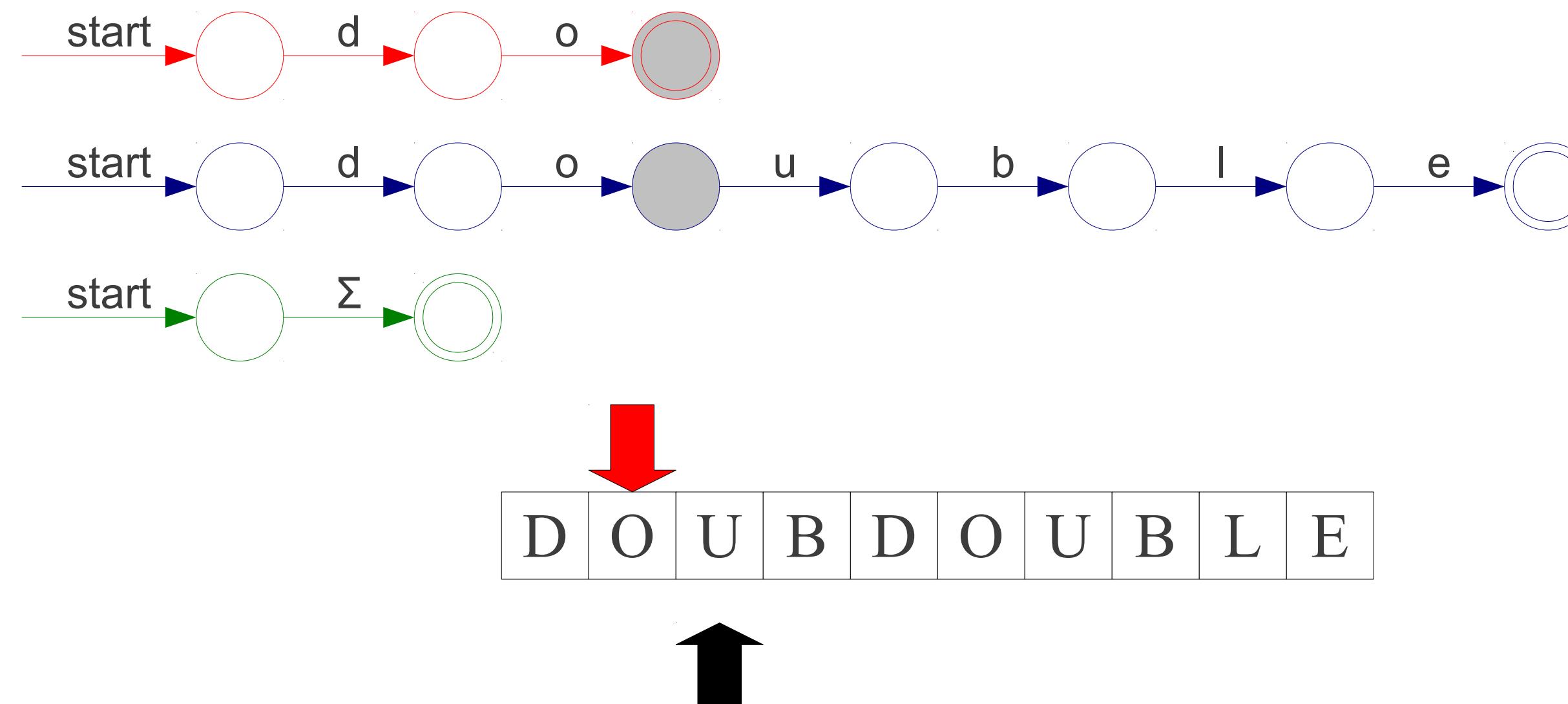
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

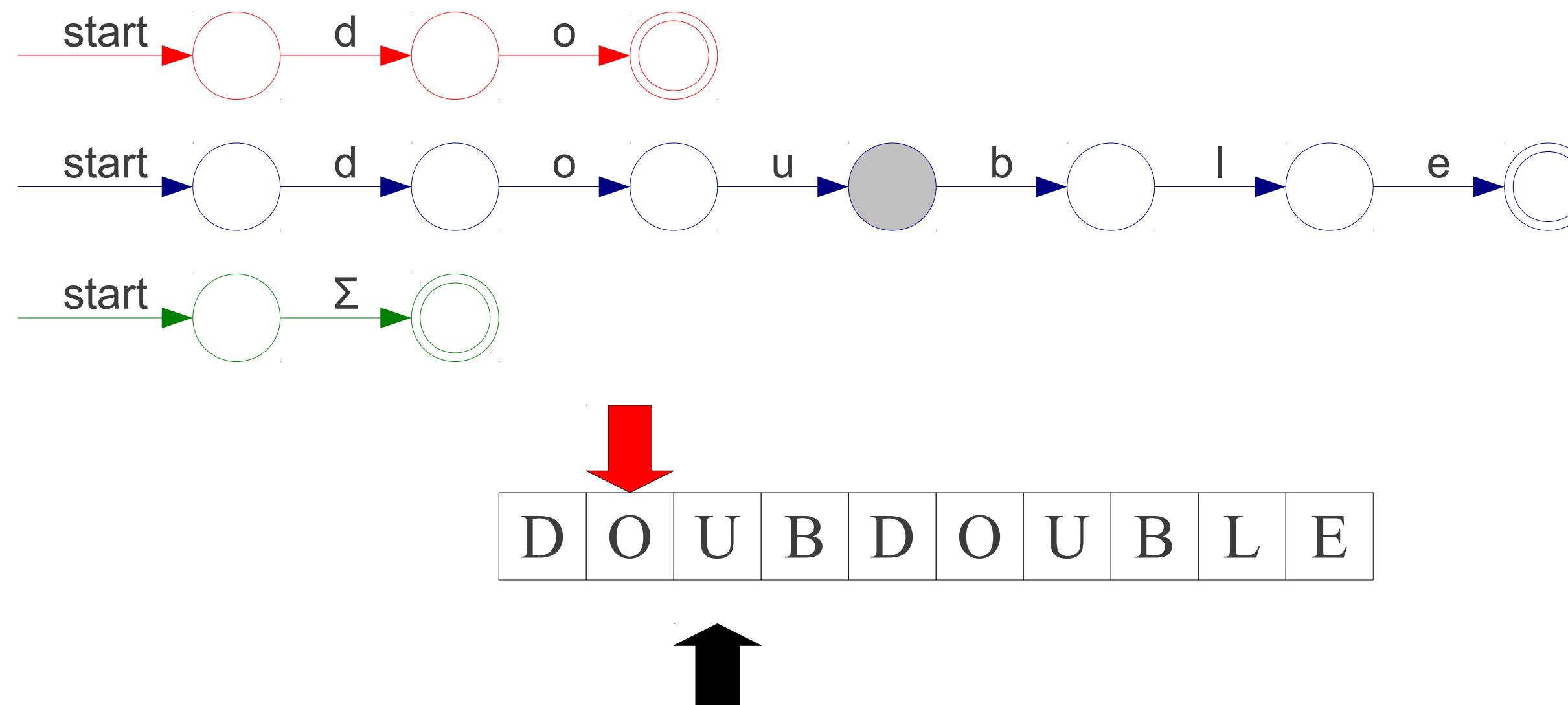
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

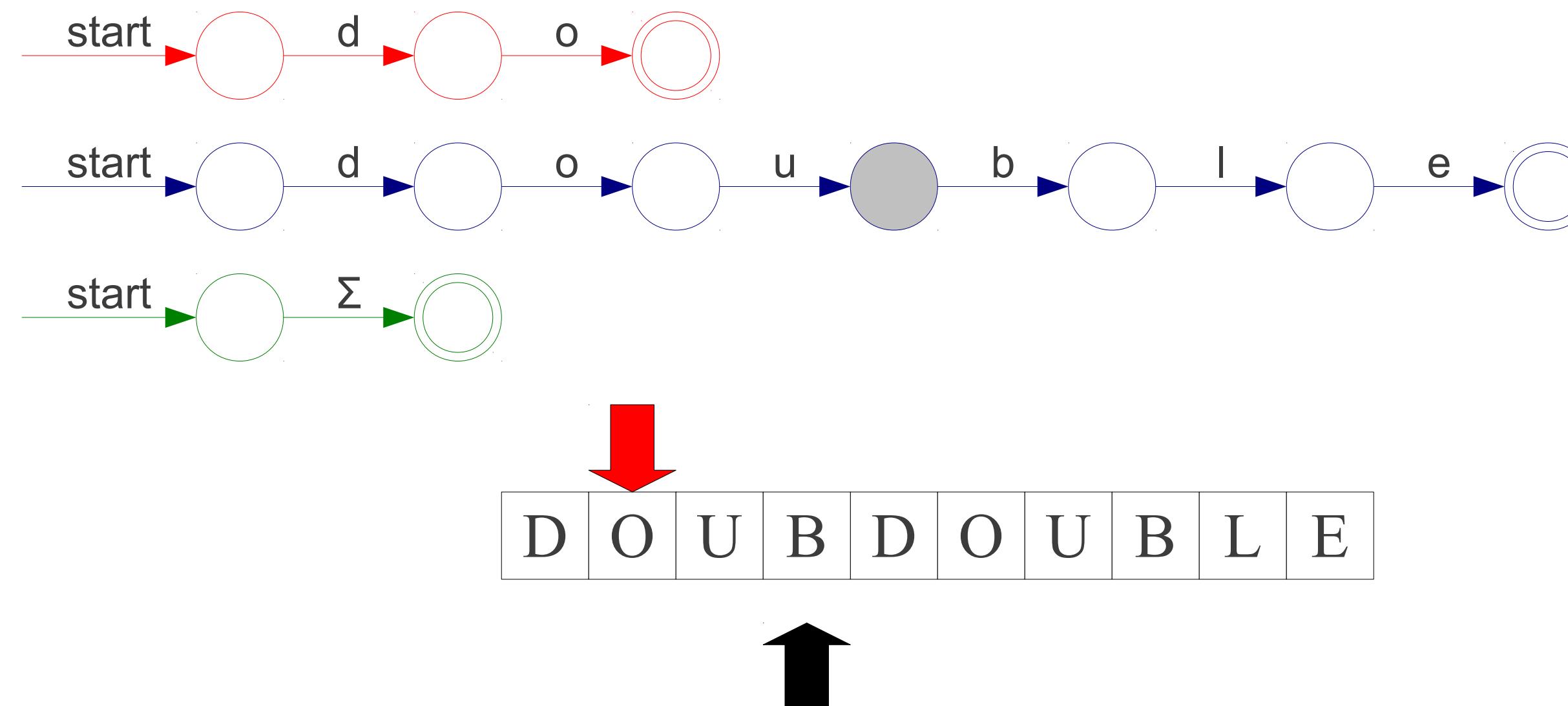
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

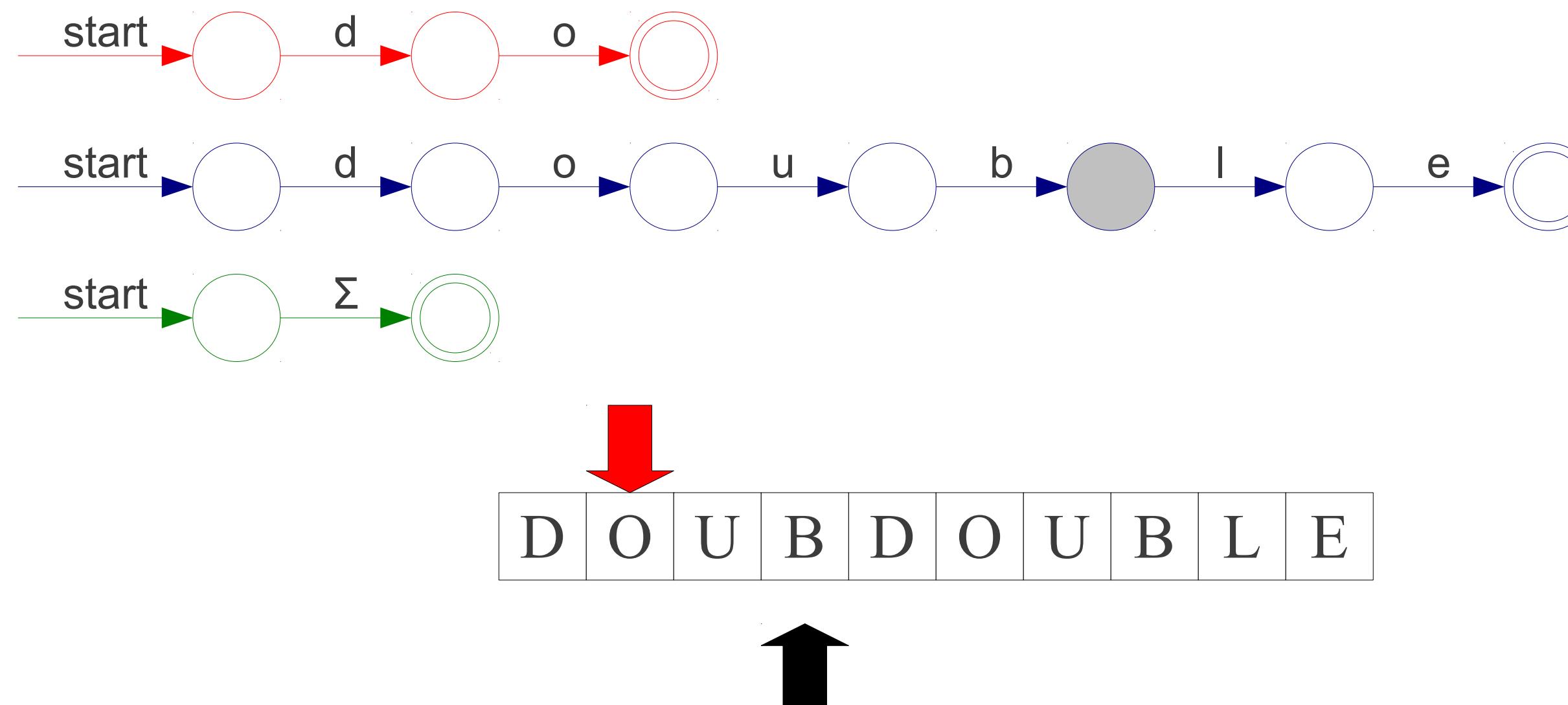
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

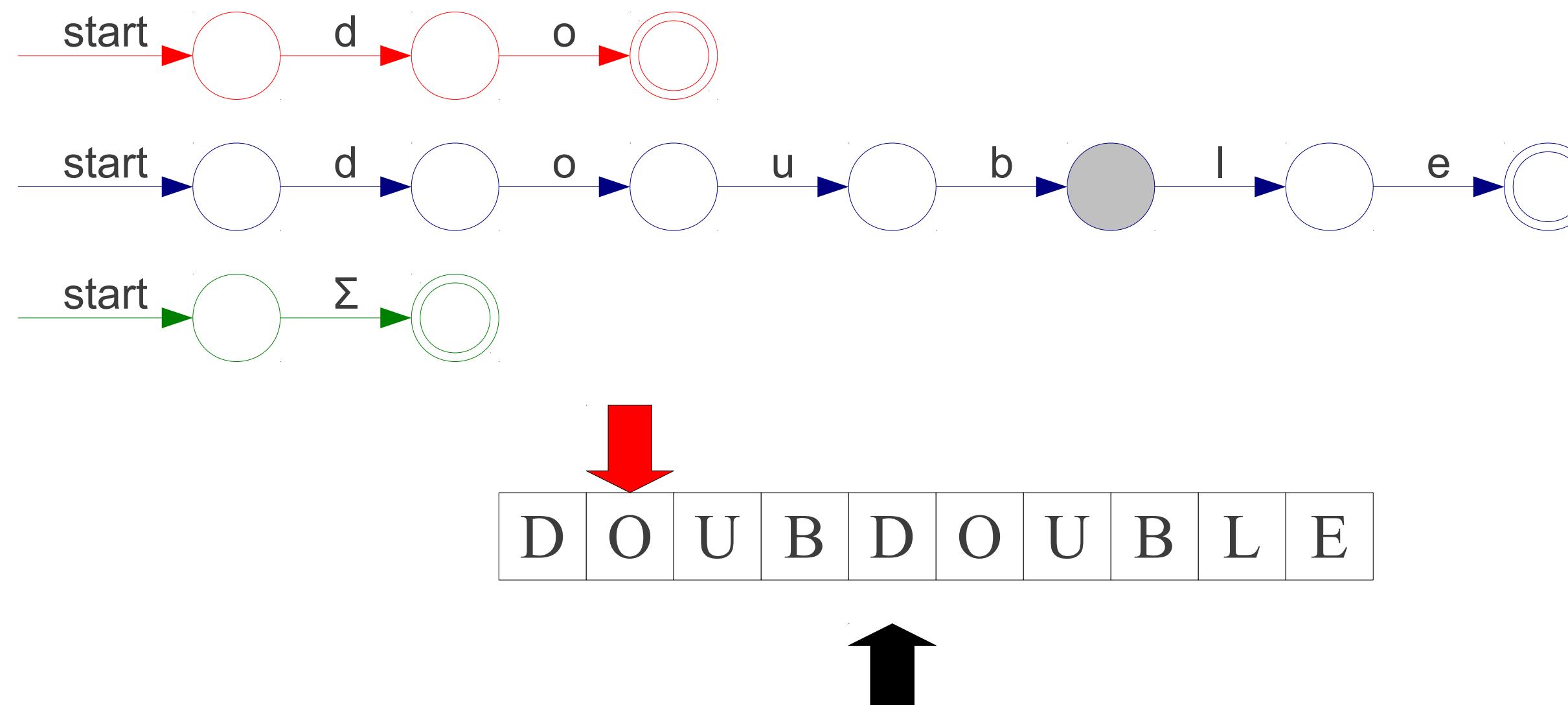
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

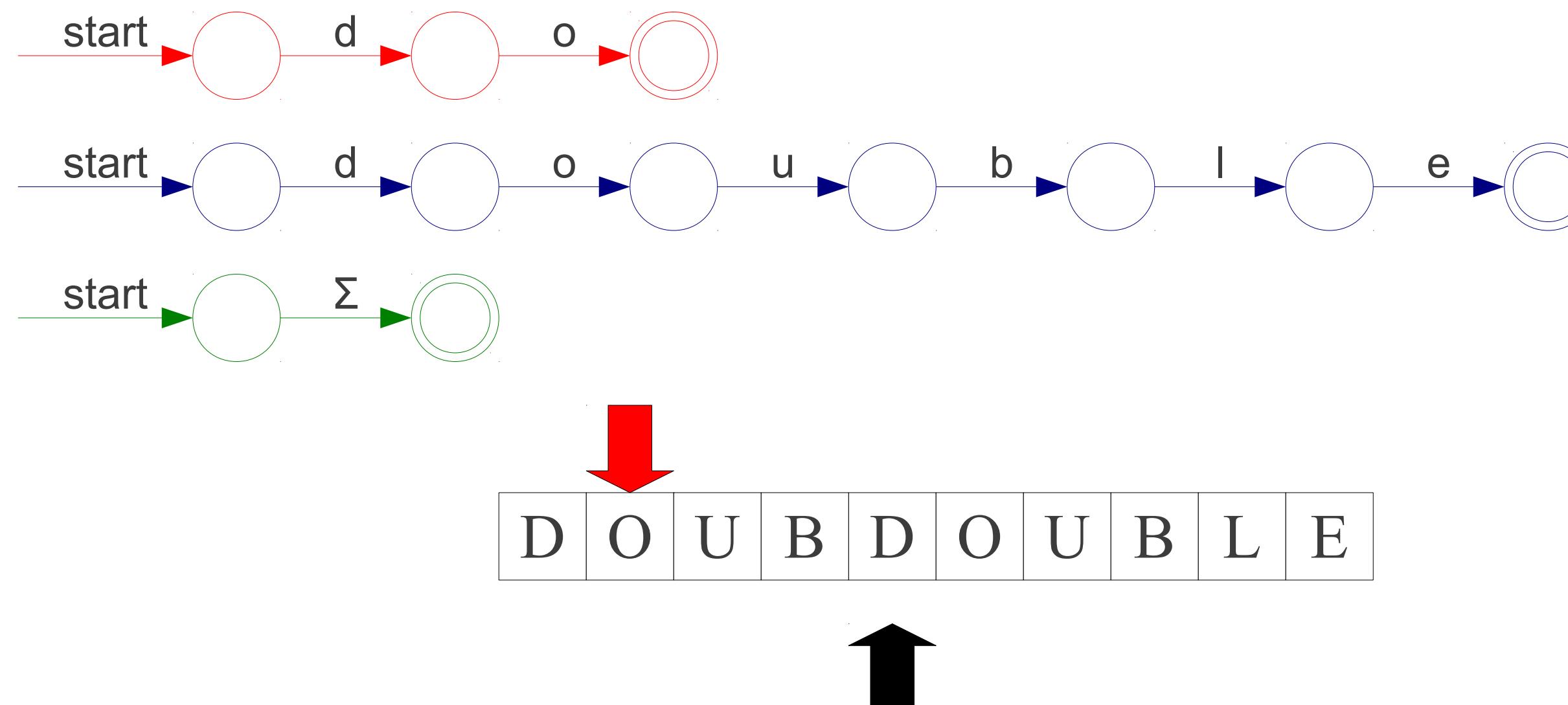
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

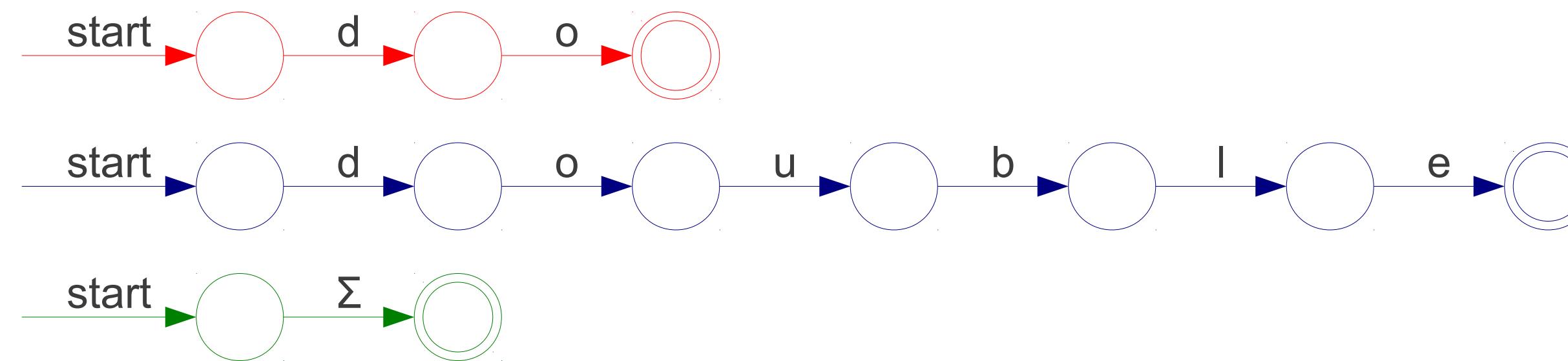
T_Double

T_Mystery

do

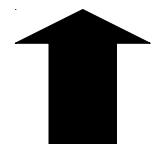
double

[A-Za-z]



D|O

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



Implementing Maximal Munch

T_Do

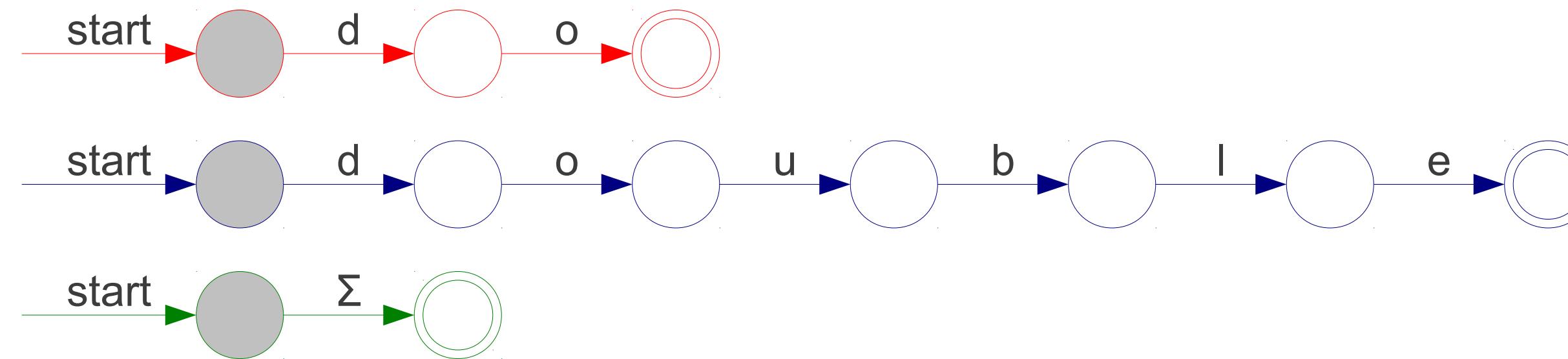
T_Double

T_Mystery

do

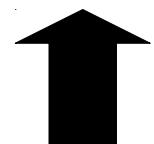
double

[A-Za-z]



D|O

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



Implementing Maximal Munch

T_Do

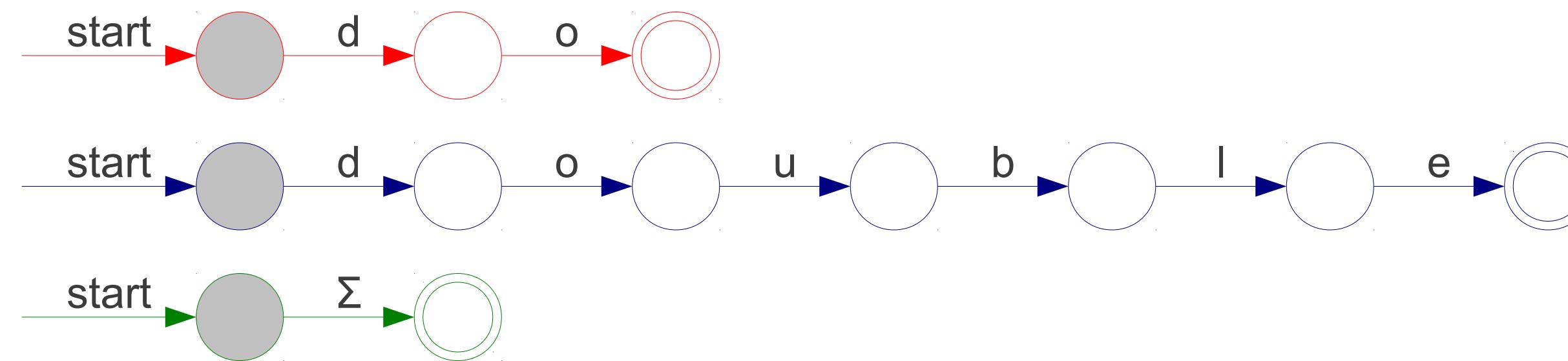
T_Double

T_Mystery

do

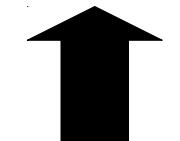
double

[A-Za-z]



D|O

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



Implementing Maximal Munch

T_Do

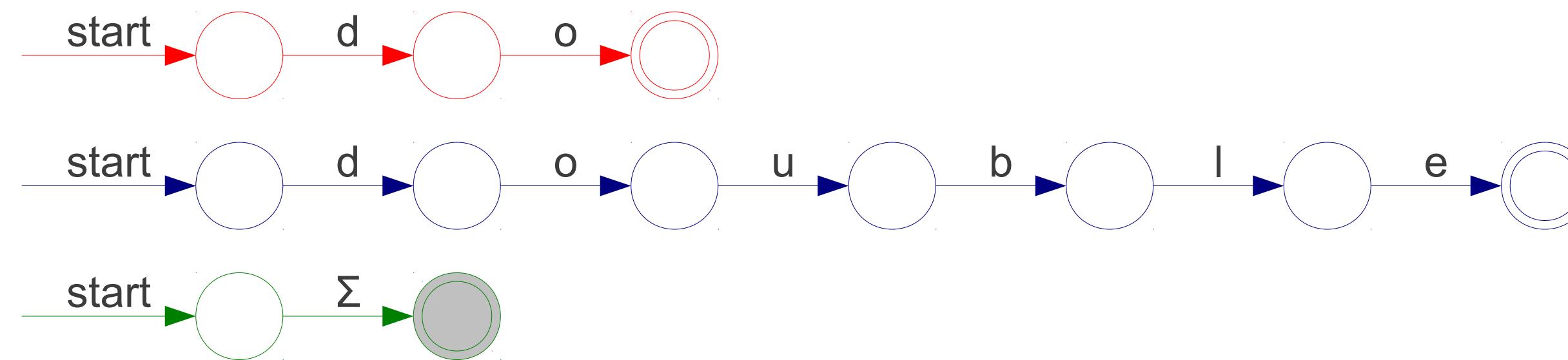
T_Double

T_Mystery

do

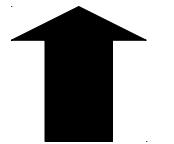
double

[A-Za-z]



D|O

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



Implementing Maximal Munch

T_Do

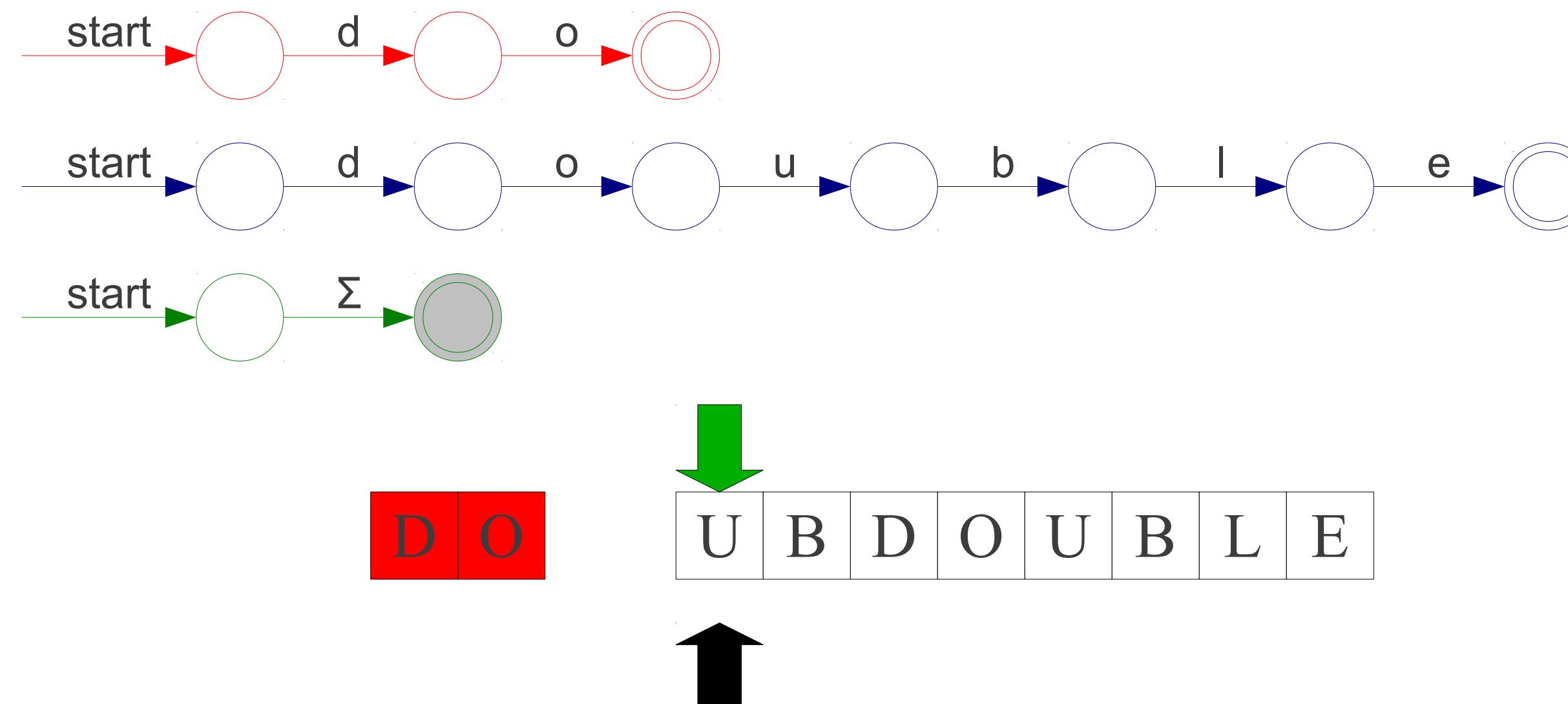
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

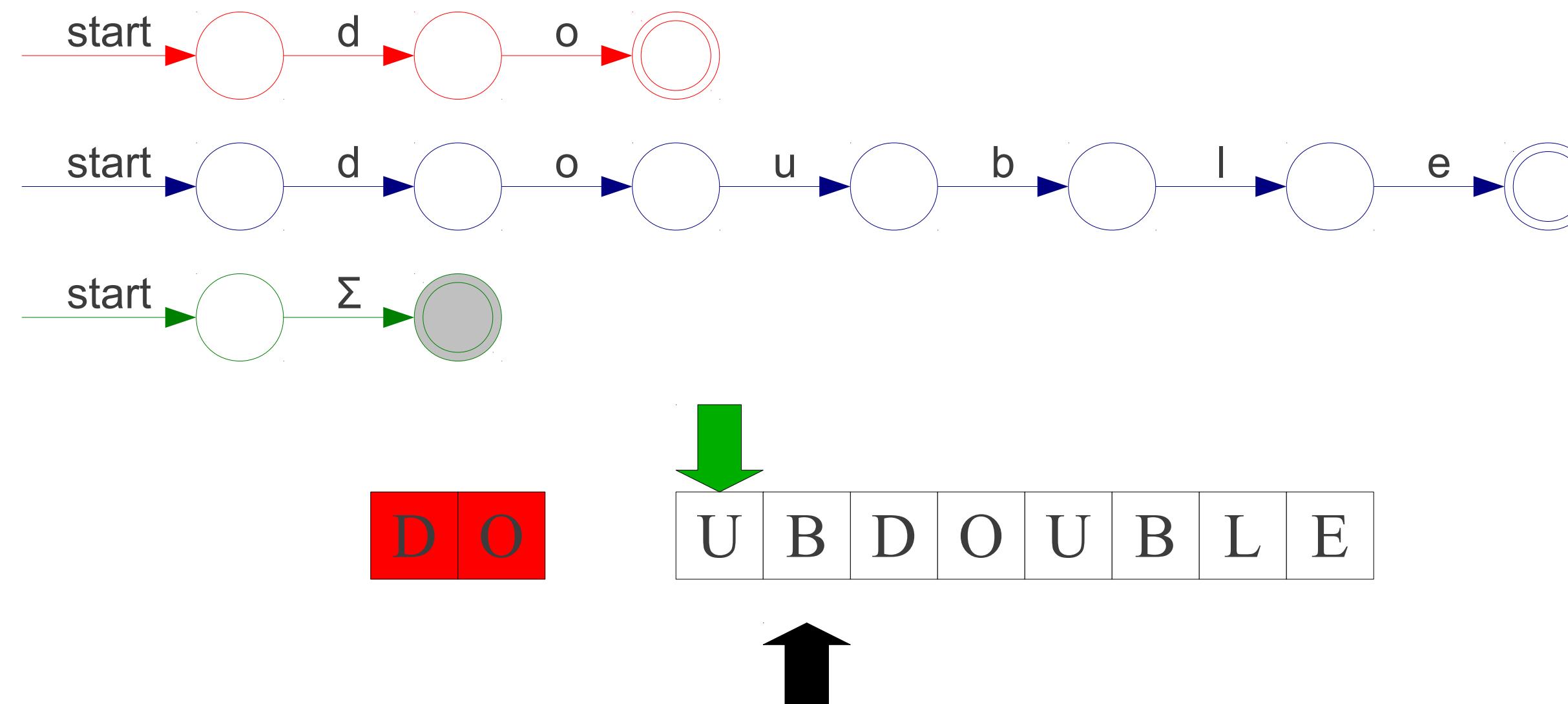
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

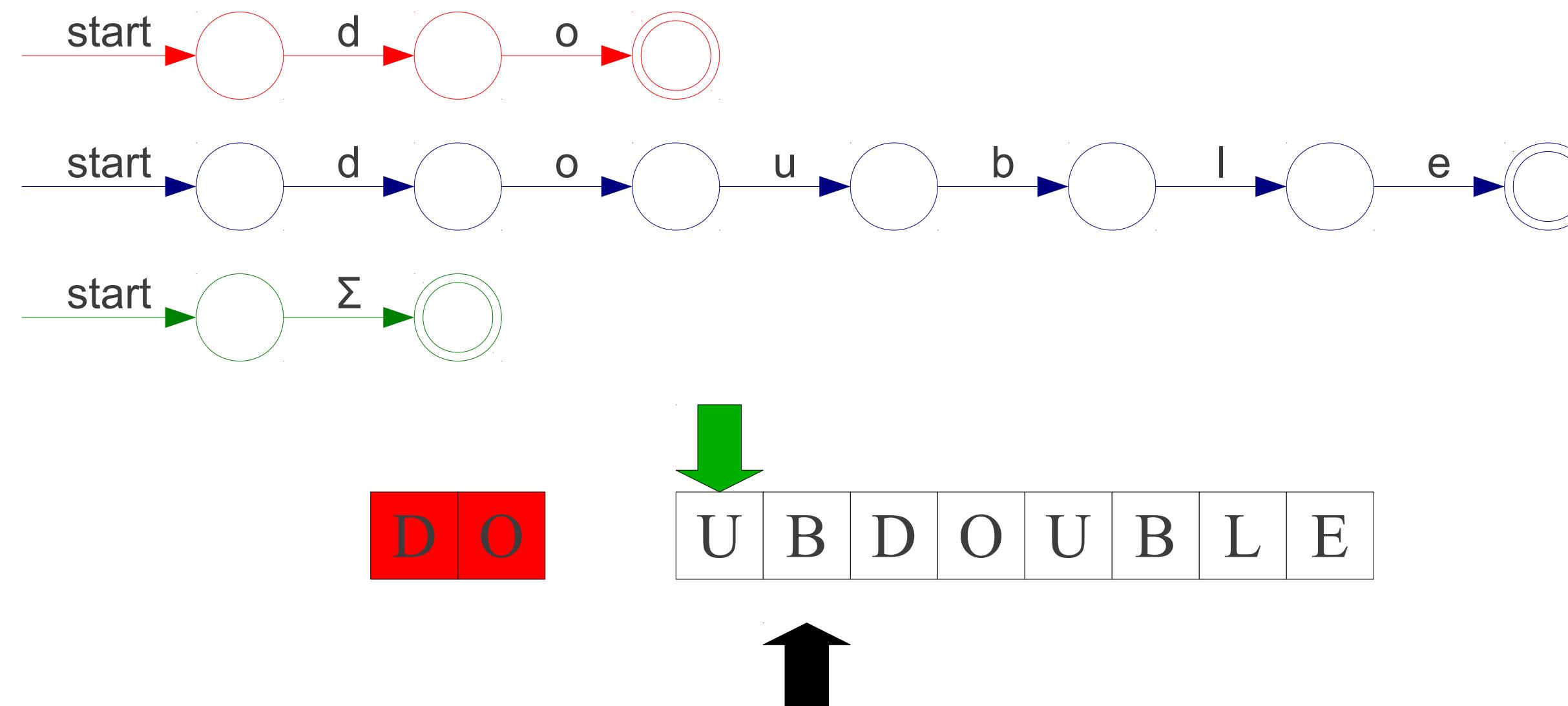
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

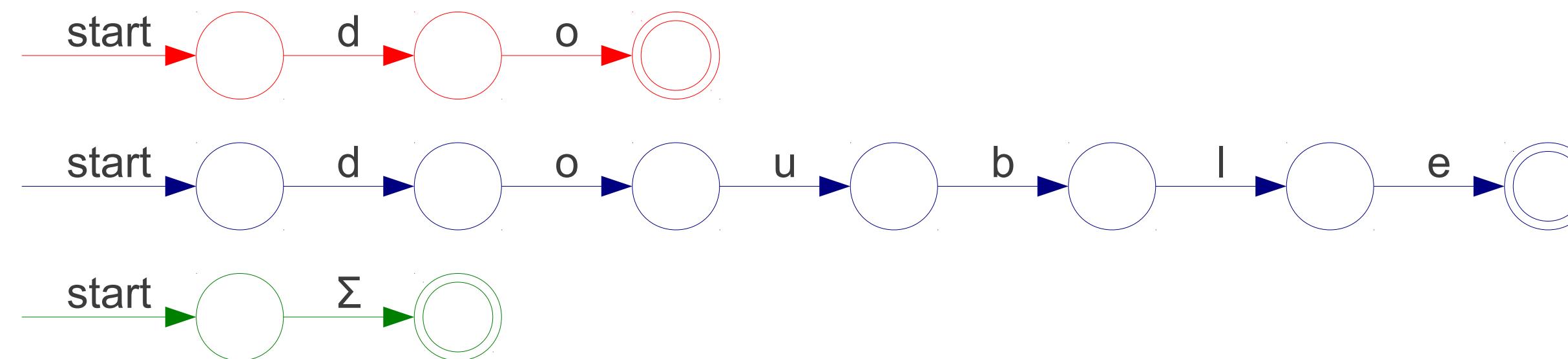
T_Double

T_Mystery

do

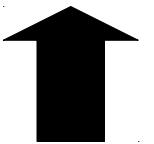
double

[A-Za-z]



D O U

B D O U B L E



Implementing Maximal Munch

T_Do

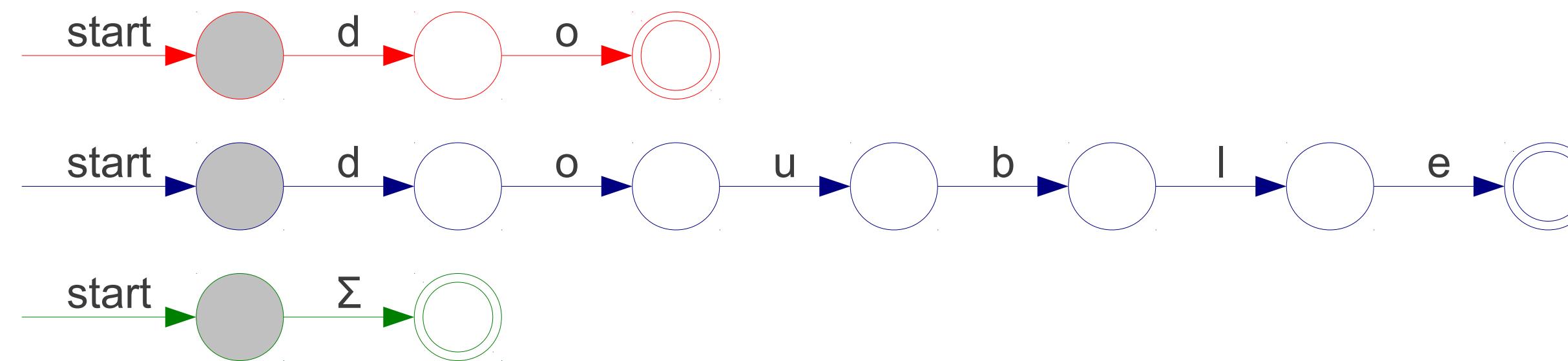
T_Double

T_Mystery

do

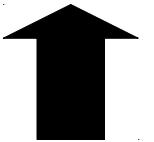
double

[A-Za-z]



D | O U

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



Implementing Maximal Munch

T_Do

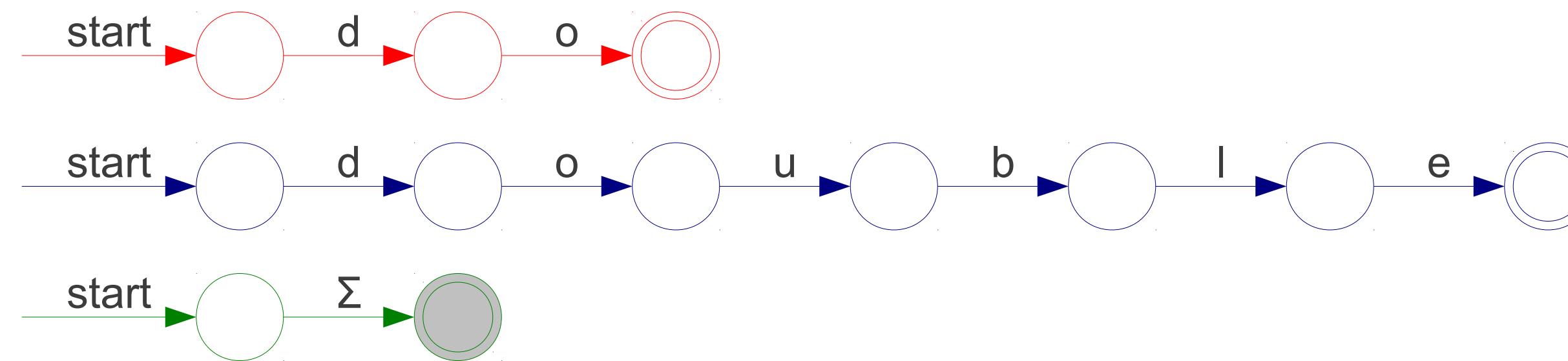
T_Double

T_Mystery

do

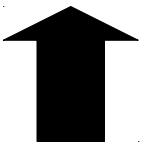
double

[A-Za-z]



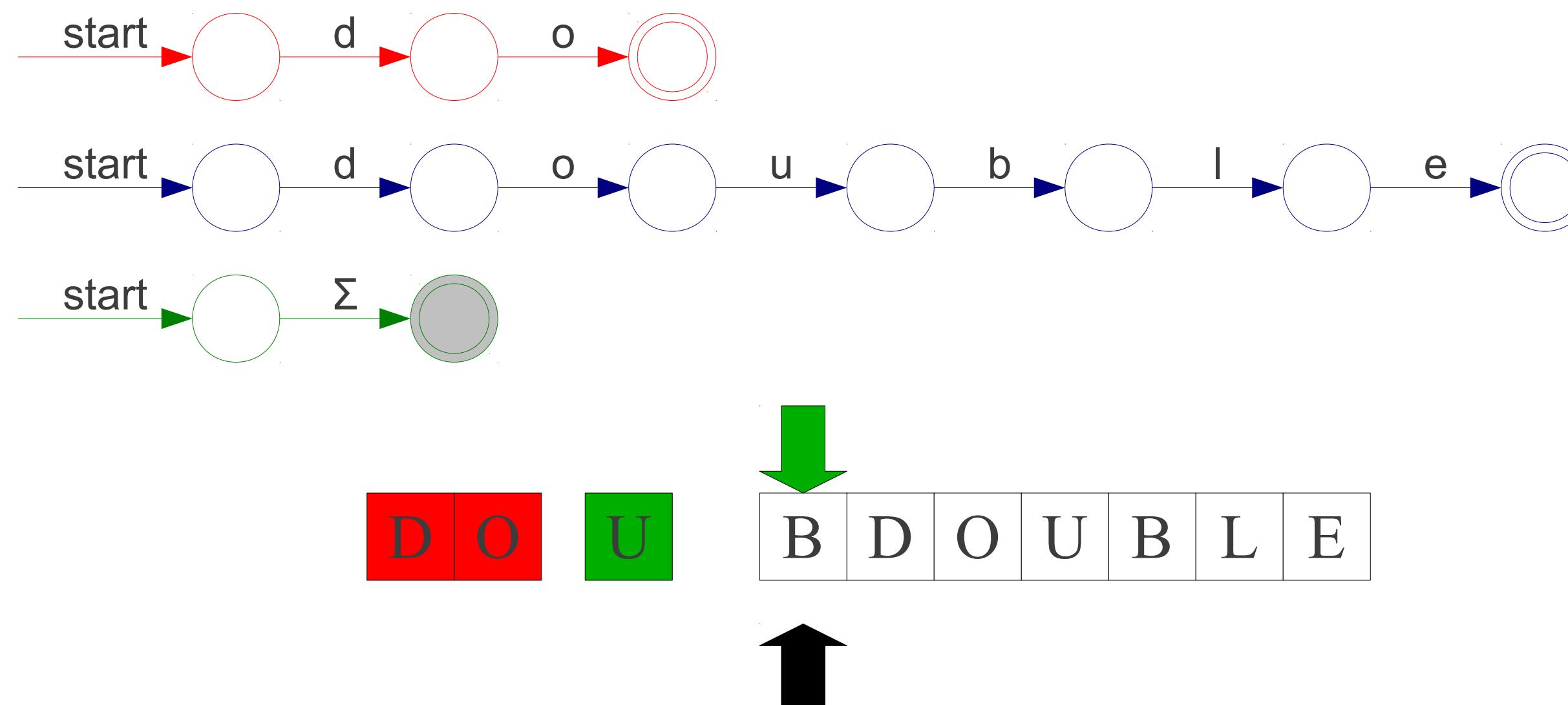
D O U

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

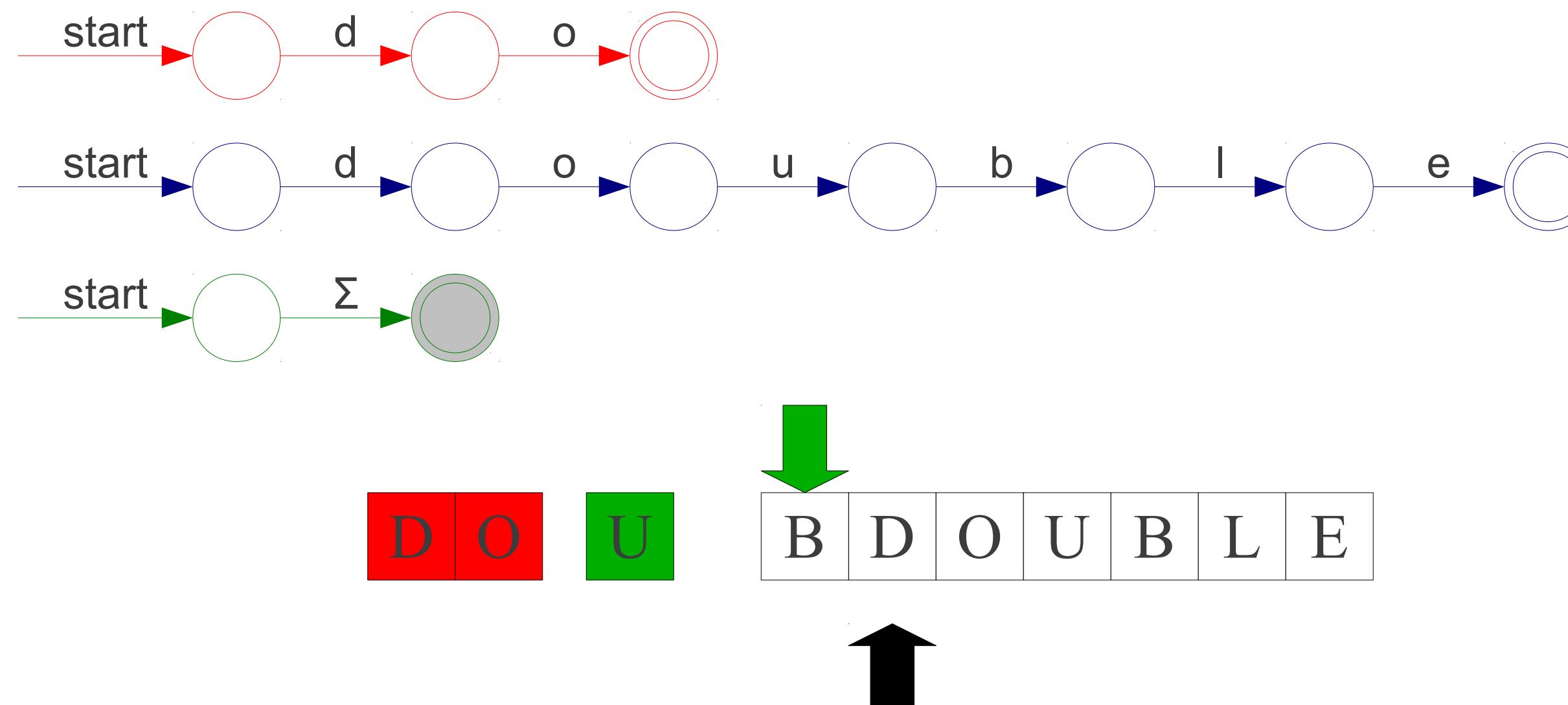
T_Double

T_Mystery

do

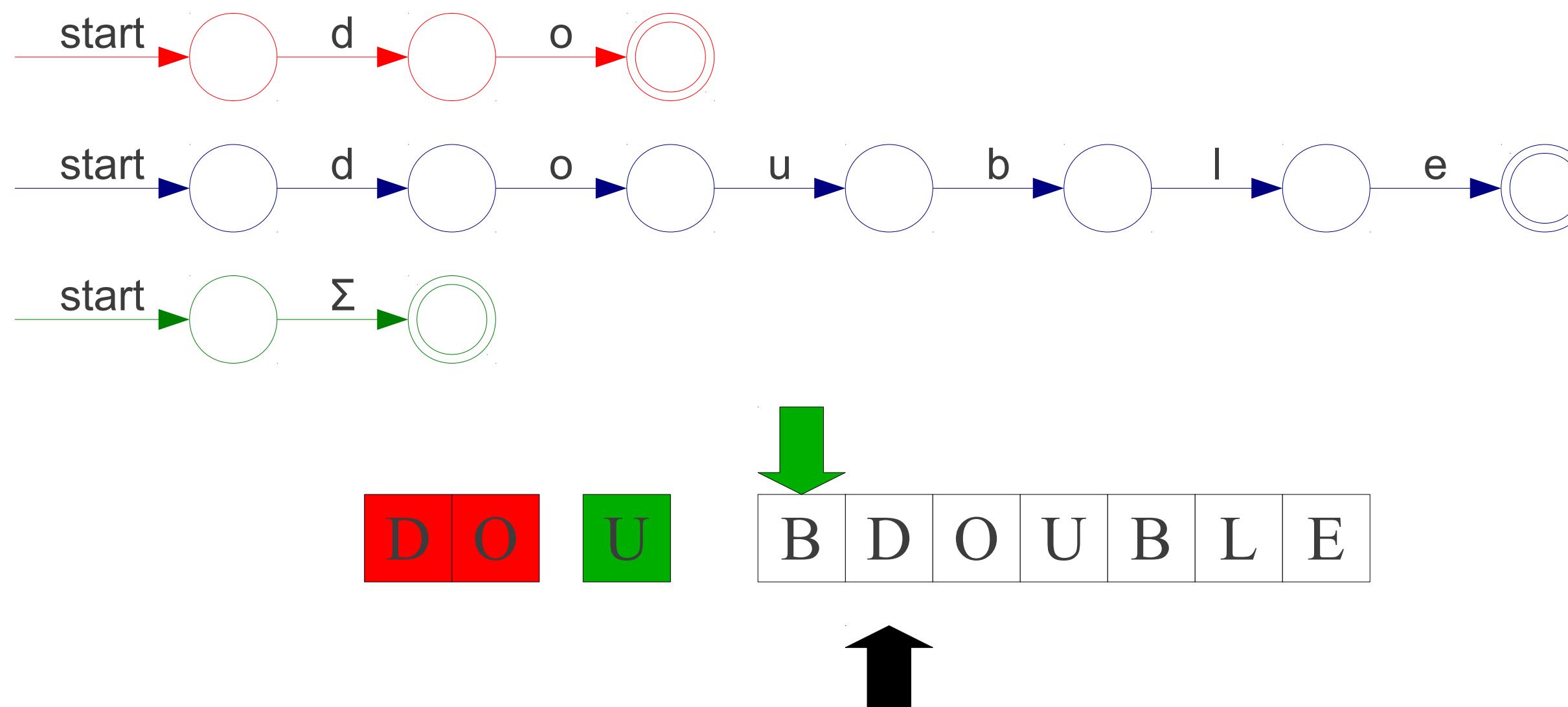
double

[A-Za-z]



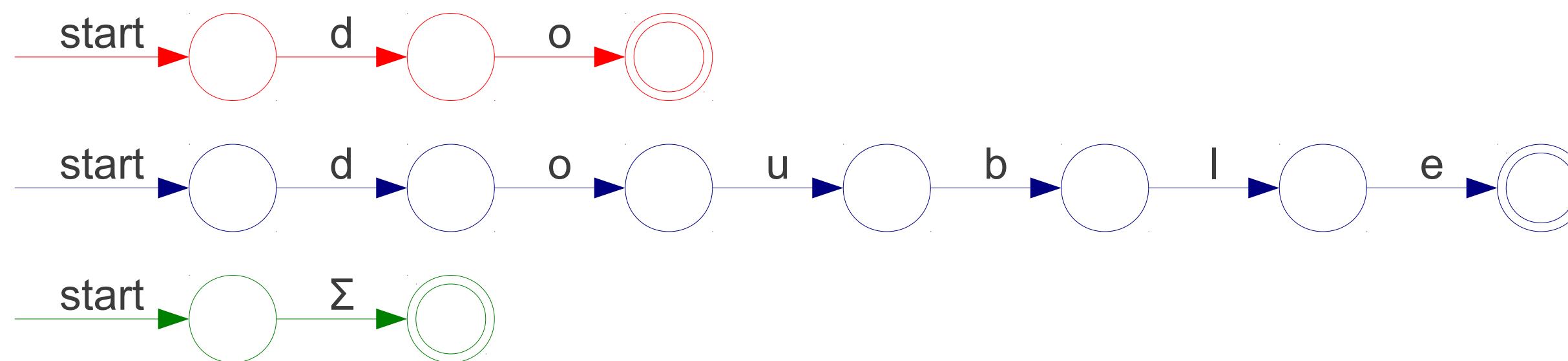
Implementing Maximal Munch

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]

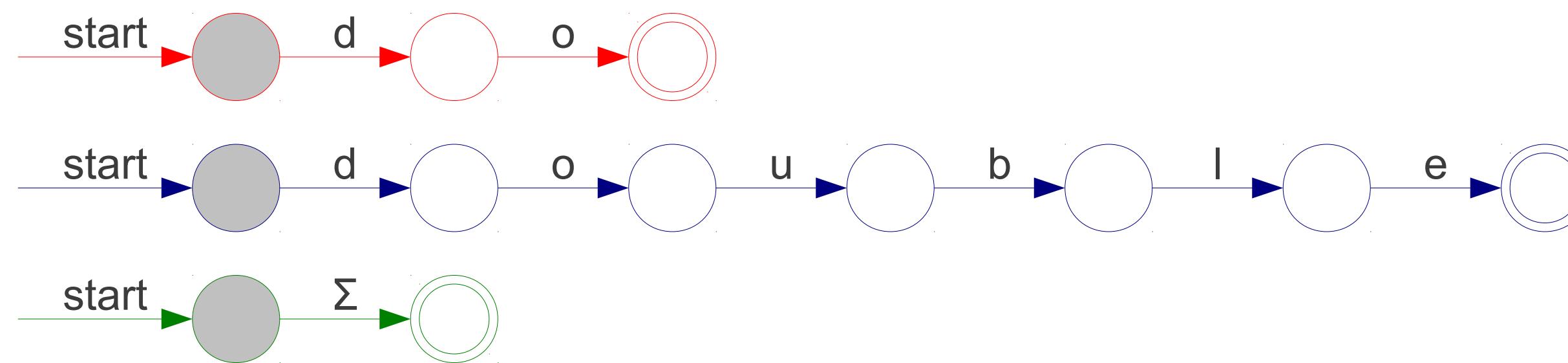


D | O U B D | O | U | B | L | E



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

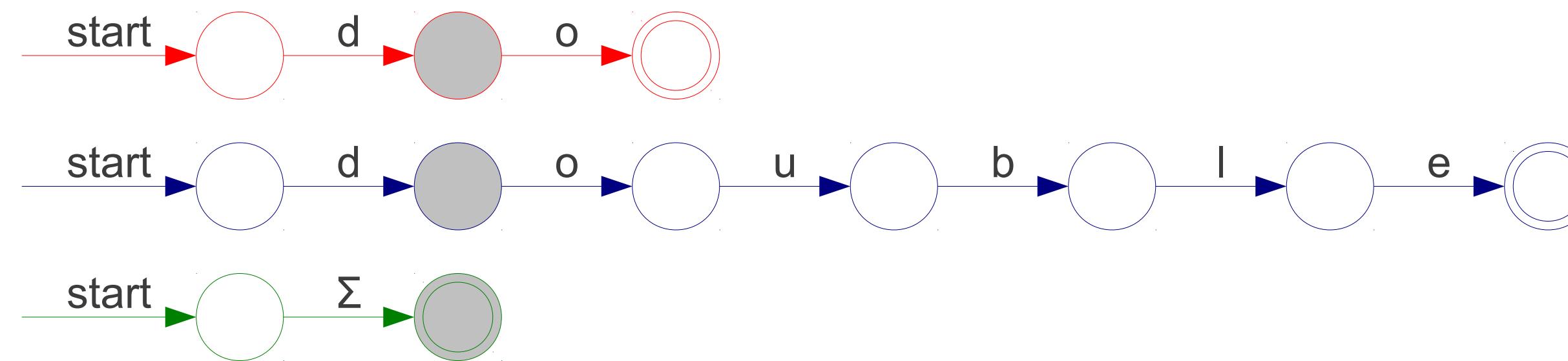
T_Double

T_Mystery

do

double

[A-Za-z]

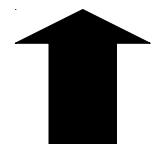


D O

U

B

D O U B L E



Implementing Maximal Munch

T_Do

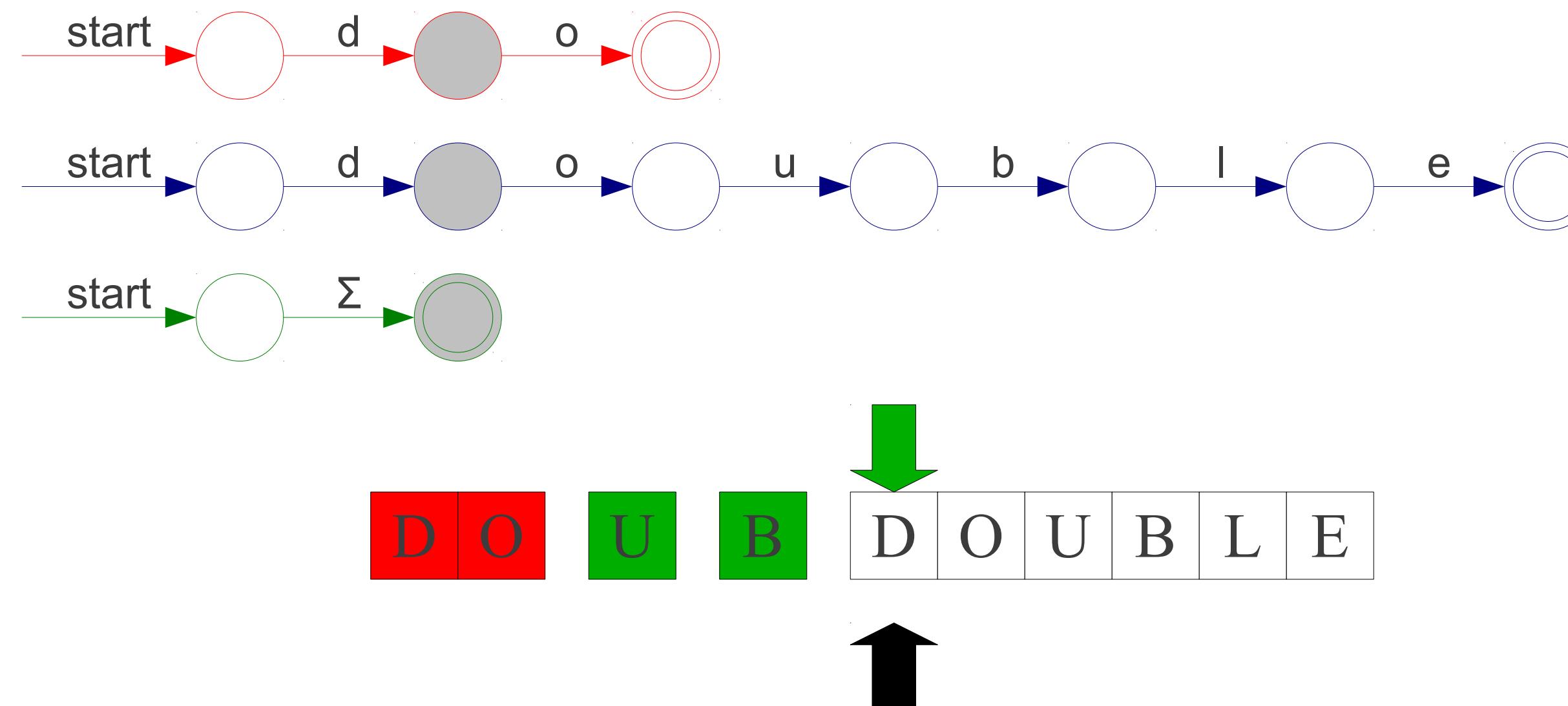
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

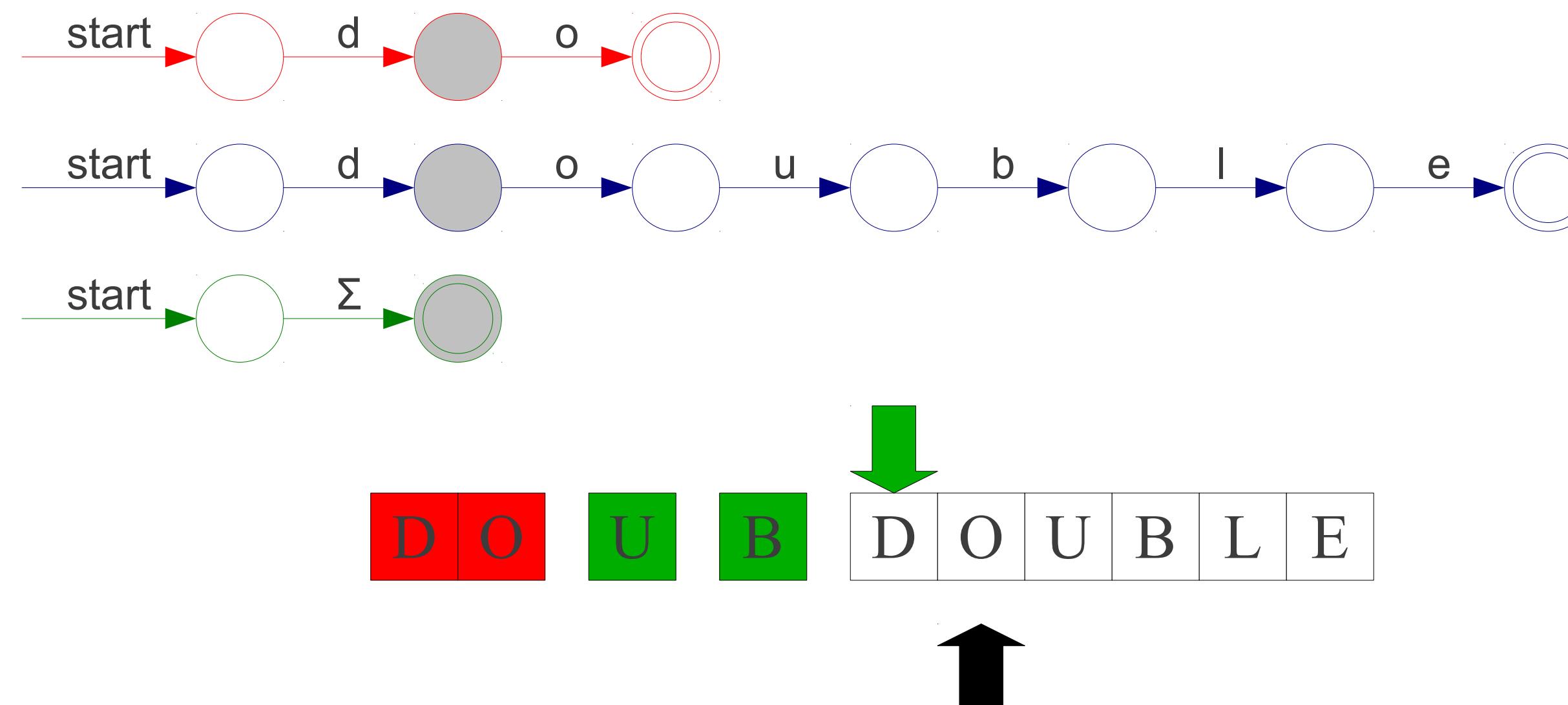
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

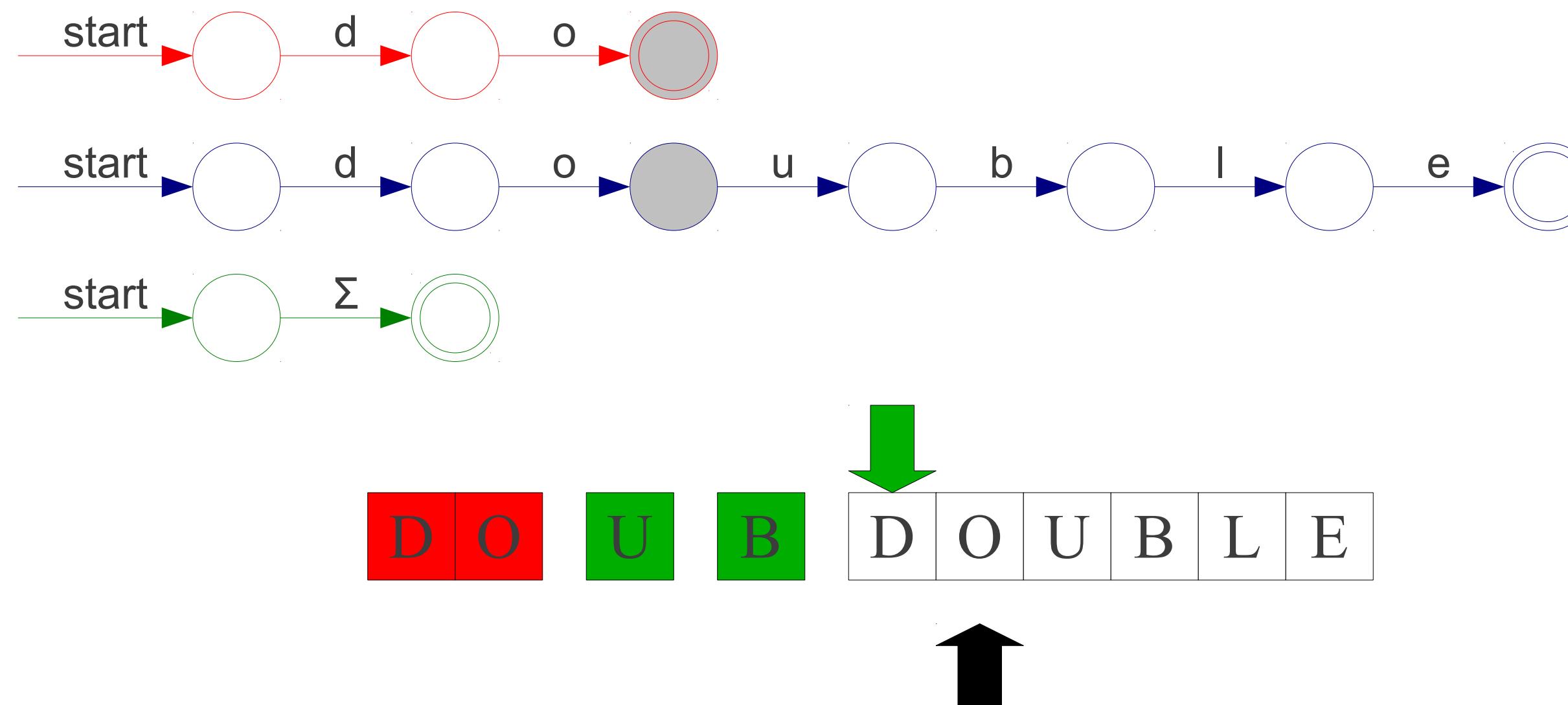
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

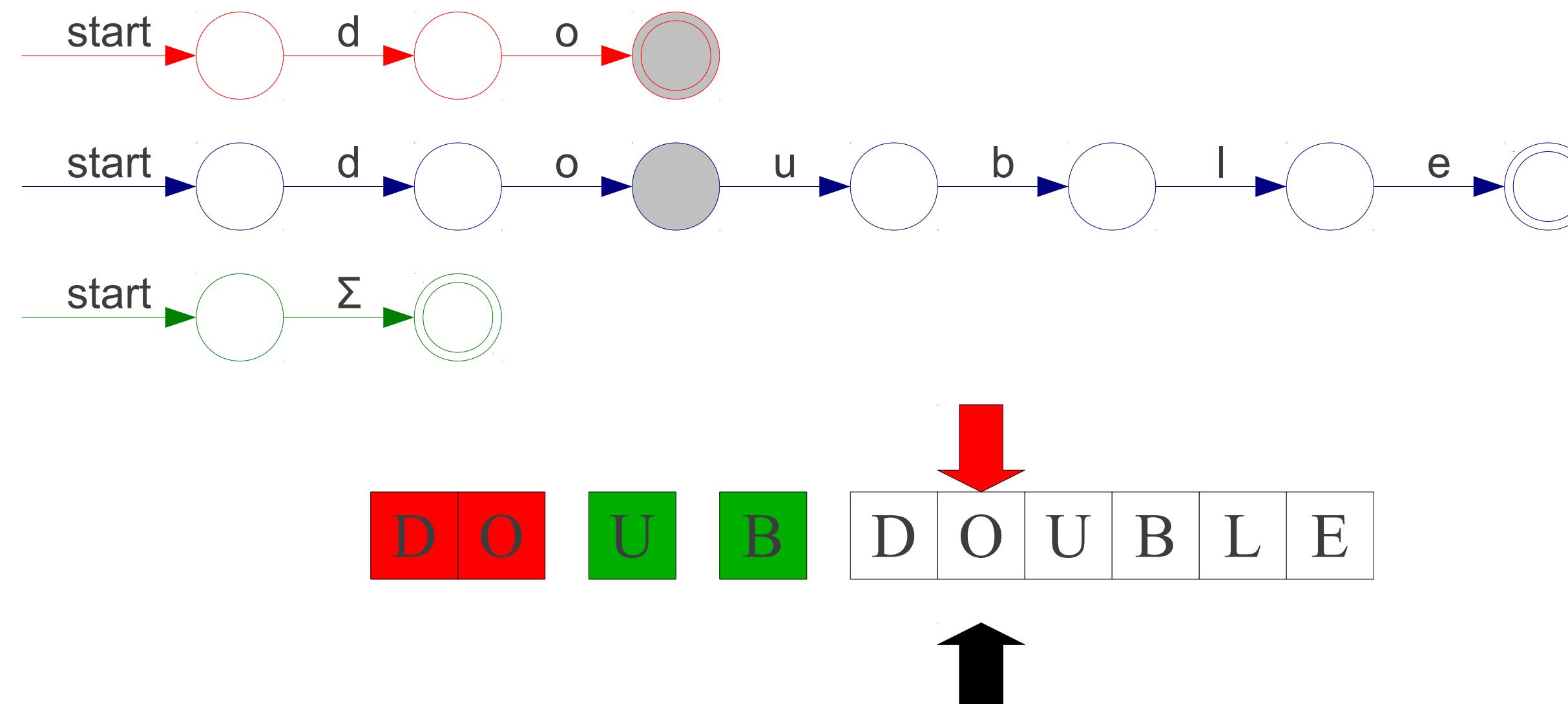
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

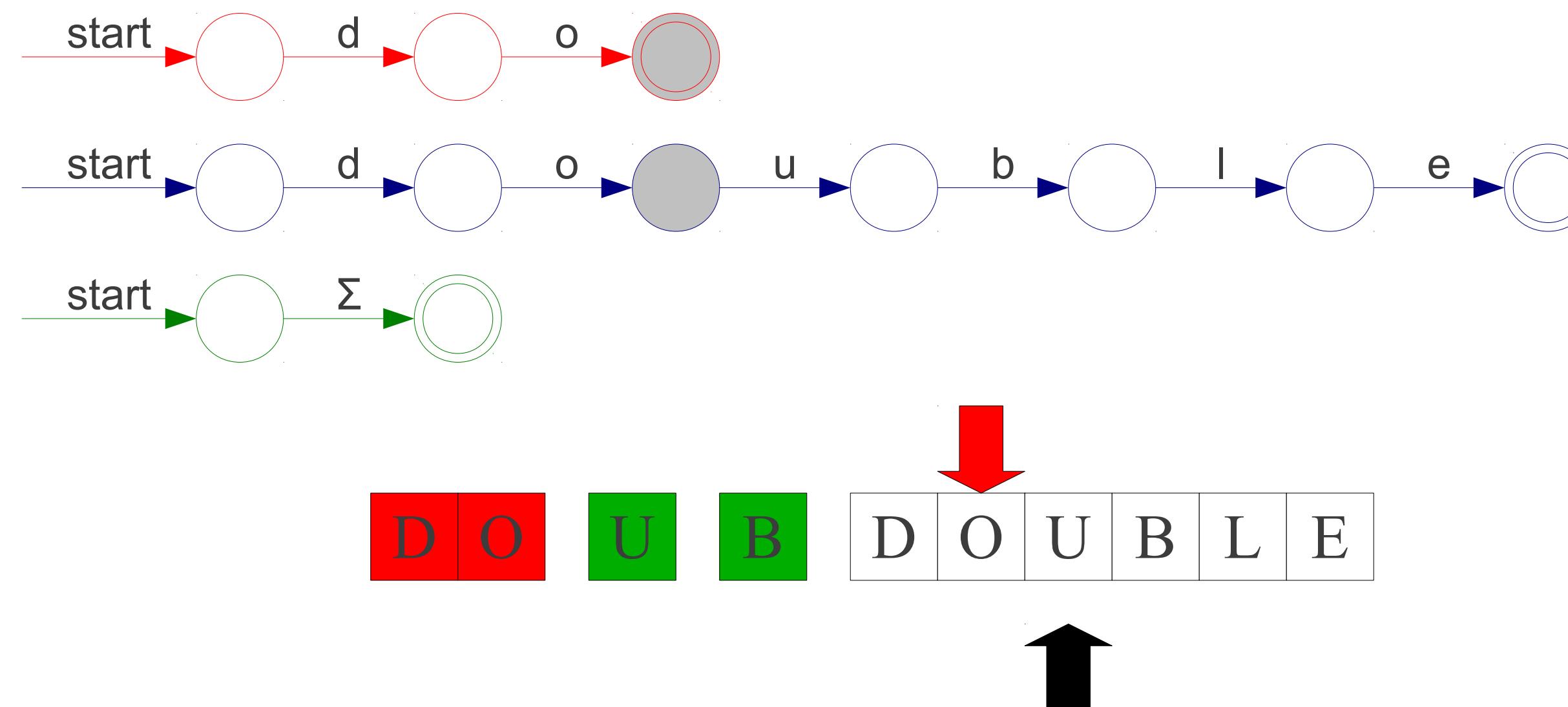
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

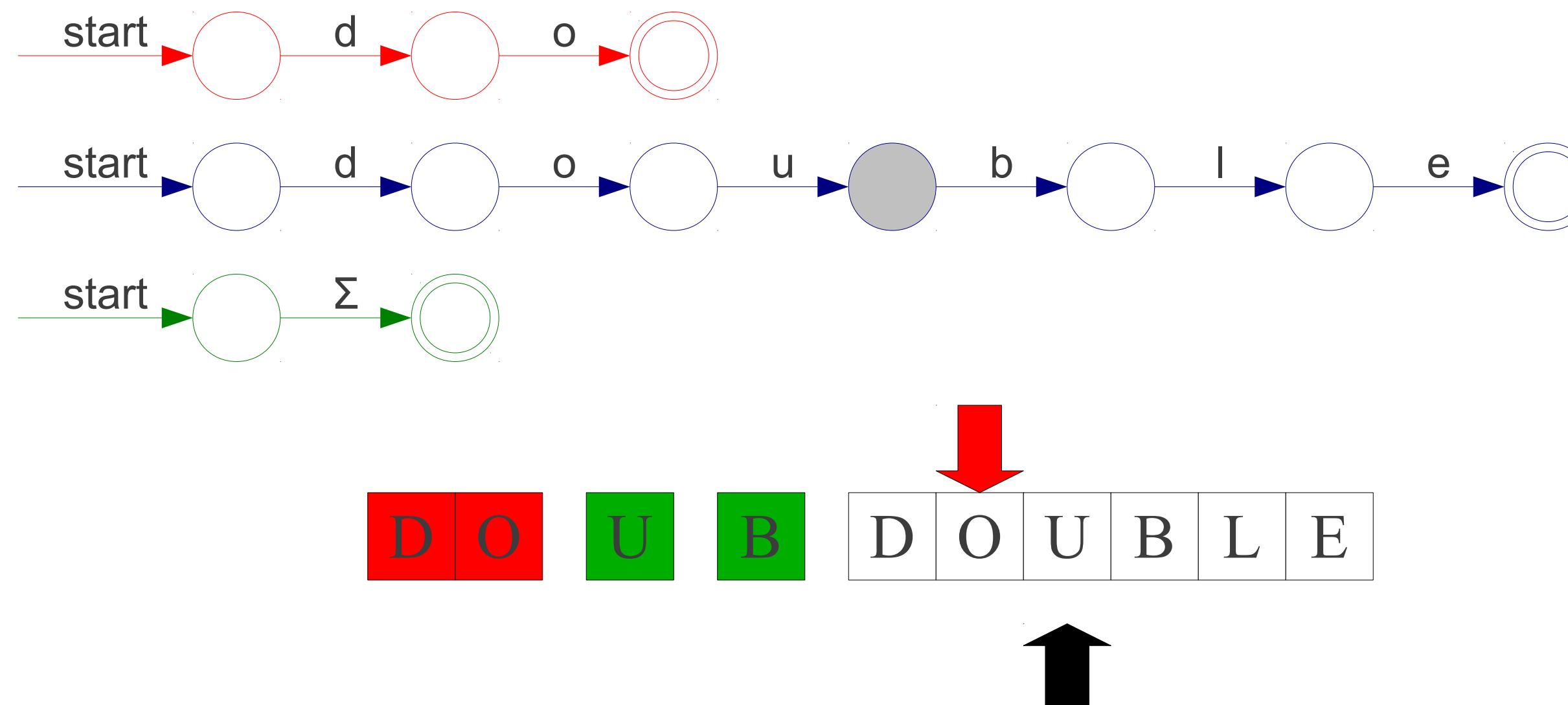
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

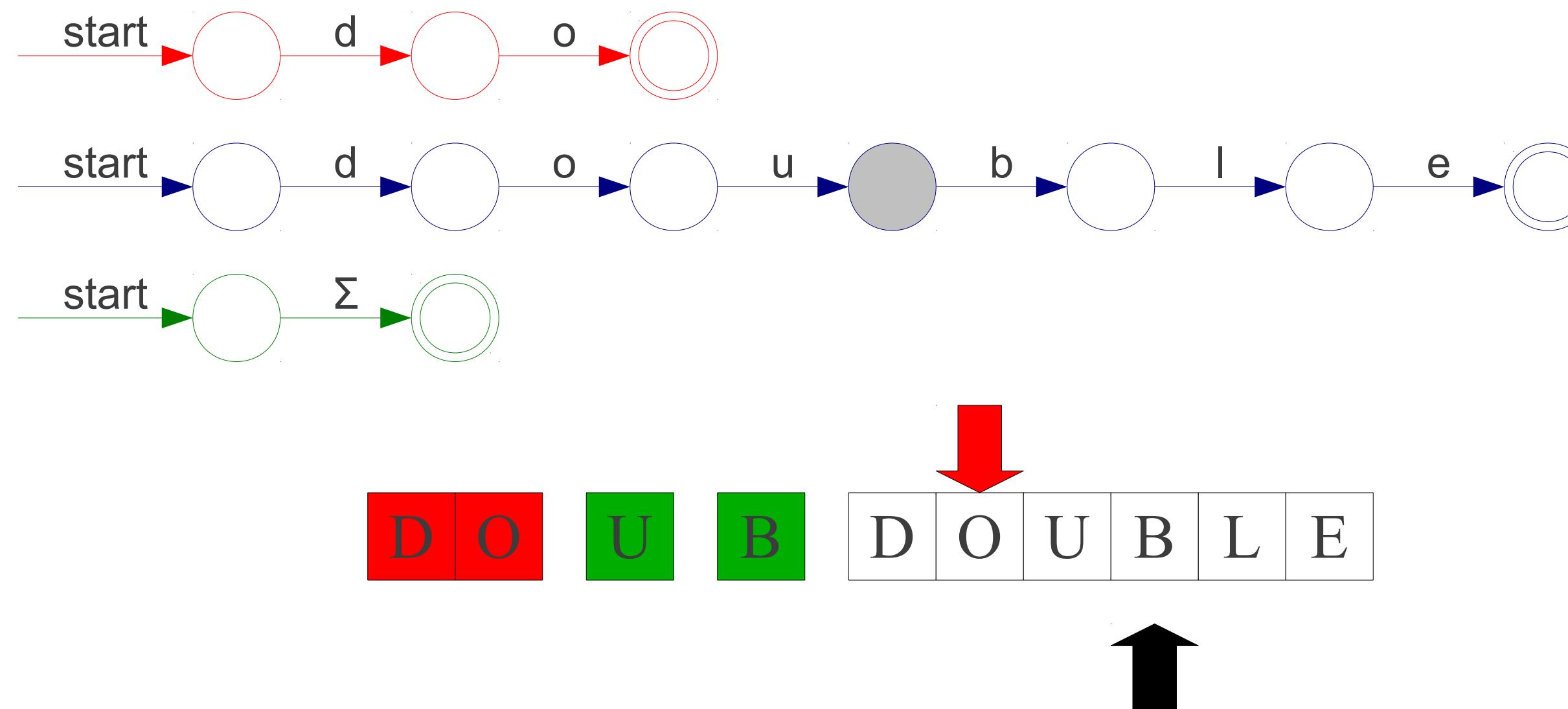
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

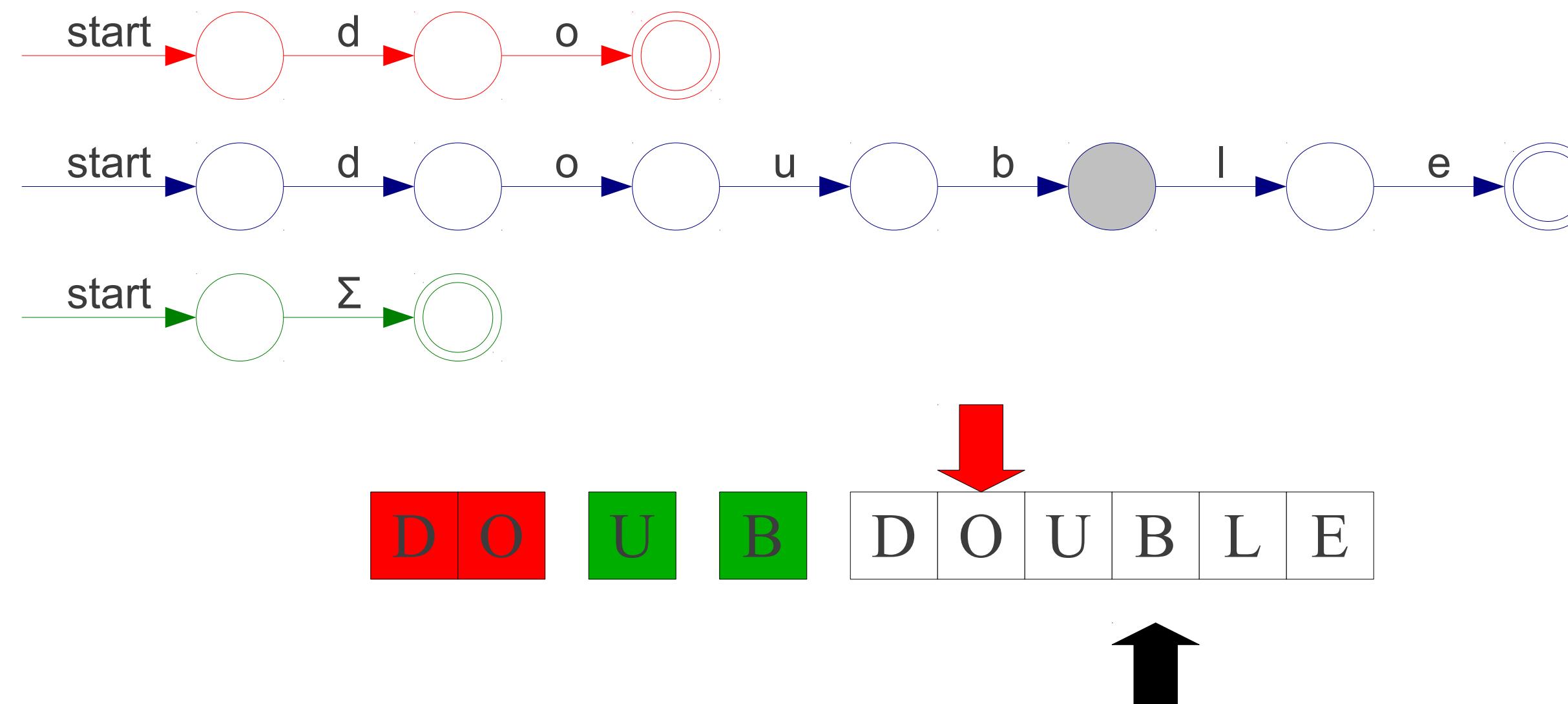
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

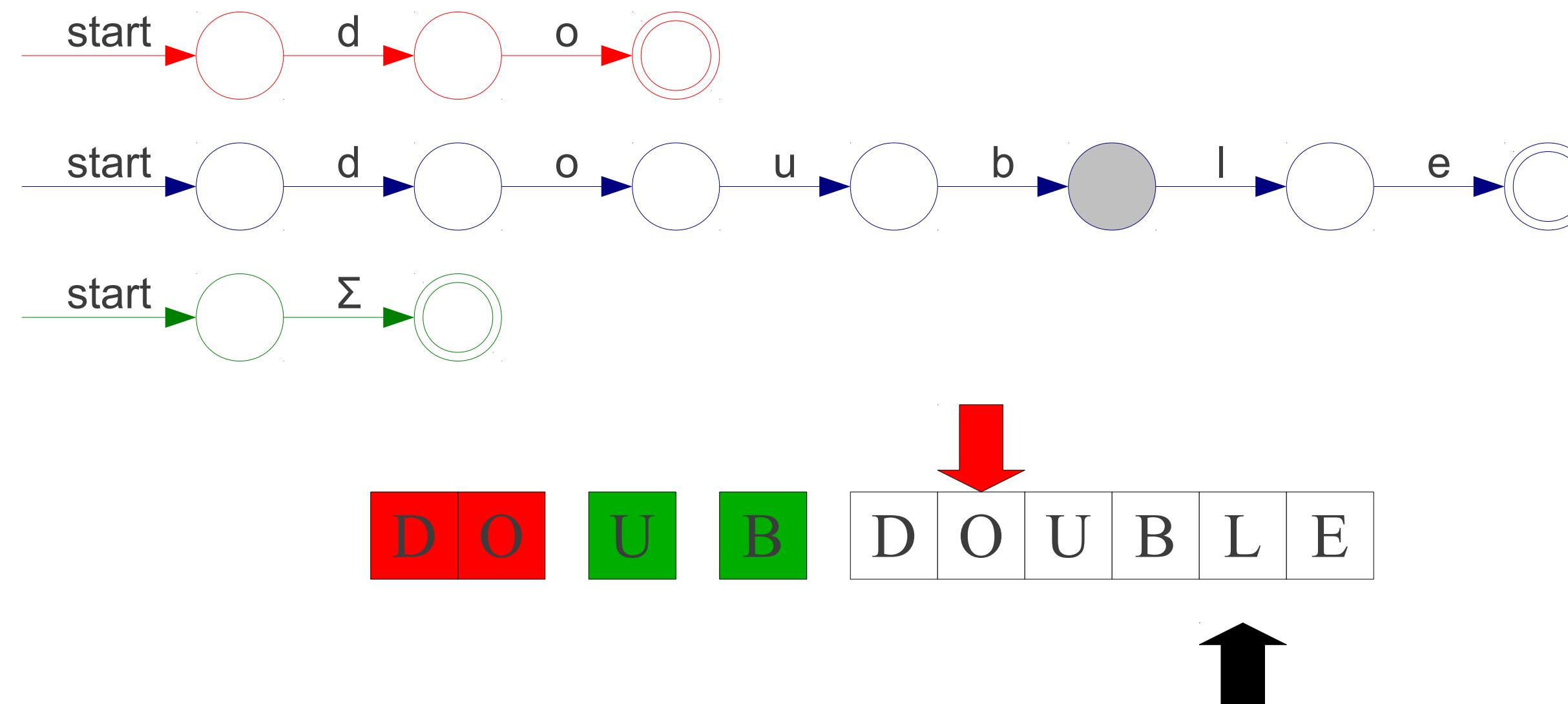
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

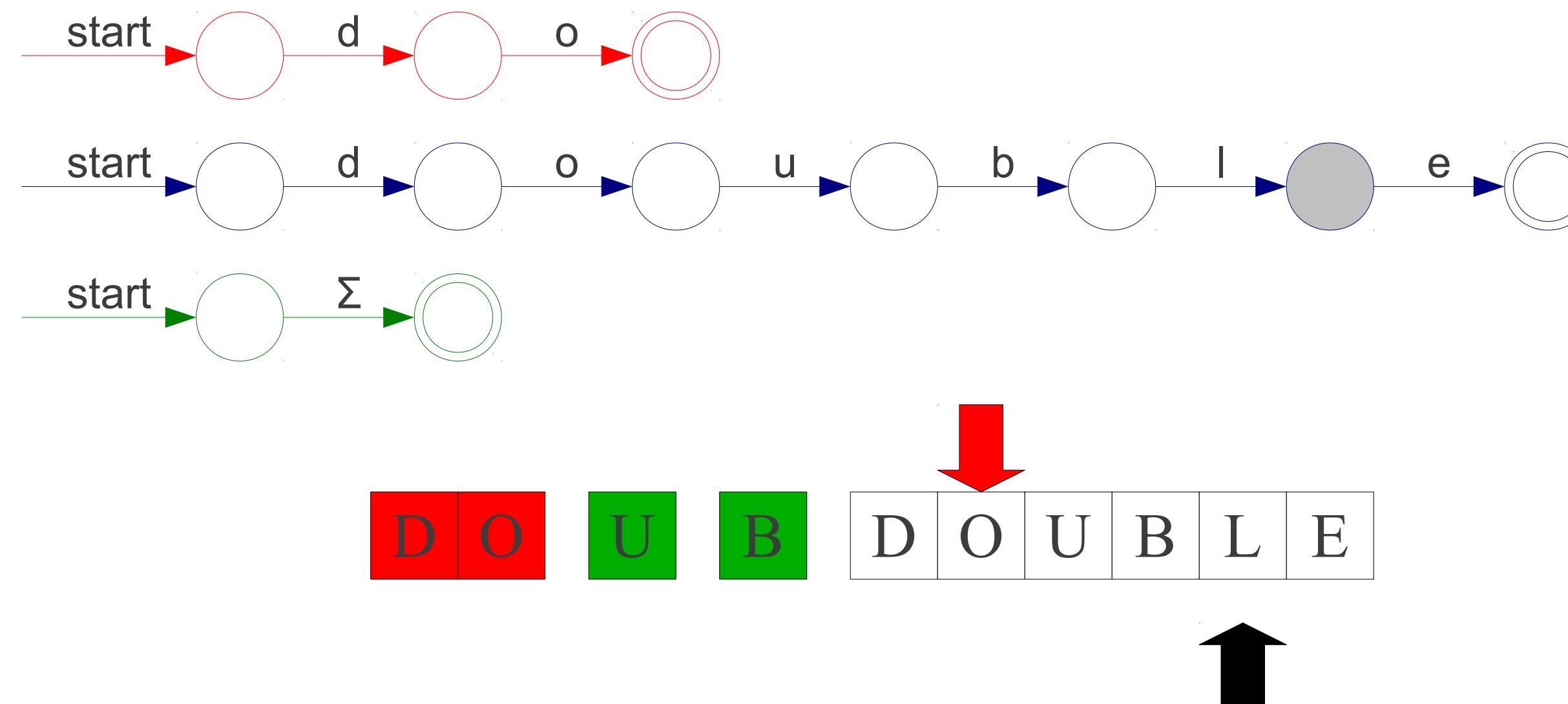
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

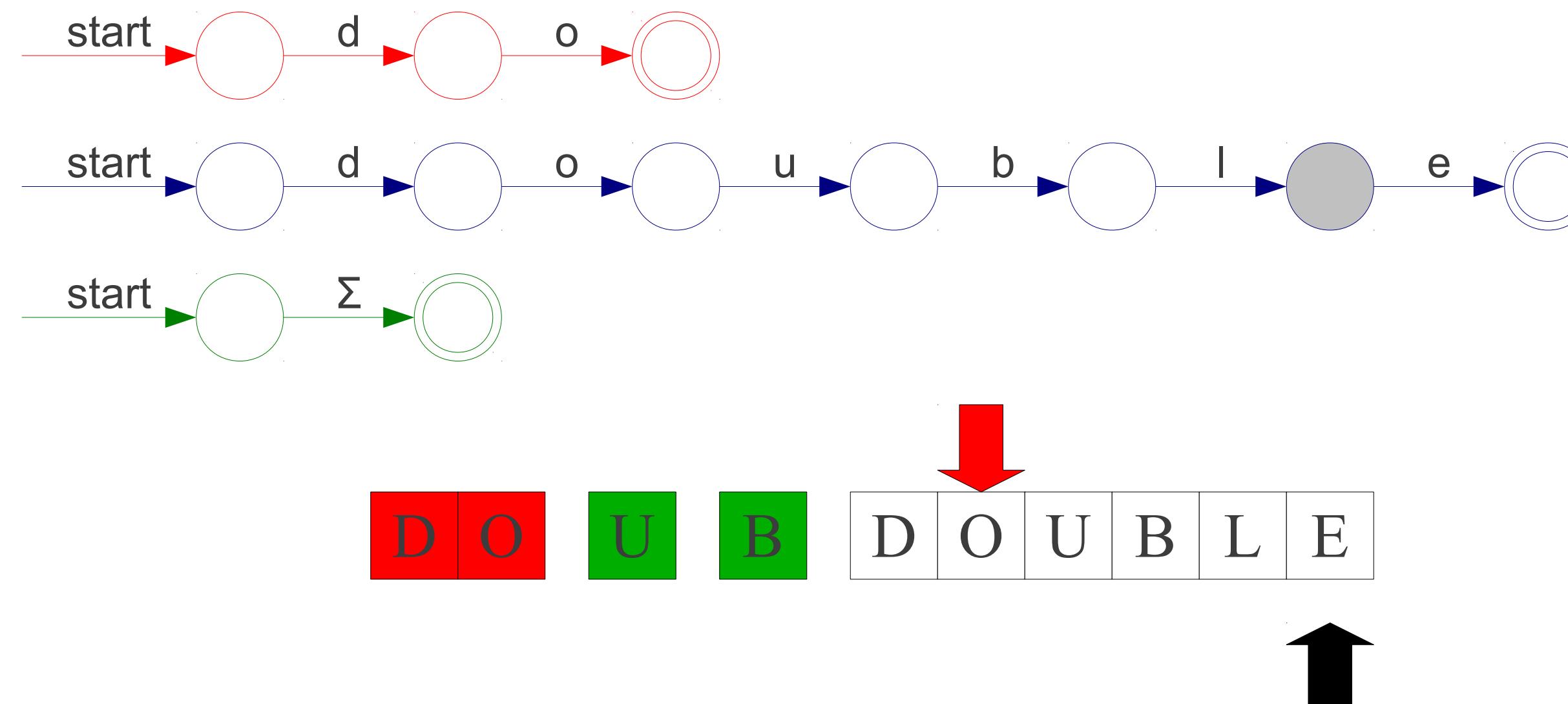
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

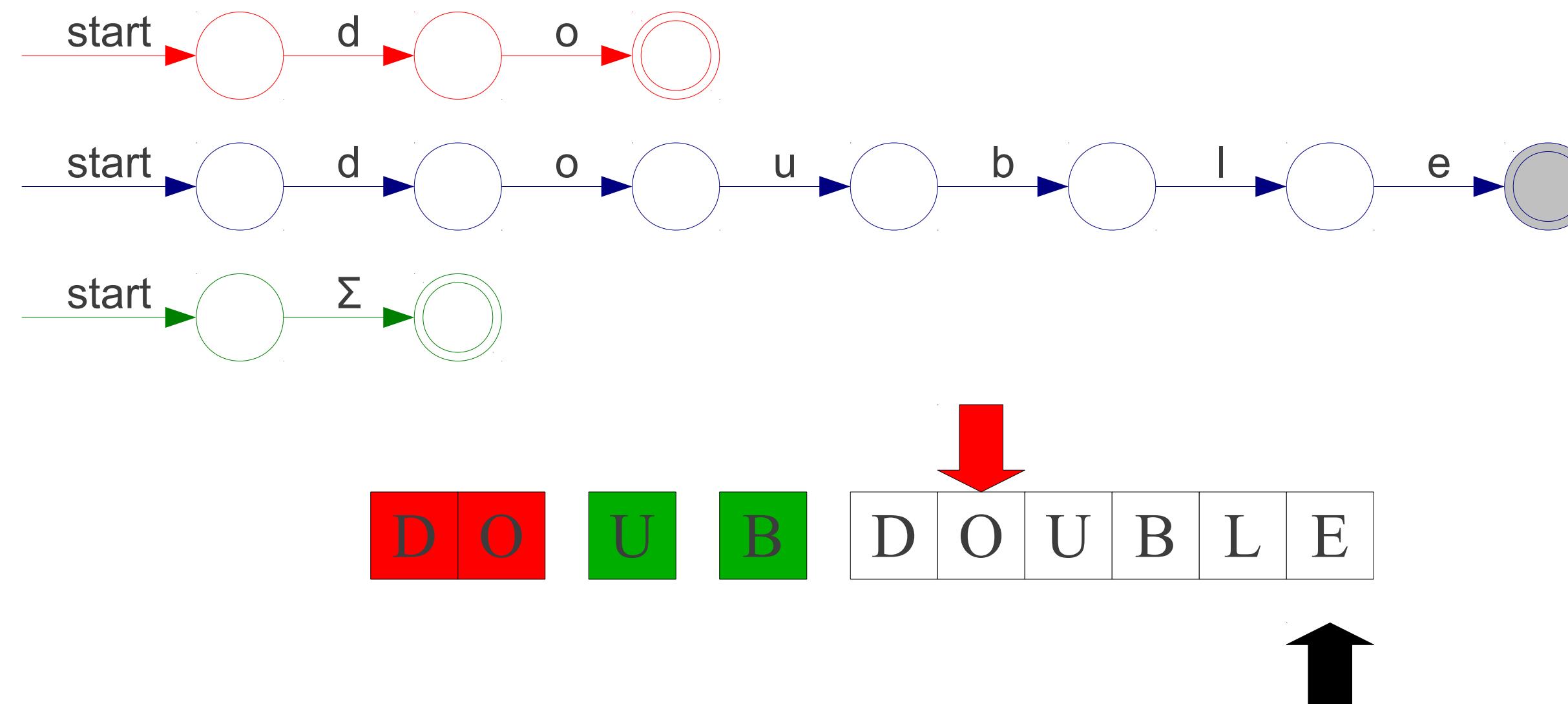
T_Double

T_Mystery

do

double

[A-Za-z]



Implementing Maximal Munch

T_Do

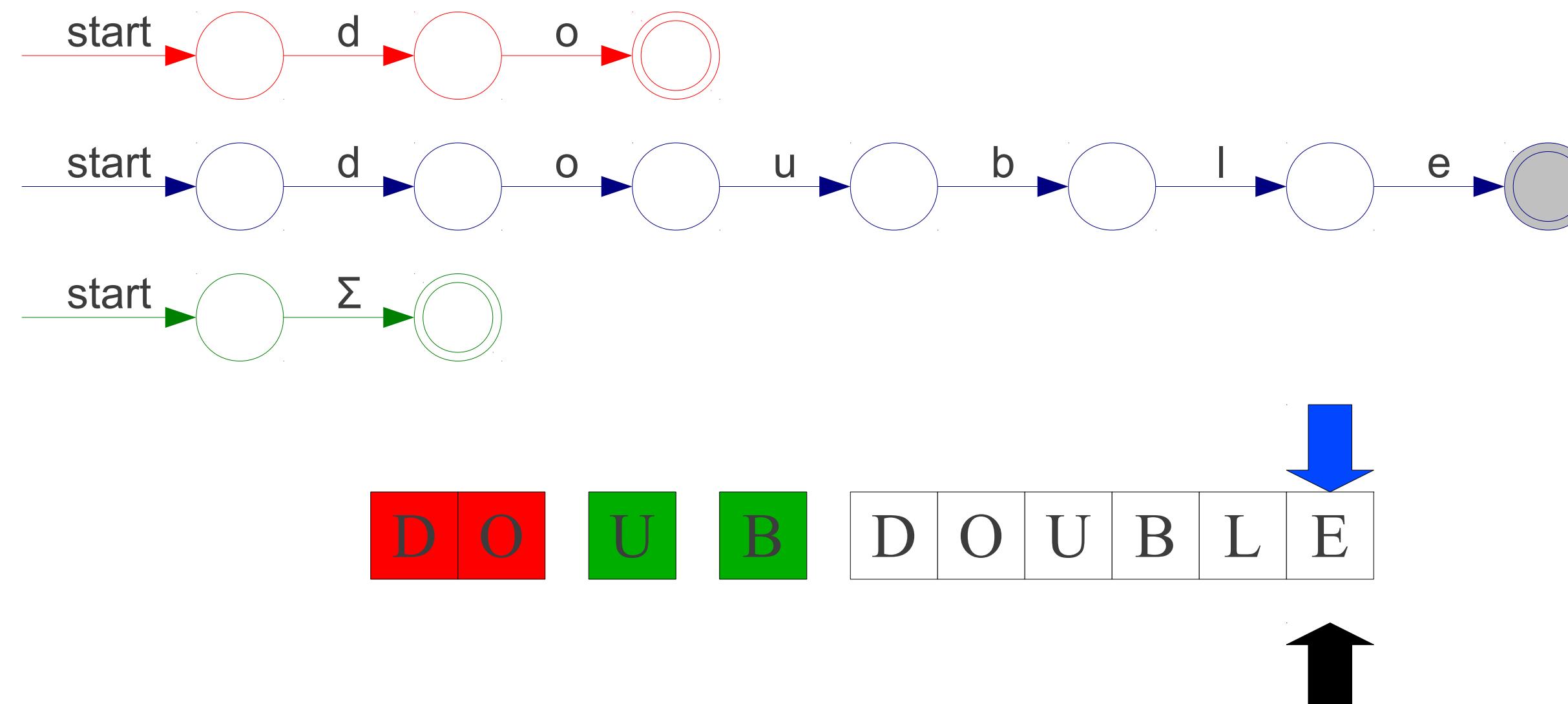
T_Double

T_Mystery

do

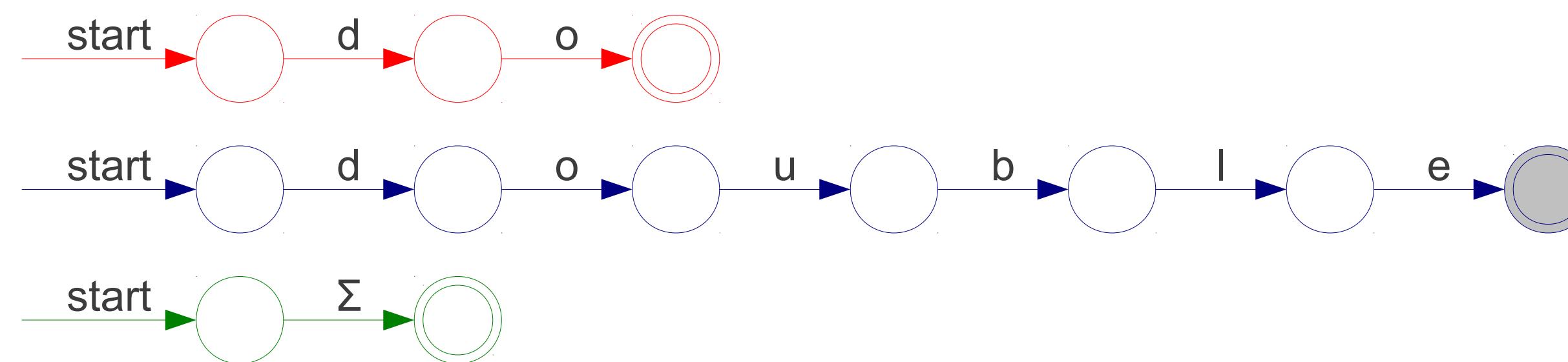
double

[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 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., "((00(((0))))" but not "(0)(0"

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 (((((...n times where n is > the number of states of the automaton. Since n is > the number of states, there must be some smaller prefix (((...m where m < n and the automaton reaches the same state.

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

CONTEXT FREE GRAMMARS

Context-free Grammars

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

$$\begin{aligned} S &\mapsto (S)S \\ S &\mapsto \epsilon \end{aligned}$$

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. 254

CFGs Mathematically

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

$$S \rightarrow (S)S$$

$$S \rightarrow \epsilon$$

- How many terminals? How many nonterminals? Productions?

Another Example: Sum Grammar

- A grammar that accepts parenthesized sums of numbers:

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

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

- Note the vertical bar ‘|’ is shorthand for multiple productions:

$$\left. \begin{array}{l} S \mapsto E + S \\ S \mapsto E \\ E \mapsto \text{number} \\ E \mapsto (S) \end{array} \right\} \begin{array}{l} 4 \text{ productions} \\ 2 \text{ nonterminals: } S, E \\ 4 \text{ terminals: (,), +, number} \\ \text{Start symbol: } S \end{array}$$

Derivations in CFGs

- Example: derive $(1 + 2 + (3 + 4)) + 5$
- $\underline{S} \rightarrow \underline{E} + S$
 $\rightarrow (\underline{S}) + S$
 $\rightarrow (\underline{E} + S) + S$
 $\rightarrow (1 + \underline{S}) + S$
 $\rightarrow (1 + \underline{E} + S) + S$
 $\rightarrow (1 + 2 + \underline{S}) + S$
 $\rightarrow (1 + 2 + \underline{E}) + S$
 $\rightarrow (1 + 2 + (\underline{S})) + S$
 $\rightarrow (1 + 2 + (\underline{E} + S)) + S$
 $\rightarrow (1 + 2 + (3 + \underline{S})) + S$
 $\rightarrow (1 + 2 + (3 + \underline{E})) + S$
 $\rightarrow (1 + 2 + (3 + 4)) + \underline{S}$
 $\rightarrow (1 + 2 + (3 + 4)) + \underline{E}$
 $\rightarrow (1 + 2 + (3 + 4)) + 5$

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

For arbitrary strings α, β, γ and production rule $A \rightarrow \beta$
a single step of the derivation is:

$$\alpha A \gamma \rightarrow \alpha \beta \gamma$$

(*substitute β for an occurrence of A*)

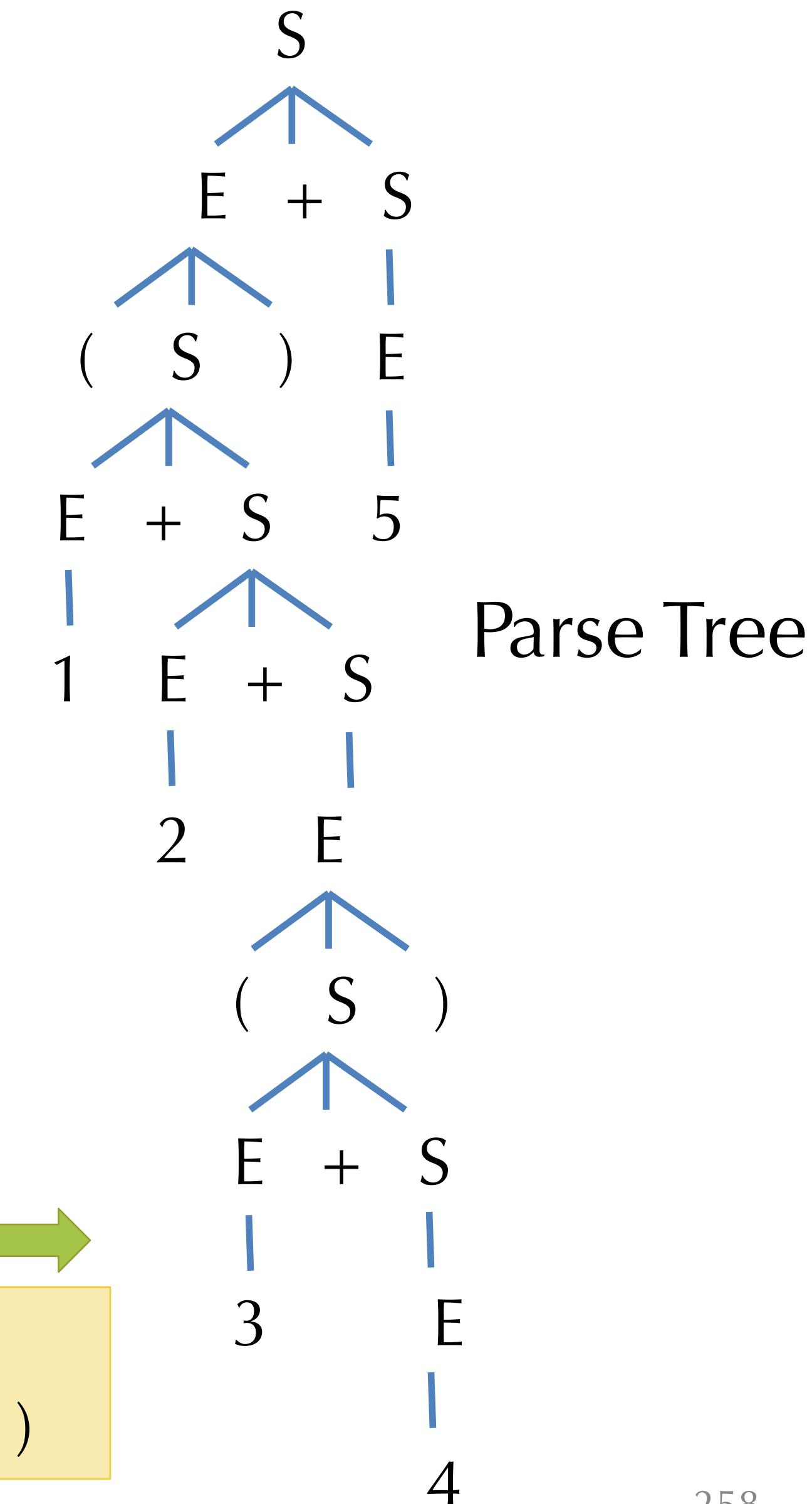
In general, there are many possible derivations for a given string

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
- $(1 + 2 + (3 + 4)) + 5$

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



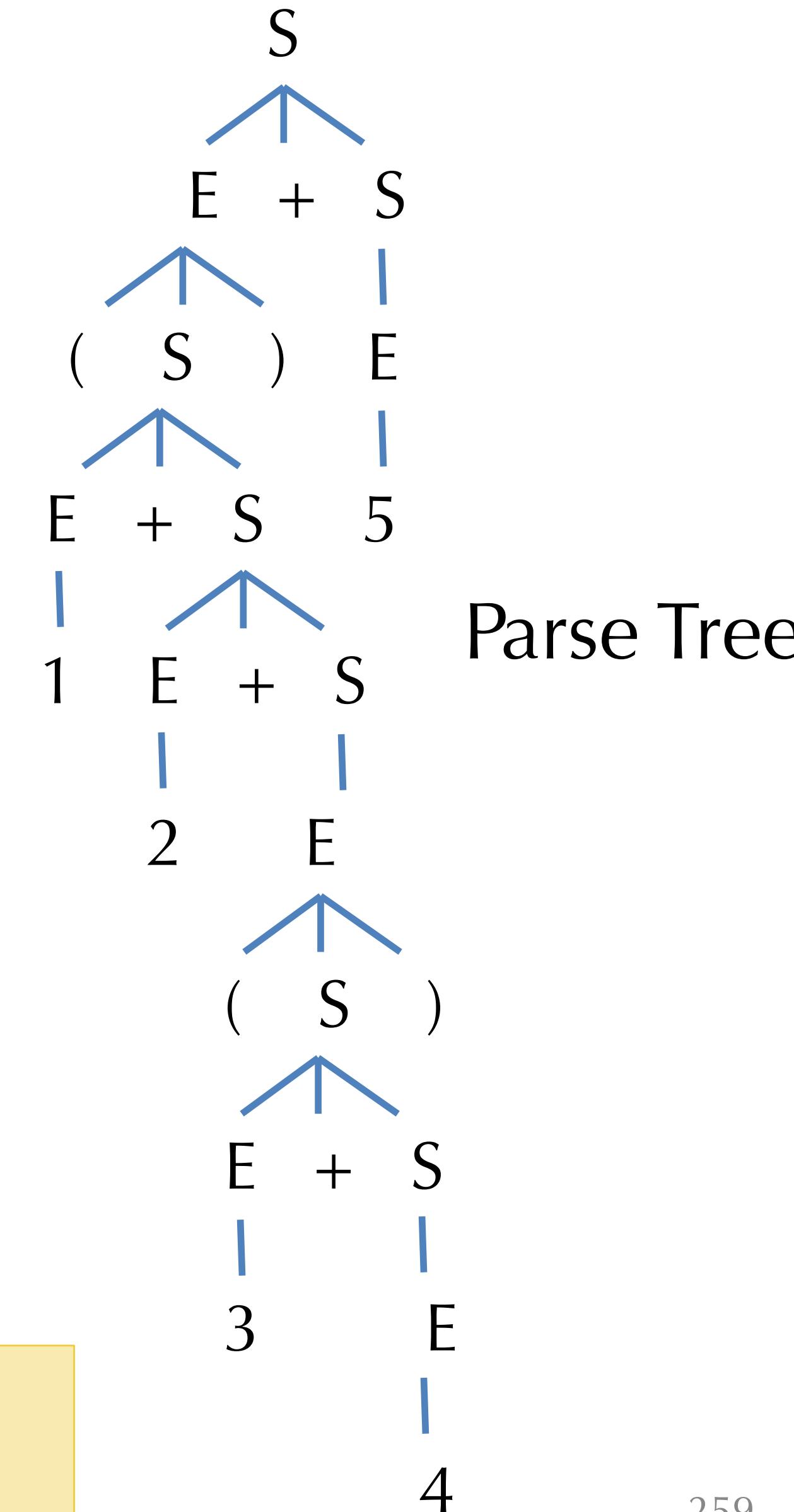
From Derivations to Parse Trees

- Idea: Think of the non-terminals of the CFG as mutually-recursive **enum** types in Rust

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

- Then the parse trees are the values of the enum type

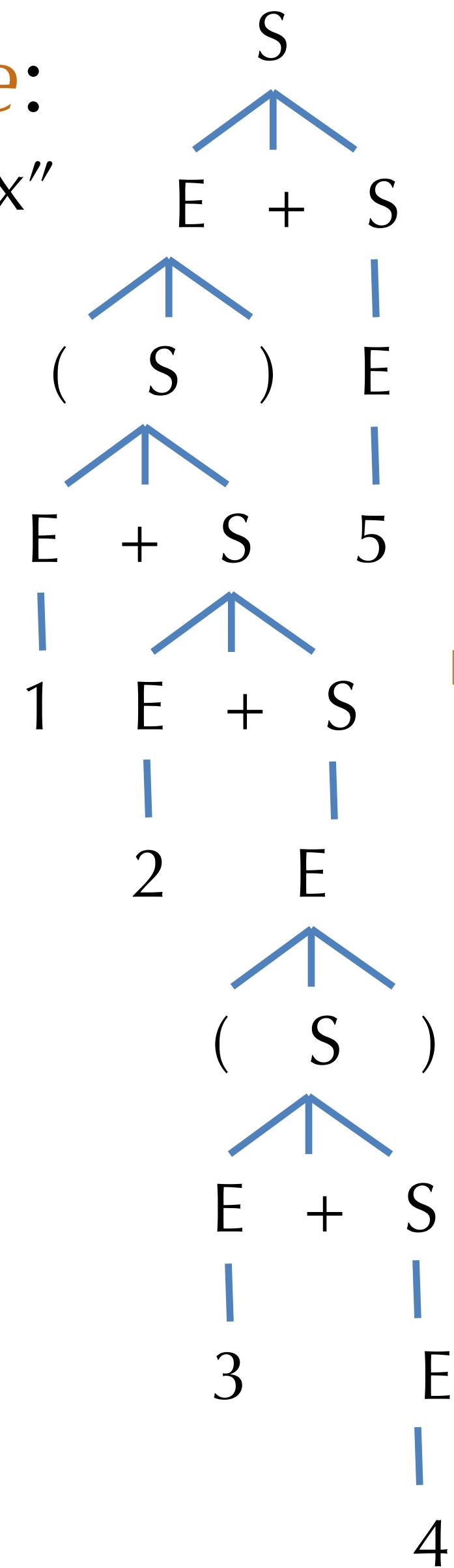
```
S  $\rightarrow$  E + S | E  
E  $\rightarrow$  number | ( S )
```



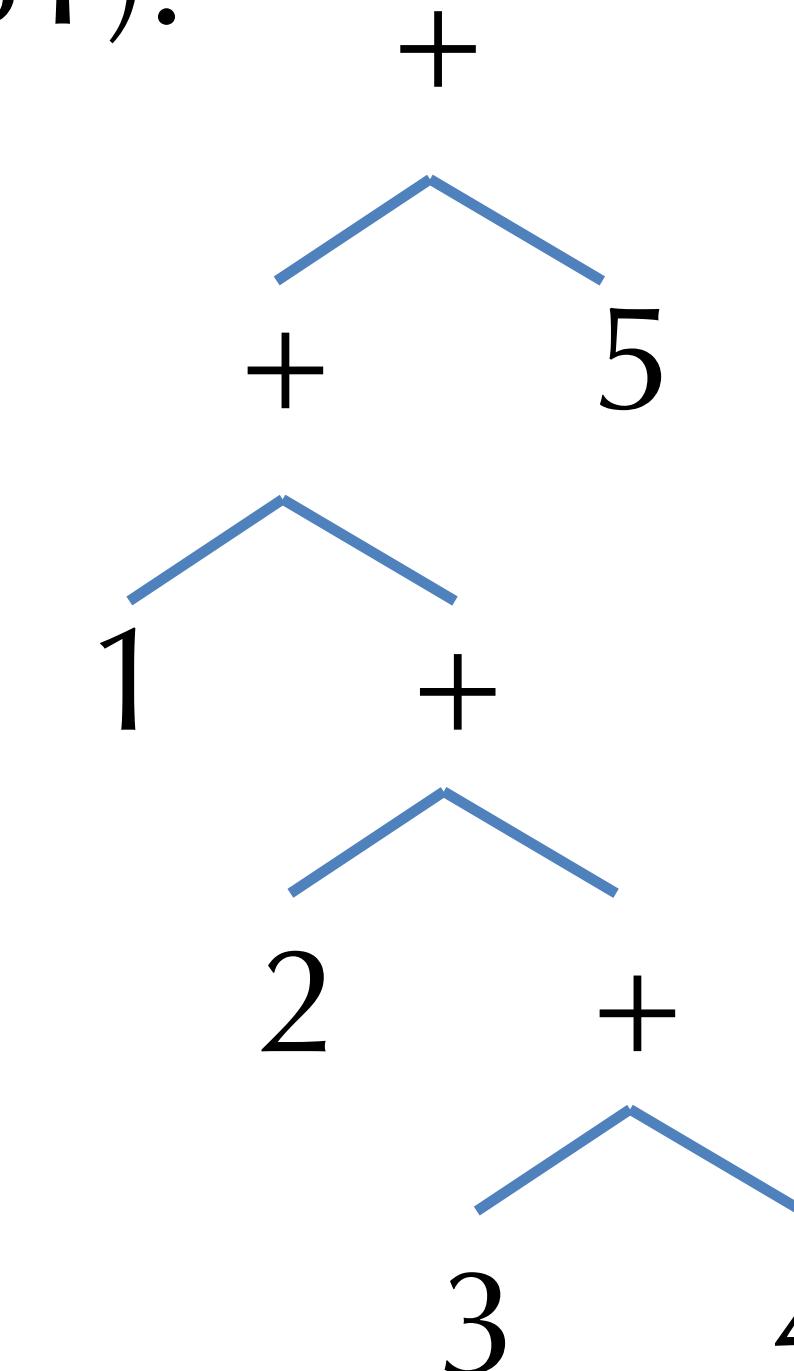
From Parse Trees to Abstract Syntax

- *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*, unneeded information.

```
S → E + S | E  
E → number | ( 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:

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

- Rightmost derivation:

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

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

Loops and Termination

- Some care is needed when defining CFGs
- Consider:

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

- 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:
$$S \rightarrow (S)$$
- 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
- ϵ $S \rightarrow \epsilon$
- $R_1 | R_2$ $S \rightarrow S_1$
 $S \rightarrow S_2$
 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, S_2 are CFGs for R_1, R_2
- R^* $S \rightarrow \epsilon$
 $S \rightarrow S_R S$
 where S_R are CFGs for R



Associativity, ambiguity, and precedence.

GRAMMARS FOR PROGRAMMING LANGUAGES

Associativity

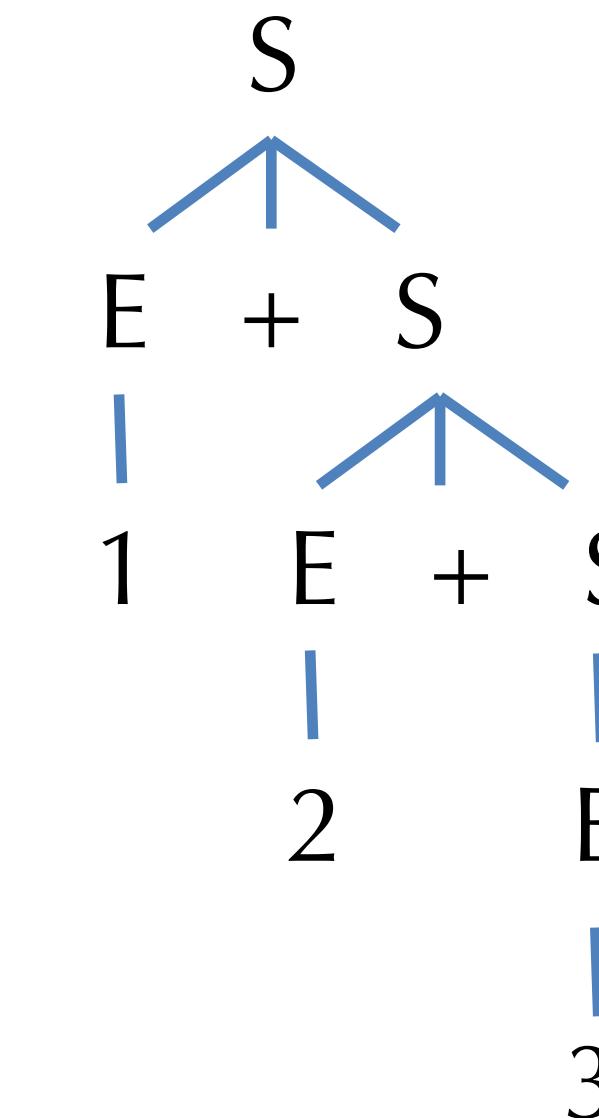
Consider the input: $1 + 2 + 3$

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

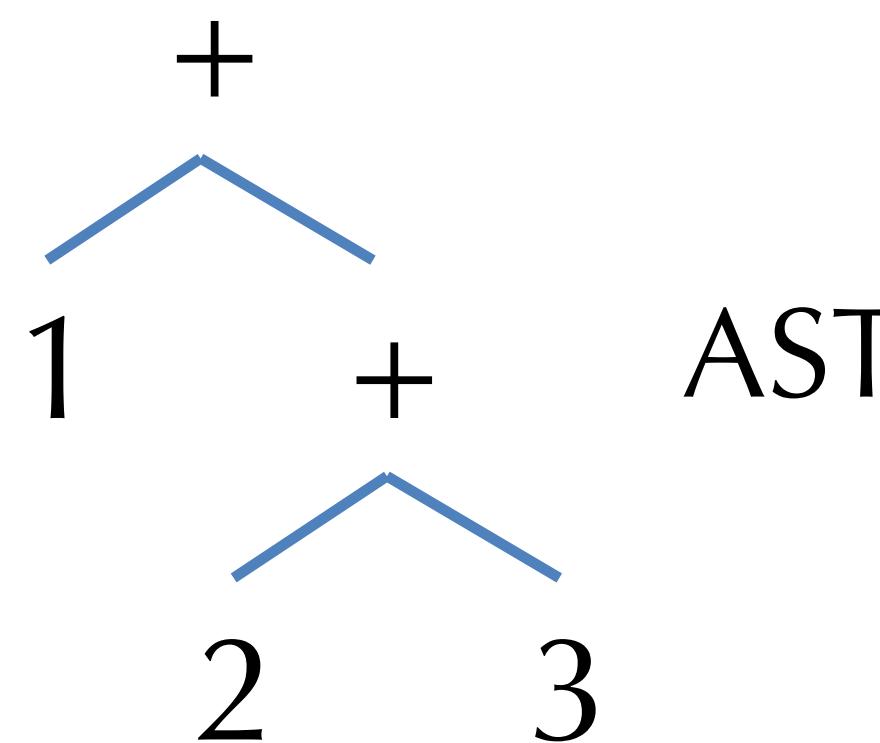
Leftmost derivation: Rightmost derivation:

$$\begin{array}{l} S \mapsto E + S \\ \quad\quad\quad \vdash 1 + S \\ \quad\quad\quad \vdash 1 + E + S \\ \quad\quad\quad \vdash 1 + 2 + S \\ \quad\quad\quad \vdash 1 + 2 + E \\ \quad\quad\quad \vdash 1 + 2 + 3 \end{array}$$

$$\begin{array}{l} S \mapsto E + S \\ \quad\quad\quad \vdash E + E + S \\ \quad\quad\quad \vdash E + E + E \\ \quad\quad\quad \vdash E + E + 3 \\ \quad\quad\quad \vdash E + 2 + 3 \\ \quad\quad\quad \vdash 1 + 2 + 3 \end{array}$$



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*...

$$\begin{array}{l} S \rightarrow E + S \quad | \quad E \\ E \rightarrow \text{number} \quad | \quad (S) \end{array}$$

S refers to itself
on the right of +

- How would you make ‘+’ left associative?
- What are the trees for “ $1 + 2 + 3$ ”?

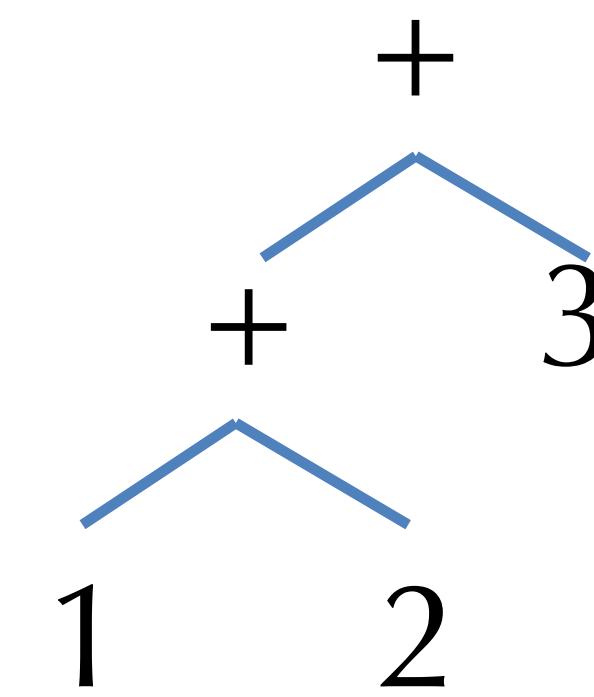
Ambiguity

- Consider this grammar:

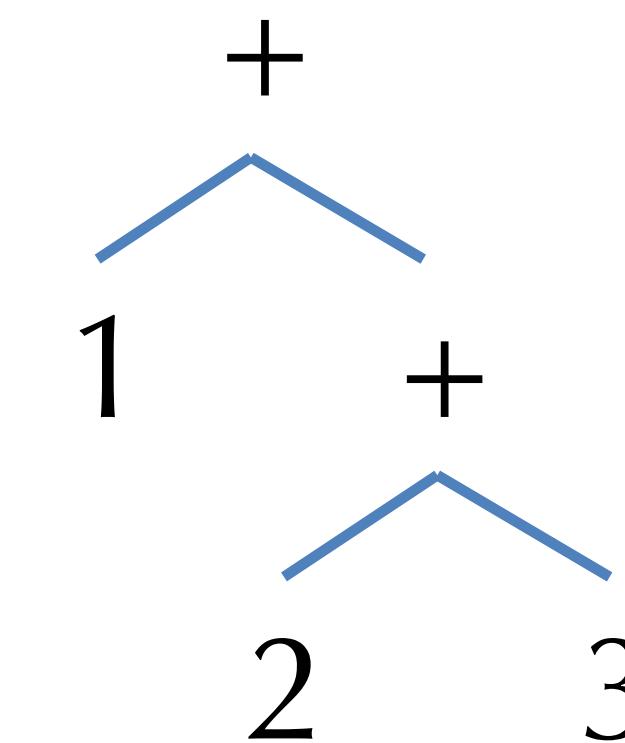
$$S \rightarrow S + S \mid (S) \mid \text{number}$$

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

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



AST 1



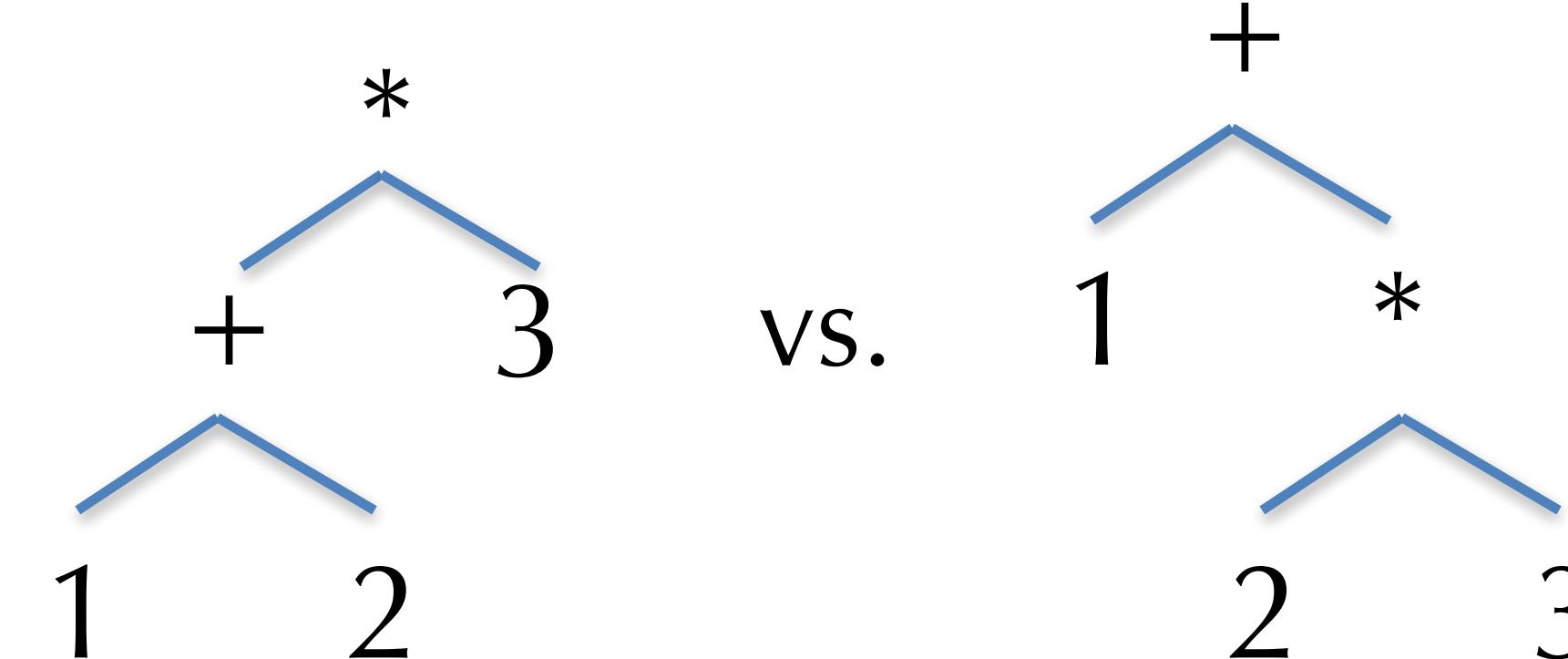
AST 2

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?
 - Some operations are only left (or right) associative. Examples?
- Moreover, if there are multiple operations, ambiguity in the grammar leads to ambiguity in their *precedence*
- Consider:

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

- 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:

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

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

$$S_0 \rightarrow S_0 + S_1 \mid S_1$$

$$S_1 \rightarrow S_2 * S_1 \mid S_2$$

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

LL(1) GRAMMARS

Consider finding left-most derivations

- Look at only one input symbol at a time.

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

Partly-derived String	Look-ahead	Parsed/Unparsed Input
<u>S</u>	($(1 + 2 + (3 + 4)) + 5$
$\rightarrow \underline{E} + S$	($(1 + 2 + (3 + 4)) + 5$
$\rightarrow (\underline{S}) + S$	1	$(1 + 2 + (3 + 4)) + 5$
$\rightarrow (\underline{E} + S) + S$	1	$(1 + 2 + (3 + 4)) + 5$
$\rightarrow (1 + \underline{S}) + S$	2	$(1 + 2 + (3 + 4)) + 5$
$\rightarrow (1 + \underline{E} + S) + S$	2	$(1 + 2 + (3 + 4)) + 5$
$\rightarrow (1 + 2 + \underline{S}) + S$	($(1 + 2 + (3 + 4)) + 5$
$\rightarrow (1 + 2 + \underline{E}) + S$	($(1 + 2 + (3 + 4)) + 5$
$\rightarrow (1 + 2 + (\underline{S})) + S$	3	$(1 + 2 + (3 + 4)) + 5$
$\rightarrow (1 + 2 + (\underline{E} + S)) + S$	3	$(1 + 2 + (3 + 4)) + 5$
$\rightarrow \dots$		

There is a problem

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

$$(1) \quad S \rightarrow E \rightarrow (S) \rightarrow (E) \rightarrow (1)$$

vs.

$$(1) + 2 \quad S \rightarrow E + S \rightarrow (S) + S \rightarrow (E) + S \rightarrow (1) + S \rightarrow (1) + E \rightarrow (1) + 2$$

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

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

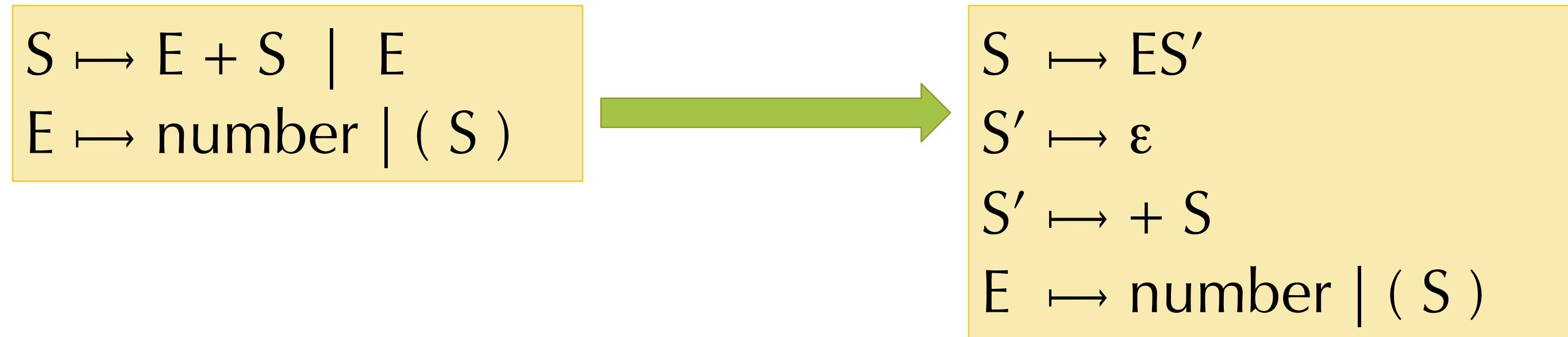
Grammar is the problem

- 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:



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

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

LL(1) Parse of the input string

- Look at only one input symbol at a time.

$$\begin{aligned}
 S &\rightarrow ES' \\
 S' &\rightarrow \epsilon \\
 S' &\rightarrow + S \\
 E &\rightarrow \text{number} \mid (S)
 \end{aligned}$$

Partly-derived String	Look-ahead	Parsed/Unparsed Input
<u>S</u>	($(1 + 2 + (3 + 4)) + 5$
$\rightarrow \underline{E} S'$	($(1 + 2 + (3 + 4)) + 5$
$\rightarrow (\underline{S}) S'$	1	$(1 + 2 + (3 + 4)) + 5$
$\rightarrow (\underline{E} S') S'$	1	$(1 + 2 + (3 + 4)) + 5$
$\rightarrow (1 \underline{S'}) S'$	+	$(1 + 2 + (3 + 4)) + 5$
$\rightarrow (1 + \underline{S}) S'$	2	$(1 + 2 + (3 + 4)) + 5$
$\rightarrow (1 + \underline{E} S') S'$	2	$(1 + 2 + (3 + 4)) + 5$
$\rightarrow (1 + 2 \underline{S'}) S'$	+	$(1 + 2 + (3 + 4)) + 5$
$\rightarrow (1 + 2 + \underline{S}) S'$	($(1 + 2 + (3 + 4)) + 5$
$\rightarrow (1 + 2 + \underline{E} S') S'$	($(1 + 2 + (3 + 4)) + 5$
$\rightarrow (1 + 2 + (\underline{S}) S') S'$	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 → production

$$\begin{aligned}
 T &\rightarrow S\$ \\
 S &\rightarrow E S' \\
 S' &\rightarrow \epsilon \\
 S' &\rightarrow + S \\
 E &\rightarrow \text{number} \mid (S)
 \end{aligned}$$

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

- 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 γ
 - Add the production $\rightarrow \gamma$ to the entry (A, token) for each such token.
- If γ can derive ϵ (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, token) for each such token.
- Note: The grammar is LL(1) *if and only if* all entries have at most one production

Example

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

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

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

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

Converting the table to code

- Define n mutually recursive functions
 - one for each nonterminal A: `parse_A`
 - The type of `parse_A` is $() \rightarrow \text{ast}$ 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	$\rightarrow S\$$		$\rightarrow S\$$		
S	$\rightarrow E S'$		$\rightarrow E S'$		
S'		$\rightarrow + S$		$\rightarrow \epsilon$	$\rightarrow \epsilon$
E	$\rightarrow \text{num.}$		$\rightarrow (S)$		

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

Bottom-up Parsing (LR Parsers)

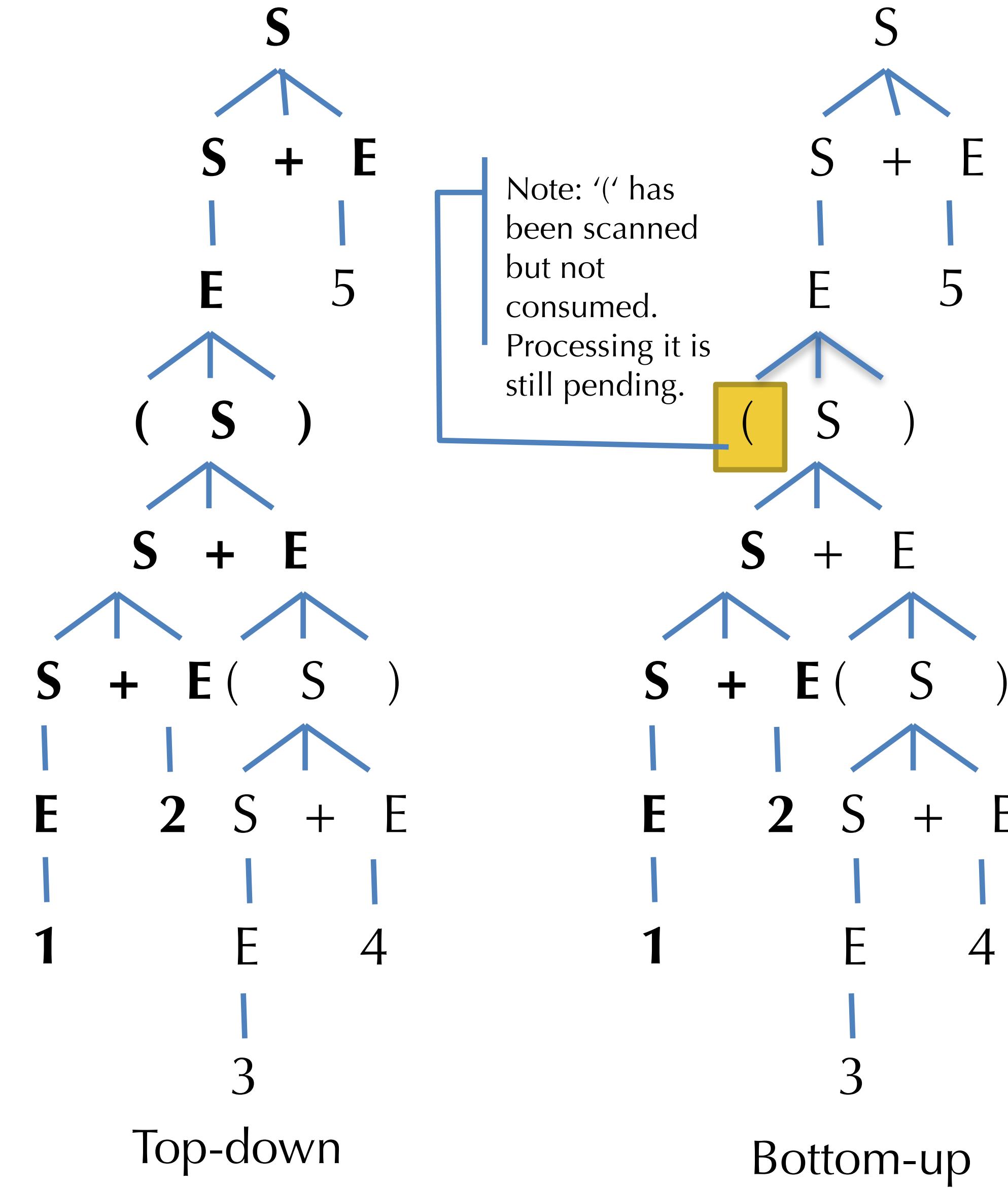
- LR(k) parser:
 - Left-to-right scanning
 - Rightmost derivation
 - k lookahead symbols
- LR grammars are more expressive than LL
 - Can handle left-recursive (and right recursive) grammars; virtually all programming languages
 - Easier to express programming language syntax (no left factoring)
- Technique: “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, ocaml yacc, lalrpop, etc.)
 - Better error detection/recovery

Top-down vs. Bottom up

- Consider the left-recursive grammar:

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

- $(1 + 2 + (3 + 4)) + 5$
- 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...



Progress of Bottom-up Parsing

Reductions	Scanned	Input Remaining
$(1 + 2 + (3 + 4)) + 5 \leftarrow$		$(1 + 2 + (3 + 4)) + 5$
$(\underline{E} + 2 + (3 + 4)) + 5 \leftarrow$	($1 + 2 + (3 + 4)) + 5$
$(\underline{S} + 2 + (3 + 4)) + 5 \leftarrow$	(1	$+ 2 + (3 + 4)) + 5$
$(S + \underline{E} + (3 + 4)) + 5 \leftarrow$	(1 + 2	$+ (3 + 4)) + 5$
$(\underline{S} + (3 + 4)) + 5 \leftarrow$	(1 + 2	$+ (3 + 4)) + 5$
$(S + (\underline{E} + 4)) + 5 \leftarrow$	(1 + 2 + (3	$+ 4)) + 5$
$(S + (\underline{S} + 4)) + 5 \leftarrow$	(1 + 2 + (3	$+ 4)) + 5$
$(S + (S + \underline{E})) + 5 \leftarrow$	(1 + 2 + (3 + 4) + 5
$(S + (\underline{S})) + 5 \leftarrow$	(1 + 2 + (3 + 4) + 5
$(S + \underline{E}) + 5 \leftarrow$	(1 + 2 + (3 + 4)) + 5
$(\underline{S}) + 5 \leftarrow$	(1 + 2 + (3 + 4)) + 5
$\underline{E} + 5 \leftarrow$	(1 + 2 + (3 + 4))	+ 5
$\underline{S} + 5 \leftarrow$	(1 + 2 + (3 + 4))	+ 5
$S + \underline{E} \leftarrow$	(1 + 2 + (3 + 4)) + 5	
S		

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

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 γ at top of stack with nonterminal X such that $X \rightarrow \gamma$ is a production. (pop γ , push X)

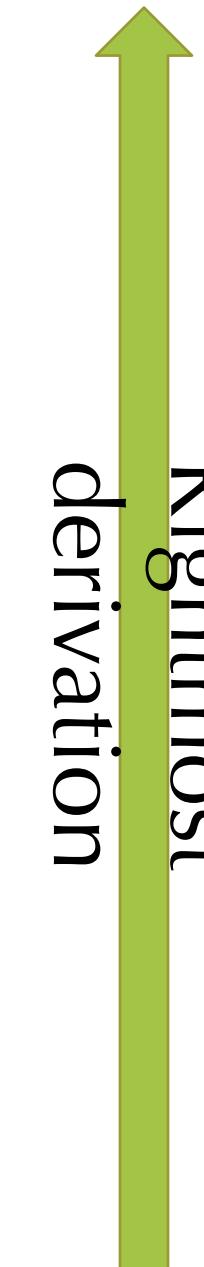
$$\begin{array}{l} S \rightarrow S + E \mid E \\ E \rightarrow \text{number} \mid (S) \end{array}$$

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

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	(E + 2 + (3 + 4)) + 5
(S	+ 2 + (3 + 4)) + 5	(S + 2 + (3 + 4)) + 5
(S +	2 + (3 + 4)) + 5	
(S + 2	+ (3 + 4)) + 5	
(S + E	+ (3 + 4)) + 5	(S + E + (3 + 4)) + 5
(S	+ (3 + 4)) + 5	(S + (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 α as a finite parser state.
 - Parser state is computed by a DFA that reads the stack σ .
 - 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.

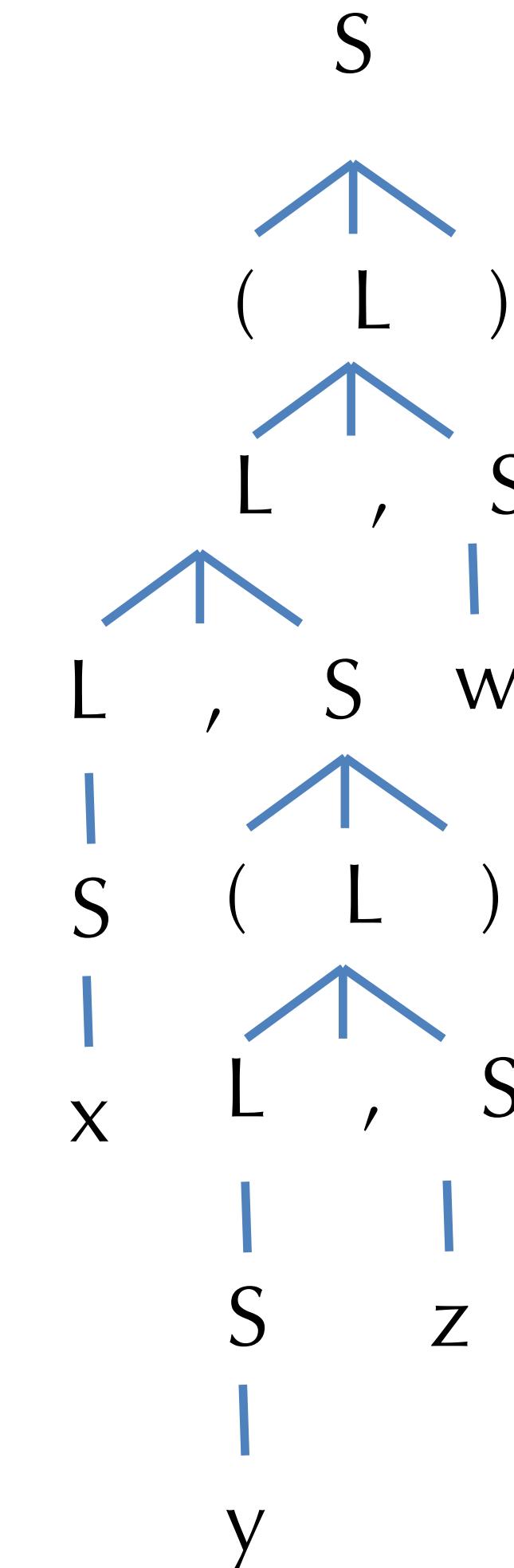
Example LR(0) Grammar: Tuples

- Example grammar for non-empty tuples and identifiers:

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

- 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.

$$\begin{array}{l} S \mapsto (L) \mid id \\ L \mapsto S \mid L , S \end{array}$$

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

- Reduce: Replace symbols γ at top of stack with nonterminal X such that $X \mapsto \gamma$ is a production. (pop γ , 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 \rightarrow id
(S	, (y, z), w)	reduce L \rightarrow S
(L	, (y, z), w)	shift ,
(L,	(y, z), w)	shift (
(L, (y, z), w)	shift y
(L, (y	, z), w)	reduce S \rightarrow id
(L, (S	, z), w)	reduce L \rightarrow S
(L, (L	, z), w)	shift ,
(L, (L,	z), w)	shift z
(L, (L, z), w)	reduce S \rightarrow id
(L, (L, S), w)	reduce L \rightarrow L, S
(L, (L), w)	shift)
(L, (L)	, w)	reduce S \rightarrow (L)
(L, S	, w)	reduce L \rightarrow L, S
(L	, w)	shift ,
(L,	w)	shift w
(L, w)	reduce S \rightarrow id
(L, S)	reduce L \rightarrow L, S
(L)	shift)
(L))	reduce S \rightarrow (L)
S		

$$S \xrightarrow{} (L) \mid id$$

$$L \xrightarrow{} S \mid L, S$$

Action Selection Problem

- Given a stack σ and a look-ahead symbol b , should the parser:
 - Shift b onto the stack (new stack is σb)
 - Reduce a production $X \mapsto \gamma$, assuming that $\sigma = \alpha\gamma$ (new stack is αX)?
- Sometimes the parser can reduce but shouldn't
 - For example, $X \mapsto \epsilon$ can *always* be reduced
- Sometimes the stack can be reduced in different ways
- Main idea: decide what to do based on a *prefix* α of the stack plus the look-ahead symbol.
 - The prefix α is different for different possible reductions since in productions $X \mapsto \gamma$ and $Y \mapsto \beta$, γ and β 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

$$\begin{array}{l} S \rightarrow (L) \mid id \\ L \rightarrow S \mid L , S \end{array}$$

- Example items: $S \rightarrow .(L)$ or $S \rightarrow (. L)$ or $L \rightarrow S.$
- Intuition:
 - Stuff before the ‘.’ is already on the stack (beginnings of possible γ 's to be reduced)
 - Stuff after the ‘.’ is what might be seen next
 - The prefixes α are represented by the state itself

Constructing the DFA: Start state & Closure

- First step: Add a new production $S' \rightarrow S\$$ to the grammar
- Start state of the DFA = empty stack, so it contains the item:
 $S' \rightarrow .S\$$
- 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: $\text{CLOSURE}(\{S' \rightarrow .S\$}\}) = \{S' \rightarrow .S\$, S \rightarrow .(L), S \rightarrow .id\}$
- Resulting “closed state” contains the set of all possible productions that might be reduced next.

$$\begin{array}{l} S' \rightarrow S\$ \\ S \rightarrow (L) \mid id \\ L \rightarrow S \mid L , S \end{array}$$

Example: Constructing the DFA

$S' \xrightarrow{} .S\$$

$S' \xrightarrow{} S\$$
 $S \xrightarrow{} (L) \mid id$
 $L \xrightarrow{} S \mid L, S$

- First, we construct a state with the initial item $S' \xrightarrow{} .S\$$

Example: Constructing the DFA

$S' \rightarrow .S\$$

$S \rightarrow .(L)$

$S \rightarrow .id$

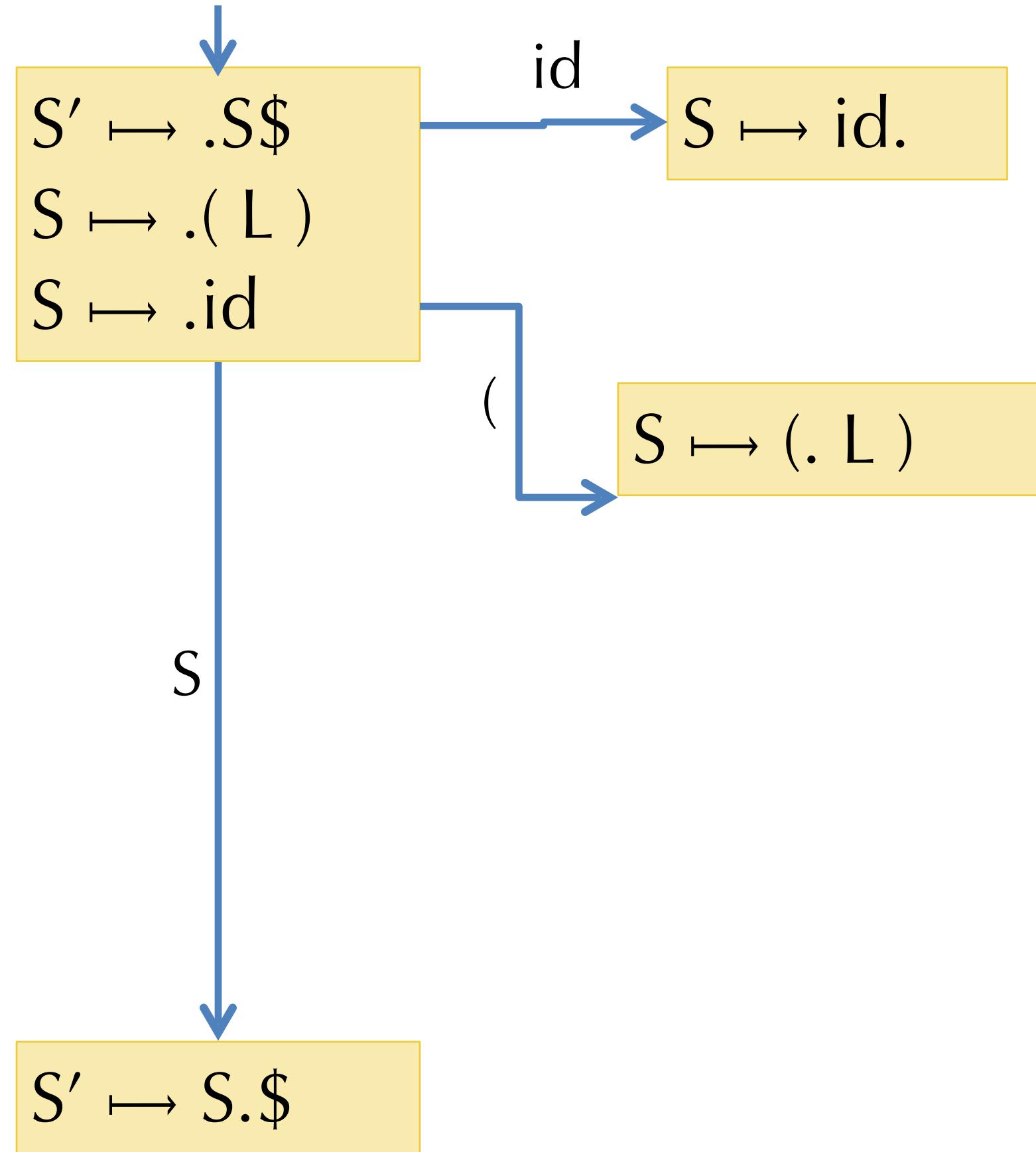
$S' \rightarrow S\$$

$S \rightarrow (L) \mid id$

$L \rightarrow S \mid L, S$

- Next, we take the closure of that state:
 $CLOSURE(\{S' \rightarrow .S\$}\}) = \{S' \rightarrow .S\$, S \rightarrow .(L), S \rightarrow .id\}$
- In the set of items, the nonterminal S appears after the ' $.$ '
- So we add items for each S production in the grammar

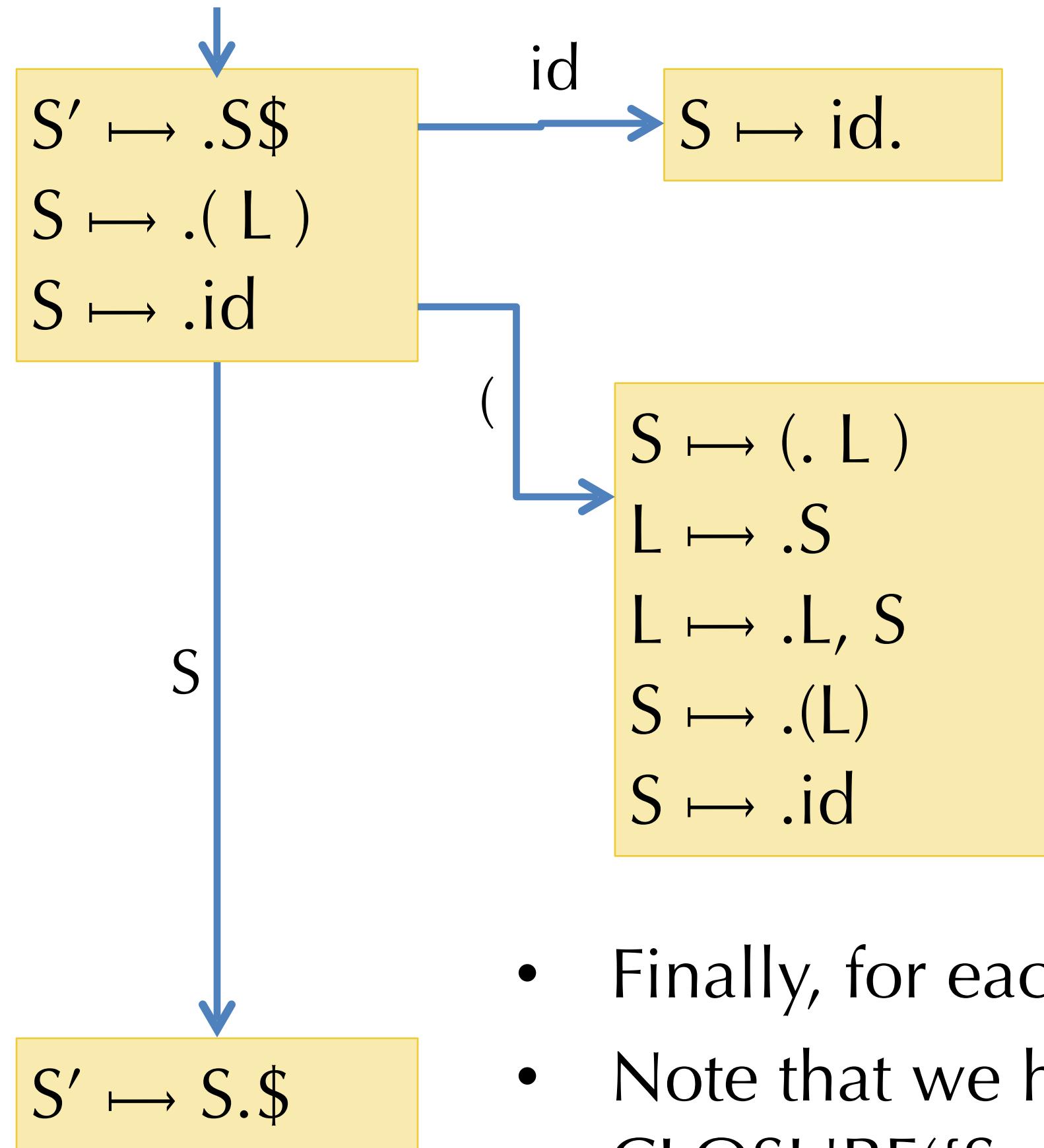
Example: Constructing the DFA



$S' \rightarrow S\$$
 $S \rightarrow (L) \mid id$
 $L \rightarrow S \mid L, S$

- 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



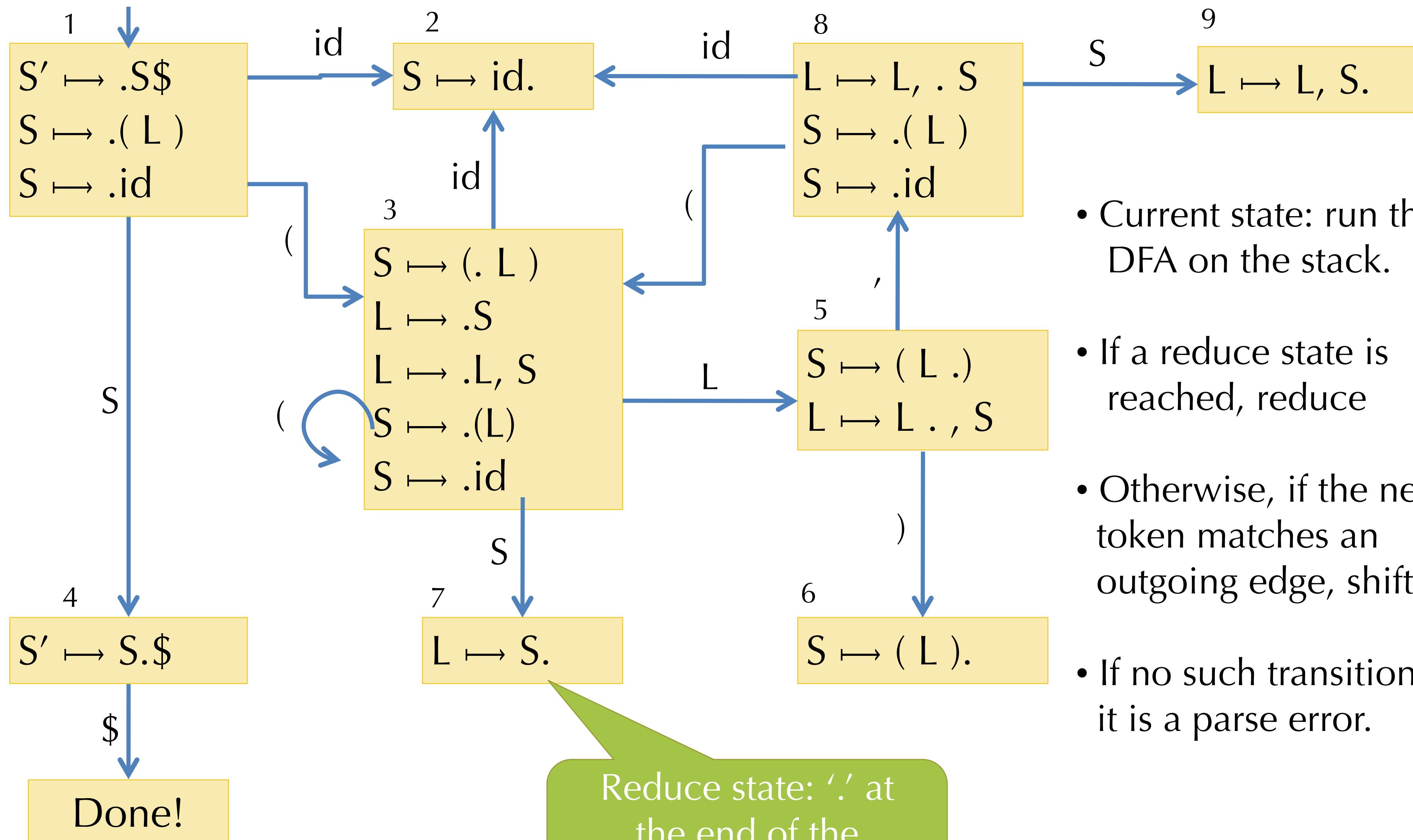
$S' \rightarrow S\$$

$S \rightarrow (L) \mid id$

$L \rightarrow S \mid L, S$

- Finally, for each new state, we take the closure.
- Note that we have to perform two iterations to compute $CLOSURE(\{S \rightarrow (. L)\})$
 - First iteration adds $L \rightarrow .S$ and $L \rightarrow .L, S$
 - Second iteration adds $S \rightarrow .(L)$ and $S \rightarrow .id$

Full DFA for the Example



Reduce state: '.' at
the end of the
production

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, $X \rightarrow \gamma$ with stack $\alpha\gamma$, pop γ 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 $X \rightarrow \gamma$, pop stack to reveal the state too:
e.g. From stack $_1(3(3L_5)_6$ reduce $S \rightarrow (L)$ 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 $X \rightarrow \gamma$
 - First pop γ 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→id	S→id	S→id	S→id	S→id		
3	s3		s2			g7	g5
4					DONE		
5		s6		s8			
6	S → (L)						
7	L → S	L → S	L → S	L → S	L → S		
8	s3		s2			g9	
9	L → L,S						

s_x = shift and goto state x

g_x = goto state x (used when we reduce)

Example

- Parse the token stream: $(x, (y, z), w)\$$

Stack	Stream	Action (according to table)
ϵ_1	$(x, (y, z), w)\$$	s3
$\epsilon_1(3$	$x, (y, z), w)\$$	s2
$\epsilon_1(3x_2$	$, (y, z), w)\$$	Reduce: $S \rightarrow id$
$\epsilon_1(3S$	$, (y, z), w)\$$	g7 (from state 3 follow S)
$\epsilon_1(3S_7$	$, (y, z), w)\$$	Reduce: $L \rightarrow S$
$\epsilon_1(3L$	$, (y, z), w)\$$	g5 (from state 3 follow L)
$\epsilon_1(3L_5$	$, (y, z), w)\$$	s8
$\epsilon_1(3L_{5,8}$	$(y, z), w)\$$	s3
$\epsilon_1(3L_{5,8}(3$	$y, z), w)\$$	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

$S \rightarrow (L).$

shift/reduce

$S \rightarrow (L).$

$L \rightarrow .L , S$

reduce/reduce

$S \rightarrow L , S.$

$S \rightarrow , S.$

- 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?
- Ambiguities in associativity/precedence usually lead to shift/reduce conflicts.

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 $\text{Follow}(A)$ for the given grammar
 - if the lookahead symbol is in $\text{Follow}(A)$, 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:
$$A \xrightarrow{\cdot} \alpha.\beta , \mathcal{L}$$
- LR(1) closure is a little more complex:
- Form the set of items just as for LR(0) algorithm.
- Whenever a new item $C \xrightarrow{\cdot} .\gamma$ is added because $A \xrightarrow{\cdot} \beta.C\delta , \mathcal{L}$ 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 δ)
 2. If δ is itself ϵ or can derive ϵ (i.e. it is nullable), then the look-ahead \mathcal{M} also contains \mathcal{L}

Example Closure

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

- Start item: $S' \rightarrow .S\$, \{\}$
- Since S is to the right of a '.', add:

$$S \rightarrow .E + S , \{ \$ \}$$

Note: $\{ \$ \}$ is FIRST(\$)

$$S \rightarrow .E , \{ \$ \}$$

- Need to keep closing, since E appears to the right of a '.' in ' $.E + S'$:

$$E \rightarrow .\text{number} , \{ + \}$$

Note: + added for reason 1

$$E \rightarrow .(S) , \{ + \}$$

FIRST(+ S) = {+}

- Because E also appears to the right of '.' in ' $.E'$ we get:

$$E \rightarrow .\text{number} , \{ \$ \}$$

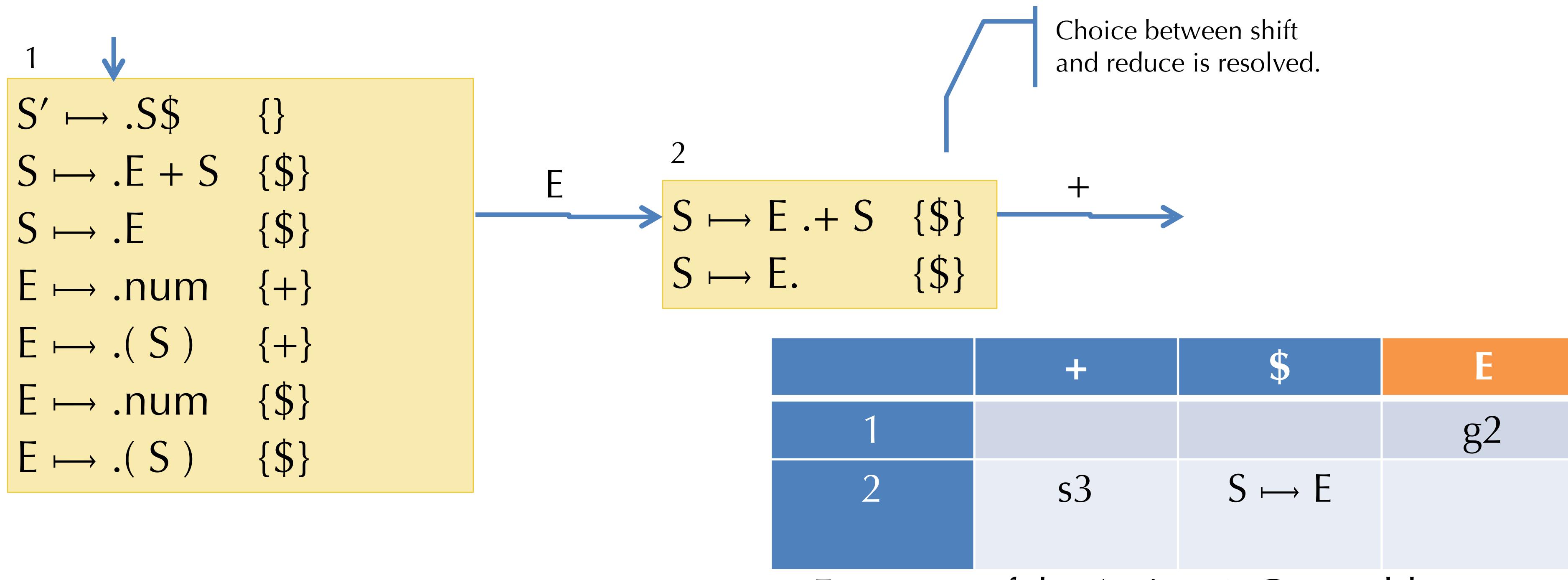
Note: \$ added for reason 2

$$E \rightarrow .(S) , \{ \$ \}$$

δ is ϵ

- All items are distinct, so we're done

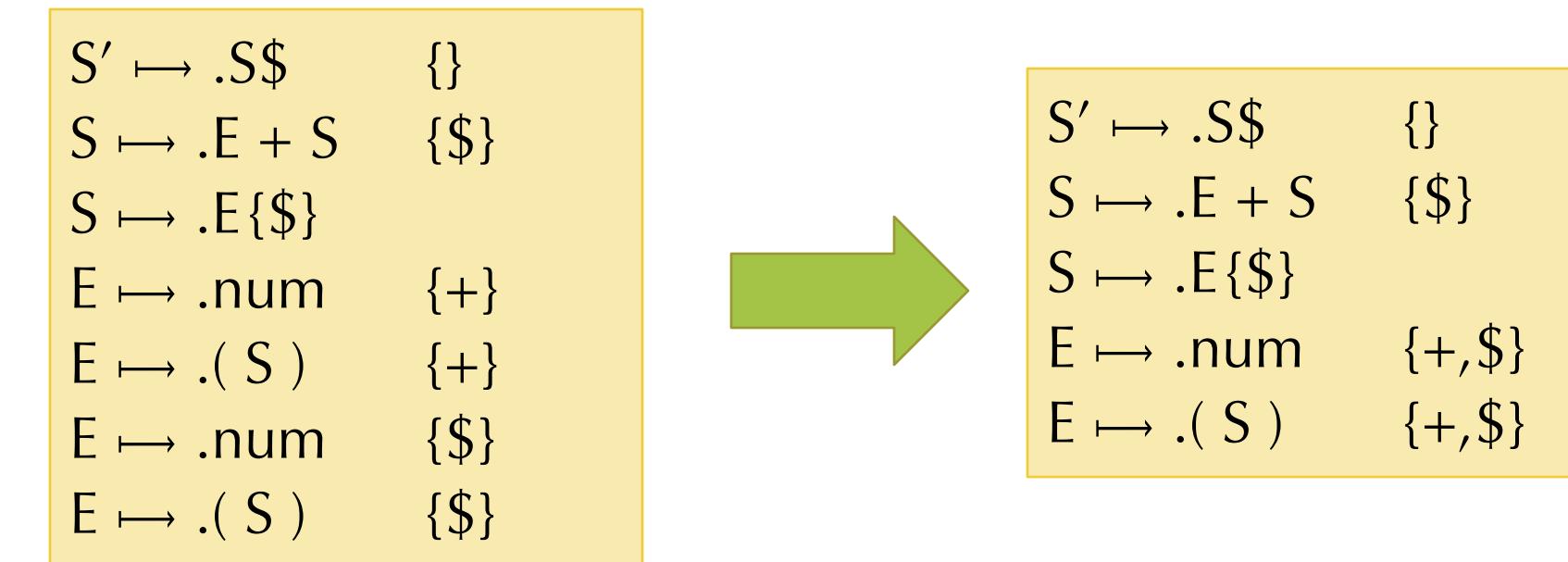
Using the DFA



- 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 ‘.’

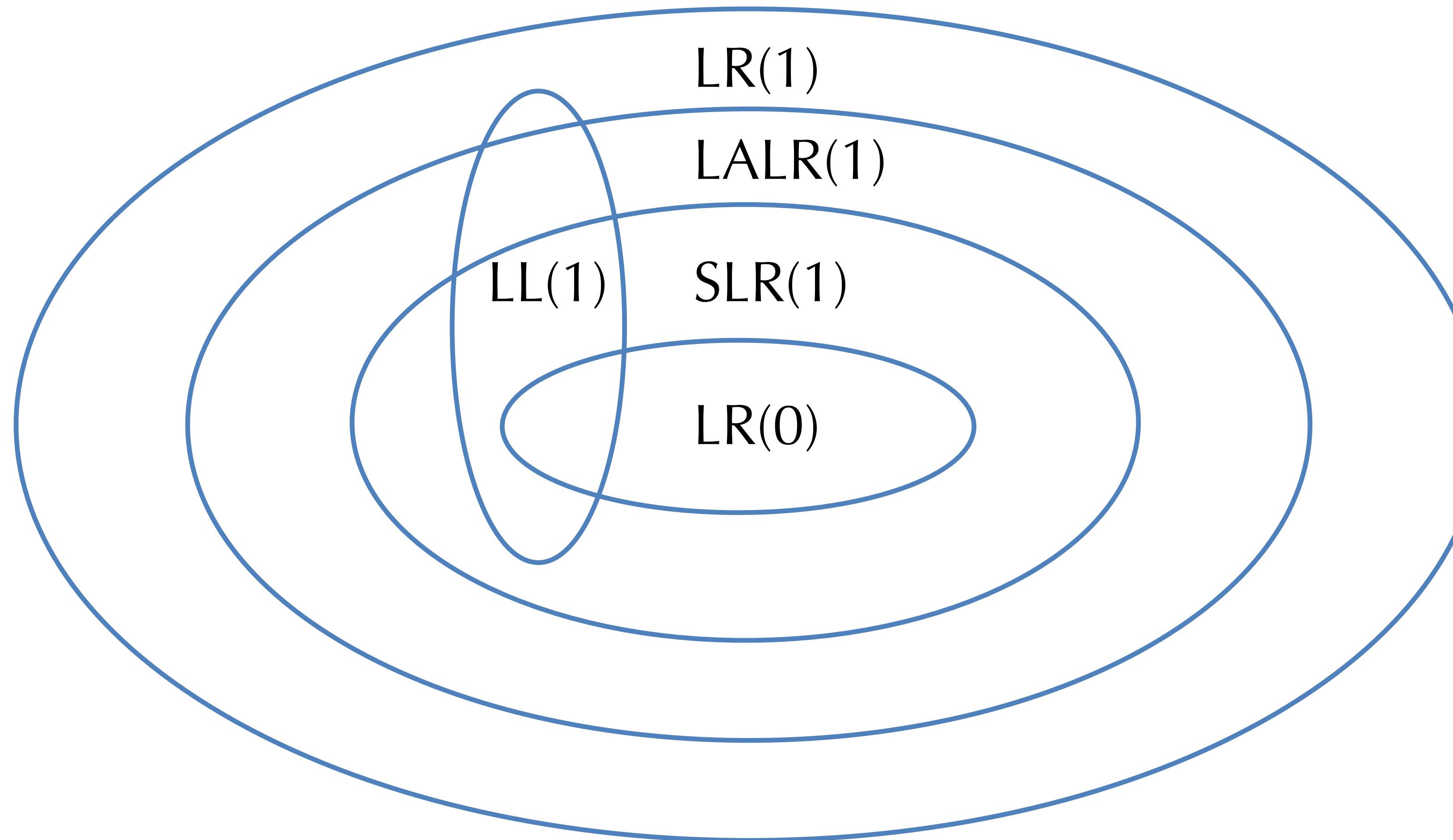
LR variants

- 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.
 - Modern implementations (e.g., menhir) directly generate code
- 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, ocamlyacc
- 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 control flow: 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

Where to Learn More?

Academic Research

- Michigan PL and Software engineering (MPLSE)
 - 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 (**Principles** of Programming Languages)
 - ICFP (**Functional** Programming)
 - OOPSLA (**Object-oriented...**)
 - CC (Compiler Construction)
 - Many more...

Thank You

- Thanks to Yuchen and David for their great work as course staff.
- 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.