RICE

Ignore § 2.4.4 in EaC2e.

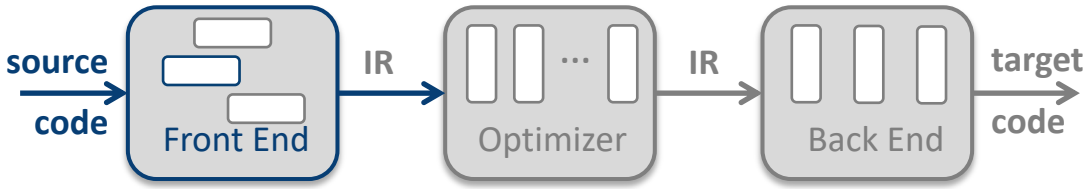
I will post a replacement secion on the course web site.

COMP 412
FALL 2018

Lexical Analysis, III

Comp 412

Minor corrections applied after class



Chapter 2 in EaC2e

The Plan for Scanner Construction

RE → NFA *(Thompson's construction)*

- Build an **NFA** for each term in the **RE**
- Combine them in patterns that model the operators

NFA → DFA *(Subset construction)*

- Build a **DFA** that simulates the **NFA**

DFA → Minimal DFA

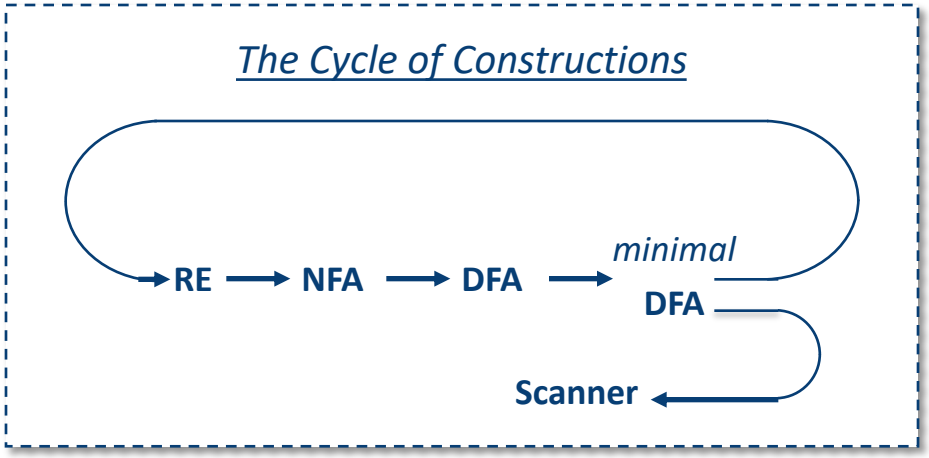
- Hopcroft's algorithm
- Brzozowski's algorithm

Minimal **DFA** → Scanner

- See § 2.5 in EaC2e

DFA → RE

- All pairs, all paths problem
- Union together paths from s_0 to a final state



DFA Minimization

The Big Picture

- Discover sets of behaviorally equivalent states in the **DFA**
- Represent each such set with a single new state

Two states s_i and s_j are **behaviorally equivalent** if and only if:

Recursive definition

- $\forall\ c \in \Sigma$, transitions from s_i & s_j on c lead to equivalent states
- The set of paths leading from s_i & s_j are equivalent

A **partition** P of a set S:

- A collection of subsets of P such that each state s is in exactly one $p_i \in P$

The algorithm iteratively constructs partitions of the **DFA**'s set of states

We want a partition $P = \{ p_0, p_1, p_2, \ldots p_n \}$ of D that has two properties:

1. If d_i & $d_j \in p_s$ and c takes $d_i \rightarrow d_x$ and $d_j \rightarrow d_y$, then d_x & $d_y \in p_t$, $\forall\ c, i, j, s, t$
2. If d_i & $d_j \in p_s$ and $d_i \in F$ then $d_j \in F$

D is the set of states for the **DFA**: $(D,\Sigma,\delta,s_0,D_A)$

DFA Minimization

Maximally sized sets ⇒ minimal number of sets

Details of the algorithm

- Group states into maximally-sized initial sets, *optimistically* **(property 2)**
- Iteratively subdivide those sets, based on transition graph **(property 1)**
- States that remain grouped together are equivalent

Initial partition: P_0 has two sets: $\{D_A\}$ & $\{D - D_A\}$

$D = (D, \Sigma, \delta, s_0, D_A)$

final states / other states

Property 1 provides the basis for refining, or splitting, the sets

- Assume s_i & $s_j \in p_s$, and $\delta(s_i, \underline{a}) = s_x$, & $\delta(s_j, \underline{a}) = s_y$
- If s_x & s_y are not in the same set p_t, then p_s must be split
  - **COROLLARY:** s_i has transition on $\underline{a}$, s_j does not ⇒ $\underline{a}$ splits p_s
- A single state in a **DFA** cannot have two transitions on $\underline{a}$
  - Each p_s will become a **DFA** state

Algorithm actually works backward; it looks at what transitions enter *p* on character *c*, and uses that to split the partition *q* where those edges begin.

DFA Minimization Algorithm (Worklist version)

```
Worklist  ← { D_A , { D – D_A } }
Partition ← { D_A , { D – D_A } }
While (Worklist ≠ ∅) do
    select a set S from Worklist and remove it
    for each α ∈ Σ do
        Image ← { x | δ(x, α) ∈ S }
        for each q ∈ Partition that has a state in Image do
            q1 ← q ∩ Image
            q2 ← q – q1
            if q2 ≠ ∅ then
                remove q from Partition
                Partition ← Partition ∪ q1 ∪ q2
            if q ∈ Worklist then
                remove q from Worklist
                Worklist ← Worklist ∪ q1 ∪ q2
            else if |q1| ≤ |q2|
                then Worklist ← Worklist ∪ q1
                else Worklist ← Worklist ∪ q2
            if s = q then
                break;   // cannot keep working on s
```

Image is the set of states that have a transition into *S* on α: $\delta^{-1}(S,\alpha)$

"split q"

q_1 is the subset of q that transitions to S on α

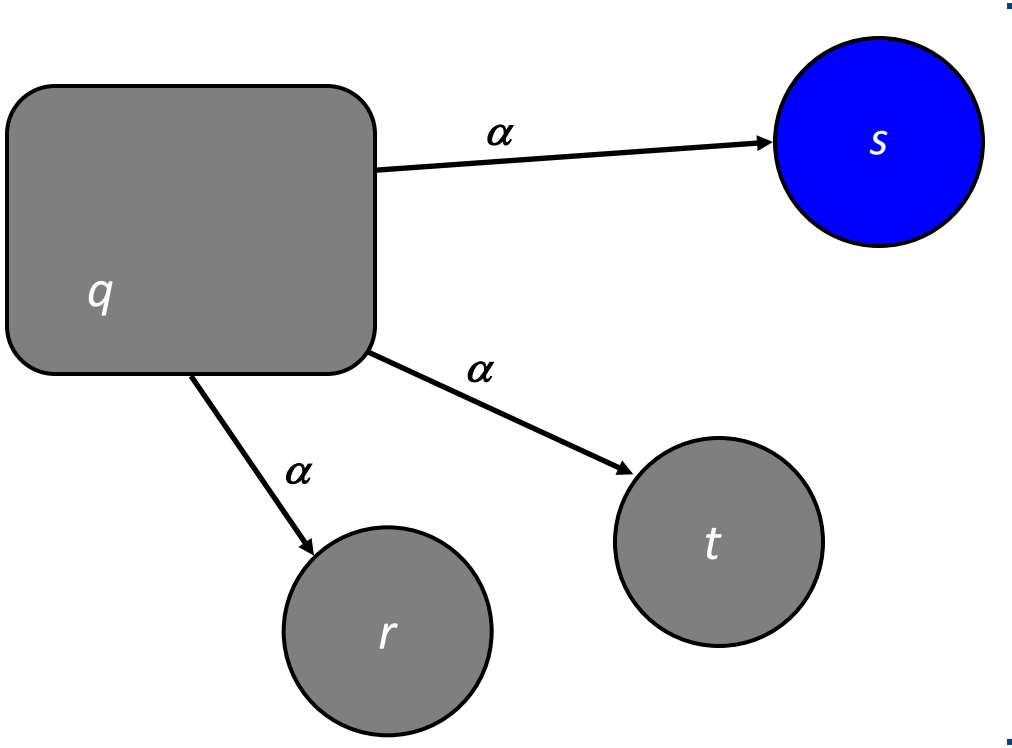
q_2 is the rest of q

adjust Worklist

And, as an implementation nit, if we just split *S* — that is, *S* was *q* & it split — we need a new *S*

Key Idea: Splitting Q Around Transitions on α

Partitioning Q around S



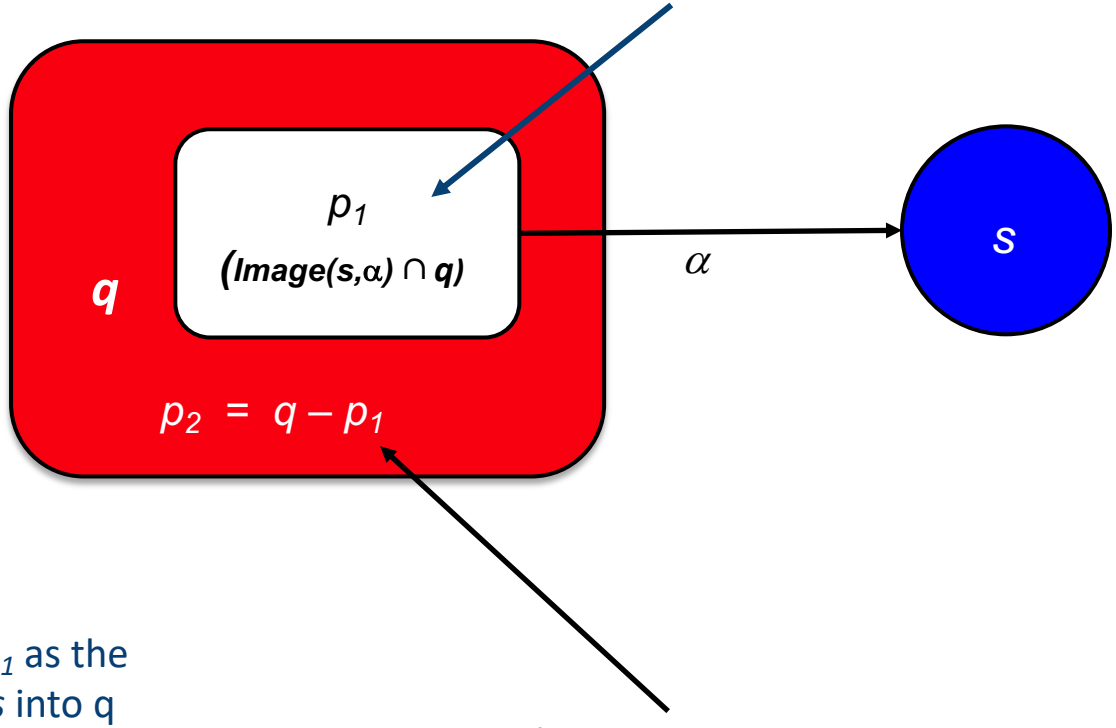
Assume that *q, r, s,* & *t* are sets in the current approximation to the final partition

q has transitions on α to *r, s,* & *t,* so it must split around α

As the algorithm considers s and α, it will split q.

Key Idea: Splitting q around *s* and α

Find maximal subset of *q* (p_1) that has an α-transition into s



Think of p_1 as the image of *s* into q under the inverse of the transition function:

$p_1 \leftarrow \delta^{-1}(s, \alpha) \cap q$

p_2 must have an α-transition to one or more other states in one or more other partitions (e.g., r & s), or states with no α-transitions.

Otherwise, q does not split!

DFA Minimization Algorithm (Worklist version)

```
Worklist  ← { D_A , { D – D_A } }
Partition ← { D_A , { D – D_A } }
While (Worklist ≠ ∅) do
    select a set S from Worklist and remove it
    for each α ∈ Σ do
        Image ← { x | δ(x, α) ∈ S }
        for each q ∈ Partition that has a state in Image do
            q1 ← q ∩ Image
            q2 ← q – q1
            if q2 ≠ ∅ then
                remove q from Partition
                Partition ← Partition ∪ q1 ∪ q2
            if q ∈ Worklist then
                remove q from Worklist
                Worklist ← Worklist ∪ q1 ∪ q2
            else if |q1| ≤ |q2|
                then Worklist ← Worklist ∪ q1
                else Worklist ← Worklist ∪ q2
            if s = q then
                break;   // cannot keep working on s
```

Projection is the set of states that have a transition into *S* on α :

$\delta^{-1}(S,\alpha)$

p_1 is the subset of q that transitions to S on α

p_2 is the rest of q

"split q"

adjust Worklist

And, as an implementation nit, if we just split *S* — that is, *S* was *q* & it split — we need a new *S*

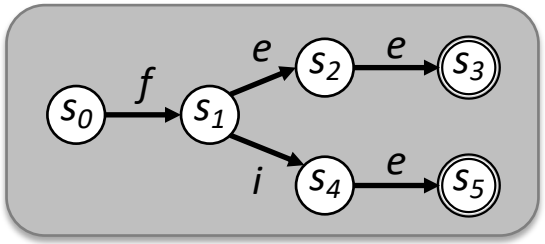
DFA Minimization Algorithm (Worklist version)

One last hack …

```
Worklist  ← { D_A , { D – D_A } }
Partition ← { D_A , { D – D_A } }

While (Worklist ≠ ∅) do
    select a set S from Worklist and remove it
    for each α ∈ Σ do
        Image ← { x | δ(x, α) ∈ S }
        for each q ∈ Partition that has a state in Image do
            q1 ← q ∩ Image
            q2 ← q – q1
            if q2 ≠ ∅ then
                remove q from Partition
                Partition ← Partition ∪ q1 ∪ q2
                if q ∈ Worklist then
                    remove q from Worklist
                    Worklist ← Worklist ∪ q1 ∪ q2
                else if |q1| ≤ |q2|
                    then Worklist ← Worklist ∪ q1
                    else Worklist ← Worklist ∪ q2
                if s = q then
                    break;   // cannot keep working on s
```

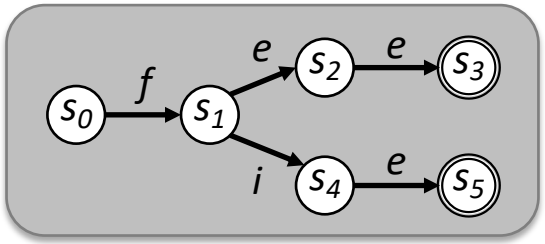
If q is a singleton, we can skip the body of the loop because a singleton cannot split.

A Detailed Example



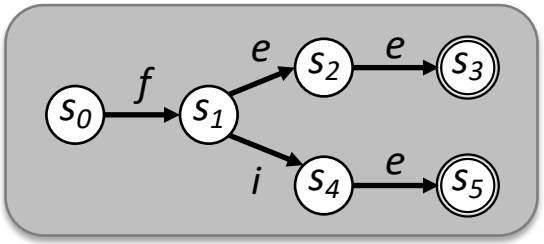
Step	Partition	W'list	s	c	Image	q	q_1	q_2	Action
0	p_0: {3,5}, p_1: {0,1,2,4}	p_0, p_1	—	—	—	—	—	—	—

A Detailed Example



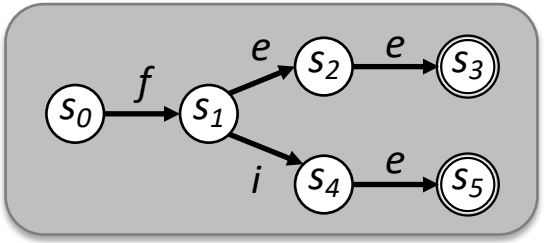
Step	Partition	W'list	s	c	Image	q	q_1	q_2	Action
0	p_0: {3,5}, p_1: {0,1,2,4}	p_0, p_1	—	—	—	—	—	—	—
1	p_0: {3,5}, p_1: {0,1,2,4}	p_1	p_0	e	s_2, s_4	p_1	s_2, s_4	s_0, s_1	split p_1

A Detailed Example



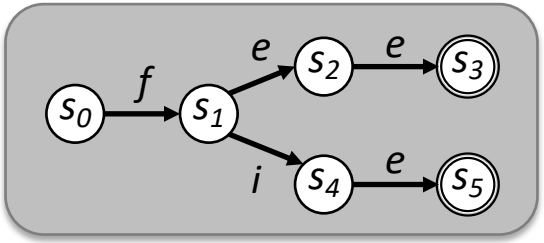
Step	Partition	W'list	s	c	Image	q	q_1	q_2	Action
0	p_0: {3,5}, p_1: {0,1,2,4}	p_0, p_1	—	—	—	—	—	—	—
1	p_0: {3,5}, p_1: {0,1,2,4}	p_1	p_0	e	s_2, s_4	p_1	s_2, s_4	s_0, s_1	split p_1
	p_0: {3,5}, p_2: {2,4}, p_3: {0,1}	p_2, p_3	p_0	f	∅	∅	∅	∅	none
		p_2, p_3	p_0	i	∅	∅	∅	∅	none

A Detailed Example



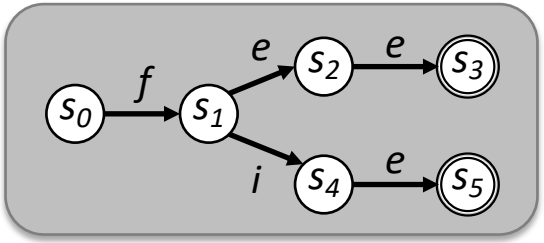
Step	Partition	W'list	s	c	Image	q	q_1	q_2	Action
0	p_0: {3,5}, p_1: {0,1,2,4}	p_0, p_1	—	—	—	—	—	—	—
1	p_0: {3,5}, p_1: {0,1,2,4}	p_1	p_0	e	s_2, s_4	p_1	s_2, s_4	s_0, s_1	split p_1
	p_0: {3,5}, p_2: {2,4}, p_3: {0,1}	p_2, p_3	p_0	f	∅	∅	∅	∅	none
		p_2, p_3	p_0	i	∅	∅	∅	∅	none
2	p_0: {3,5}, p_2: {2,4}, p_3: {0,1}	p_3	p_2	e	s_1	p_3	s_1	s_0	split p_3

A Detailed Example



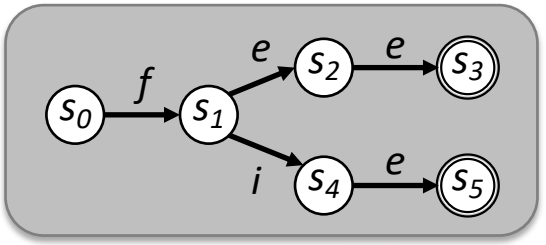
Step	Partition	W'list	s	c	Image	q	q_1	q_2	Action
0	p_0: {3,5}, p_1: {0,1,2,4}	p_0, p_1	—	—	—	—	—	—	—
1	p_0: {3,5}, p_1: {0,1,2,4}	p_1	p_0	e	s_2, s_4	p_1	s_2, s_4	s_0, s_1	**split p_1**
	p_0: {3,5}, p_2: {2,4}, p_3: {0,1}	p_2, p_3	p_0	f	∅	∅	∅	∅	*none*
		p_2, p_3	p_0	i	∅	∅	∅	∅	*none*
2	p_0: {3,5}, p_2: {2,4}, p_3: {0,1}	p_3	p_2	e	s_1	p_3	s_1	s_0	**split p_3**
	p_0: {3,5}, p_2: {2,4}, p_4: {1}, p_5: {0}	p_4, p_5	p_2	f	∅	∅	∅	∅	*none*
		p_4, p_5	p_2	i	s_1	p_4	s_1	∅	*none*

A Detailed Example



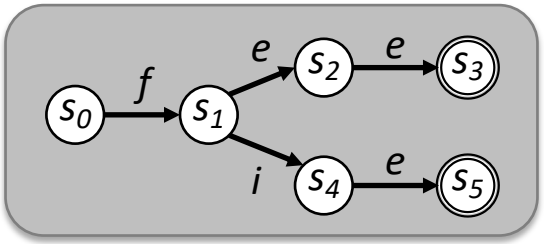
Step	Partition	W'list	s	c	Image	q	q_1	q_2	Action
0	p_0: {3,5}, p_1: {0,1,2,4}	p_0, p_1	—	—	—	—	—	—	—
1	p_0: {3,5}, p_1: {0,1,2,4}	p_1	p_0	e	s_2, s_4	p_1	s_2, s_4	s_0, s_1	**split p_1**
	p_0: {3,5}, p_2: {2,4}, p_3: {0,1}	p_2, p_3	p_0	f	∅	∅	∅	∅	*none*
		p_2, p_3	p_0	i	∅	∅	∅	∅	*none*
2	p_0: {3,5}, p_2: {2,4}, p_3: {0,1}	p_3	p_2	e	s_1	p_3	s_1	s_0	**split p_3**
	p_0: {3,5}, p_2: {2,4}, p_4: {1}, p_5: {0}	p_4, p_5	p_2	f	∅	∅	∅	∅	*none*
		p_4, p_5	p_2	i	s_1	p_4	s_1	∅	*none*
3		p_5	p_4	e	∅	∅	∅	∅	*none*

A Detailed Example



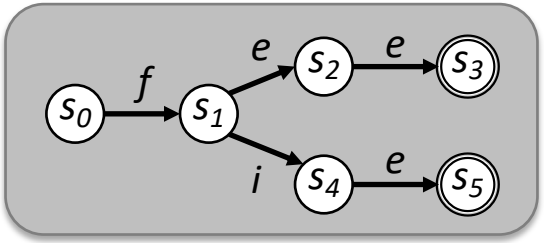
Step	Partition	W'list	s	c	Image	q	q_1	q_2	Action
0	p_0: {3,5}, p_1: {0,1,2,4}	p_0, p_1	—	—	—	—	—	—	—
1	p_0: {3,5}, p_1: {0,1,2,4}	p_1	p_0	e	s_2, s_4	p_1	s_2, s_4	s_0, s_1	**split p_1**
	p_0: {3,5}, p_2: {2,4}, p_3: {0,1}	p_2, p_3	p_0	f	∅	∅	∅	∅	*none*
		p_2, p_3	p_0	i	∅	∅	∅	∅	*none*
2	p_0: {3,5}, p_2: {2,4}, p_3: {0,1}	p_3	p_2	e	s_1	p_3	s_1	s_0	**split p_3**
	p_0: {3,5}, p_2: {2,4}, p_4: {1}, p_5: {0}	p_4, p_5	p_2	f	∅	∅	∅	∅	*none*
		p_4, p_5	p_2	i	s_1	p_4	s_1	∅	*none*
3	p_0: {3,5}, p_2: {2,4}, p_4: {1}, p_5: {0}	p_5	p_4	e	∅	∅	∅	∅	*none*
		p_5	p_4	f	s_0	p_5	s_0	∅	*none*

A Detailed Example



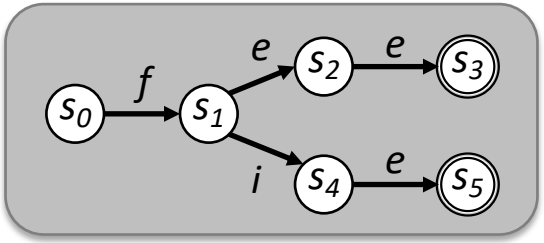
Step	Partition	W'list	s	c	Image	q	q_1	q_2	Action
0	p_0: {3,5}, p_1: {0,1,2,4}	p_0, p_1	—	—	—	—	—	—	—
1	p_0: {3,5}, p_1: {0,1,2,4}	p_1	p_0	e	s_2, s_4	p_1	s_2, s_4	s_0, s_1	**split p_1**
	p_0: {3,5}, p_2: {2,4}, p_3: {0,1}	p_2, p_3	p_0	f	∅	∅	∅	∅	*none*
		p_2, p_3	p_0	i	∅	∅	∅	∅	*none*
2	p_0: {3,5}, p_2: {2,4}, p_3: {0,1}	p_3	p_2	e	s_1	p_3	s_1	s_0	**split p_3**
	p_0: {3,5}, p_2: {2,4}, p_4: {1}, p_5: {0}	p_4, p_5	p_2	f	∅	∅	∅	∅	*none*
		p_4, p_5	p_2	i	s_1	p_4	s_1	∅	*none*
3	p_0: {3,5}, p_2: {2,4}, p_4: {1}, p_5: {0}	p_5	p_4	e	∅	∅	∅	∅	*none*
		p_5	p_4	f	s_0	p_5	s_0	∅	*none*
		p_5	p_4	i	∅	∅	∅	∅	*none*

A Detailed Example



Step	Partition	W'list	s	c	Image	q	q_1	q_2	Action
0	p_0: {3,5}, p_1: {0,1,2,4}	p_0, p_1	—	—	—	—	—	—	—
1	p_0: {3,5}, p_1: {0,1,2,4}	p_1	p_0	e	s_2, s_4	p_1	s_2, s_4	s_0, s_1	**split p_1**
	p_0: {3,5}, p_2: {2,4}, p_3: {0,1}	p_2, p_3	p_0	f	∅	∅	∅	∅	*none*
		p_2, p_3	p_0	i	∅	∅	∅	∅	*none*
2	p_0: {3,5}, p_2: {2,4}, p_3: {0,1}	p_3	p_2	e	s_1	p_3	s_1	s_0	**split p_3**
	p_0: {3,5}, p_2: {2,4}, p_4: {1}, p_5: {0}	p_4, p_5	p_2	f	∅	∅	∅	∅	*none*
		p_4, p_5	p_2	i	s_1	p_4	s_1	∅	*none*
3	p_0: {3,5}, p_2: {2,4}, p_4: {1}, p_5: {0}	p_5	p_4	e	∅	∅	∅	∅	*none*
		p_5	p_4	f	s_0	p_5	s_0	∅	*none*
		p_5	p_4	i	∅	∅	∅	∅	*none*
4	p_0: {3,5}, p_2: {2,4}, p_4: {1}, p_5: {0}	∅	p_5	e	∅	∅	∅	∅	*none*
		∅	p_5	f	∅	∅	∅	∅	*none*
		∅	p_5	i	∅	∅	∅	∅	*none*

A Detailed Example

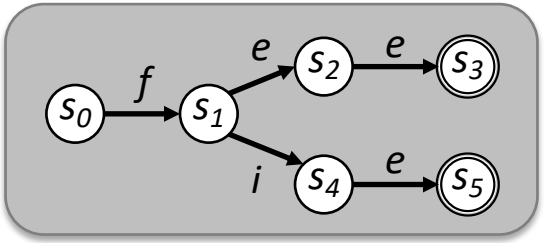


Step	Partition	W'list	s	c	Image	q	q_1	q_2	Action
0	p_0: {3,5}, p_1: {0,1,2,4}	p_0, p_1	—	—	—	—	—	—	—
1	p_0: {3,5}, p_1: {0,1,2,4}	p_1	p_0	e	s_2, s_4	p_1	s_2, s_4	s_0, s_1	**split p_1**
	p_0: {3,5}, p_2: {2,4}, p_3: {0,1}	p_2, p_3	p_0	f	∅	∅	∅	∅	*none*
		p_2, p_3	p_0	i	∅	∅	∅	∅	*none*
2	p_0: {3,5}, p_2: {2,4}, p_3: {0,1}	p_3	p_2	e	s_1	p_3	s_1	s_0	**split p_3**
	p_0: {3,5}, p_2: {2,4}, p_4: {1}, p_5: {0}	p_4, p_5	p_2	f	∅	∅	∅	∅	*none*
		p_4, p_5	p_2	i	s_1	p_4	s_1	∅	*none*
3	p_0: {3,5}, p_2: {2,4}, p_4: {1}, p_5: {0}	p_5	p_4	e	∅	∅	∅	∅	*none*
		p_5	p_4	f	s_0	p_5	s_0	∅	*none*
						∅	∅	∅	*none*
						∅	∅	∅	*none*
						∅	∅	∅	*none*
						∅	∅	∅	*none*

Reconstructing the DFA

- Each set in Partition forms a state
- For each line in the table where both q_1 and s, have values, *add an edge from* q_1 to s labelled c

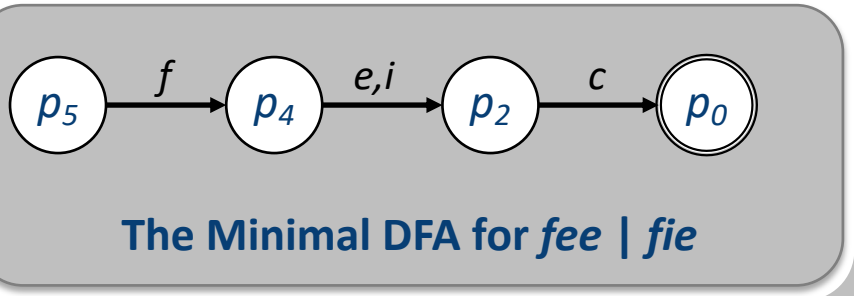
A Detailed Example



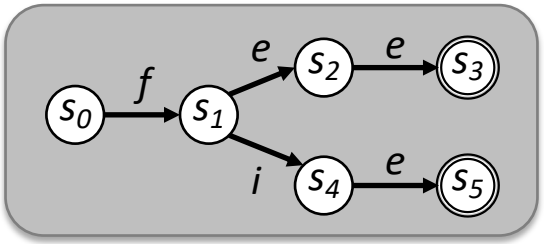
Step	Partition	W'list	s	c	Image	q	q_1	q_2	Action
0	p_0: {3,5}, p_1: {0,1,2,4}	p_0, p_1	—	—	—	—	—	—	—
1	p_0: {3,5}, p_1: {0,1,2,4}	p_1	p_0	e	s_2, s_4	p_1	s_2, s_4	s_0, s_1	**split p_1**
	p_0: {3,5}, p_2: {2,4}, p_3: {0,1}	p_2, p_3	p_0	f	∅	∅	∅	∅	*none*
		p_2, p_3	p_0	i	∅	∅	∅	∅	*none*
2	p_0: {3,5}, p_2: {2,4}, p_3: {0,1}	p_3	p_2	e	s_1	p_3	s_1	s_0	**split p_3**
	p_0: {3,5}, p_2: {2,4}, p_4: {1}, p_5: {0}	p_4, p_5	p_2	f	∅	∅	∅	∅	*none*
		p_4, p_5	p_2	i	s_1	p_4	s_1	∅	*none*
3	p_0: {3,5}, p_2: {2,4}, p_4: {1}, p_5: {0}	p_5	p_4	e	∅	∅	∅	∅	*none*
		p_5	p_4	f	s_0	p_5	s_0	∅	*none*
				i	∅	∅	∅	∅	*none*
				e	∅	∅	∅	∅	*none*
				f	∅	∅	∅	∅	*none*
				i	∅	∅	∅	∅	*none*

Reconstructing the DFA

- Each set in Partition forms a state
- For each line in the table where $q_1 \neq \emptyset$, *add an edge from* q_1 to s labelled c



The Minimal DFA for *fee | fie*



Step	Partition	W list	s	c	Image	q	q_1	q_2	Action
0	p_0: {3,5}, p_1: {0,1,2,4}	p_0, p_1	—	—	—	—	—	—	—
1	p_0: {3,5}, p_1: {0,1,2,4}	p_1	p_0	e	s_2, s_4	p_1	s_2, s_4	s_0, s_1	**split p_1**
	p_0: {3,5}, p_2: {2,4}, p_3: {0,1}	p_2, p_3	p_0	f	∅	∅	∅	∅	none
		p_2, p_3	p_0	i	∅	∅	∅	∅	none
2	p_0: {3,5}, p_2: {2,4}, p_3: {0,1}	p_3	p_2	e	s_1	p_3	s_1	s_0	**split p_3**
	p_0: {3,5}, p_2: {2,4}, p_4: {1}, p_5: {0}	p_4, p_5	p_2	f	∅	∅	∅	∅	none
		p_4, p_5	p_2	i	s_1	p_4	s_1	∅	none
3	p_0: {3,5}, p_2: {2,4}, p_4: {1}, p_5: {0}	p_5	p_4	e	∅	∅	∅	∅	none
		p_5	p_4	f	s_0	p_5	s_0	∅	none
				i	∅	∅	∅	∅	none
				e	∅	∅	∅	∅	none
				f	∅	∅	∅	∅	none
				i	∅	∅	∅	∅	none

Reconstructing the DFA

- Each set in Partition forms a state
- For each line in the table where $q_1 \neq \emptyset$, *add an edge from* q_1 to s labelled c

DFA Minimization Algorithm (Worklist version)

```
Worklist  ← { D_A , { D – D_A } }
Partition ← { D_A , { D – D_A } }
While (Worklist ≠ ∅) do
    select a set S from Worklist and remove it
    for each α ∈ Σ do
        Image ← { x | δ(x, α) ∈ S }
        for each q ∈ Partition that has a state in Image
            q_1 ← q ∩ Image
            q_2 ← q – q_1
            if q_2 ≠ ∅ then
                remove q from Partition
                Partition ← Partition ∪ q_1 ∪ q_2
                if q ∈ Worklist then
                    remove q from Worklist
                    Worklist ← Worklist ∪ q_1 ∪ q_2
                else if |q_1| ≤ |q_2|
                    then Worklist ← Worklist ∪ q_1
                    else Worklist ← Worklist ∪ q_2
                if s = q then
                    break;   // cannot keep working on s
```

Why does this algorithm halt?

- Fixed-point algorithm
- DFA has finite number of states
- Start with 2 sets in Partition
- Splitting breaks 1 set into 2 smaller ones but never makes a set larger
  → *Monotone behavior*
- Simple, finite limit on |*Partition* |; it cannot be > |*States* |
- Finite # steps, monotone increasing construction ⇒ algorithm halts

DFA Minimization Algorithm (Worklist version)

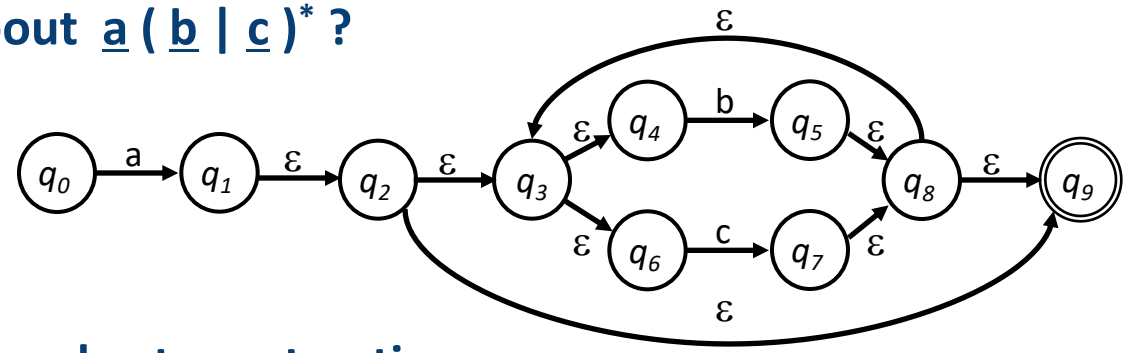
```
Worklist  ← { D_A , { D – D_A } }
Partition ← { D_A , { D – D_A } }

While (Worklist ≠ ∅) do
    select a set S from Worklist and remove it
    for each α ∈ Σ do
        Image ← { x | δ(x, α) ∈ S }
        for each q ∈ Partition that has a state in Image do
            q1 ← q ∩ Image
            q2 ← q – q1
            if q2 ≠ ∅ then
                remove q from Partition
                Partition ← Partition ∪ q1 ∪ q2
                if q ∈ Worklist then
                    remove q from Worklist
                    Worklist ← Worklist ∪ q1 ∪ q2
                else if |q1| ≤ |q2|
                    then Worklist ← Worklist ∪ q1
                    else Worklist ← Worklist ∪ q2
                if s = q then
                    break;   // cannot keep working on s
```

One last hack …

To make an implementation faster, it should maintain an efficient way to determine, for a given state, which set currently contain that state.

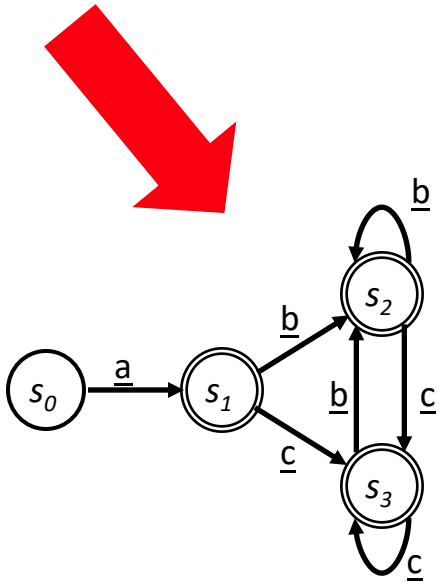
DFA Minimization

What about a (b | c)* ?



From the subset construction:

States		ε-closure(Move(s,*))		
DFA	**NFA**	a	b	c
s_0	q_0	s_1	*none*	*none*
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	*none*	s_2	s_3
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$	*none*	s_2	s_3
s_3	$q_7, q_8, q_9, q_3, q_4, q_6$	*none*	s_2	s_3

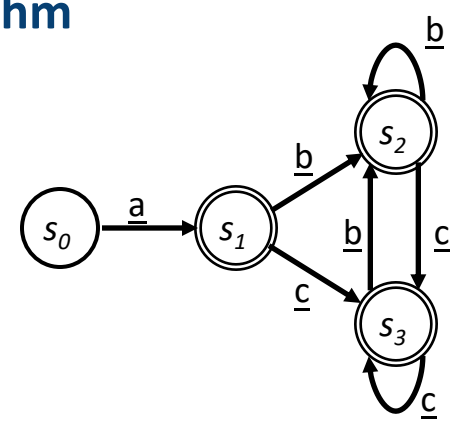


From last lecture …

DFA Minimization

Applying Hopcroft's DFA minimization algorithm

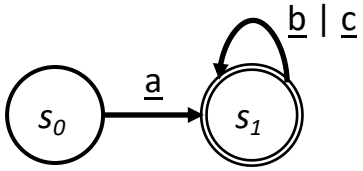
	Current Partition	*Split on* a	b	c
P_0	$\{s_1,s_2,s_3\}$ $\{s_0\}$	*none*	*none*	*none*



It splits no states after the initial partition

⟹ The minimal **DFA** has two states

- One for $\{s_0\}$
- One for $\{s_1, s_2, s_3\}$



Earlier, I suggested that a human would design a simpler automaton than Thompson's construction & the subset construction did.

Minimizing that DFA produces exactly the DFA that I claimed a human would design!

Abbreviated Register Specification

Start with a regular expression

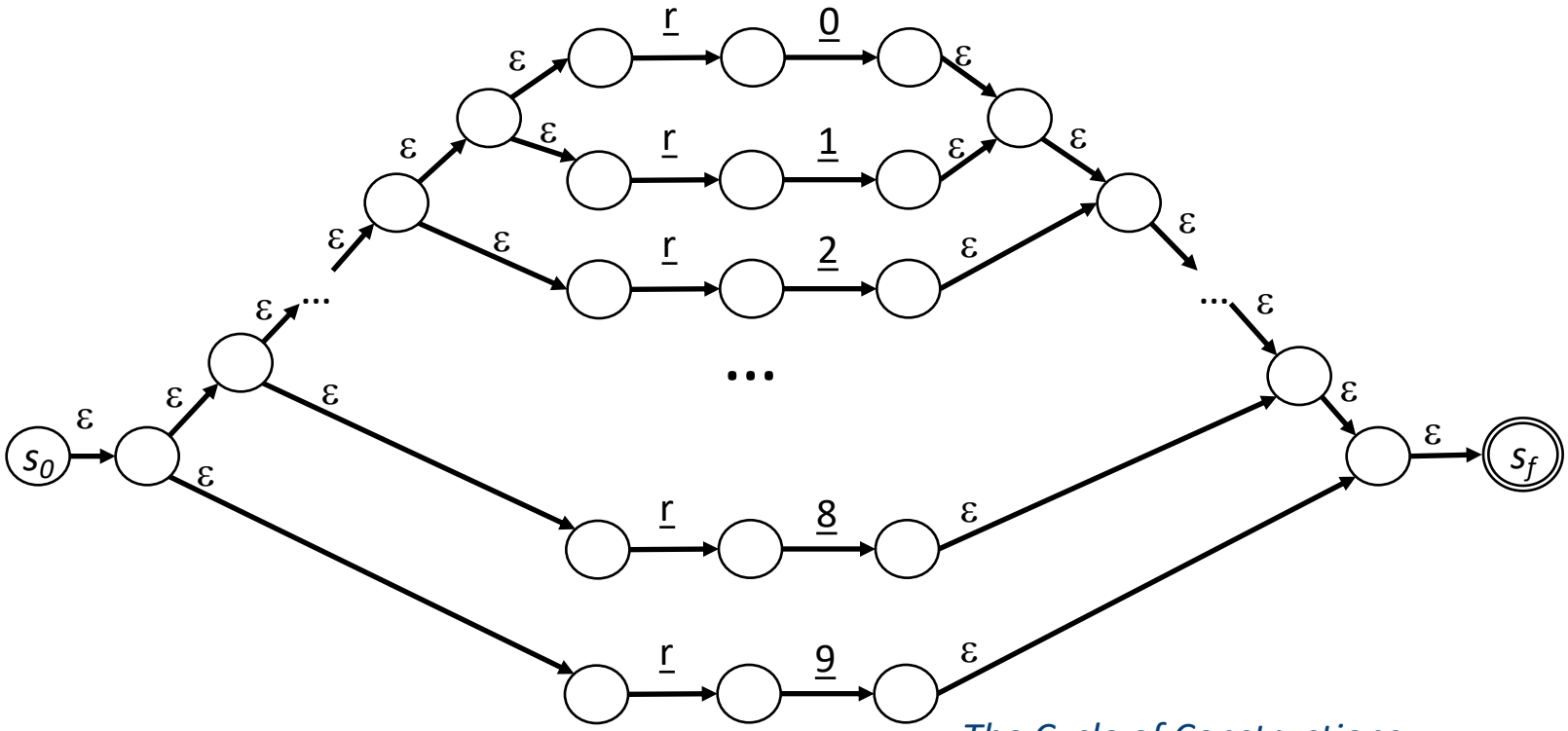
r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7 | r8 | r9

Register names from zero to nine

The Cycle of Constructions

Abbreviated Register Specification

Thompson's construction produces

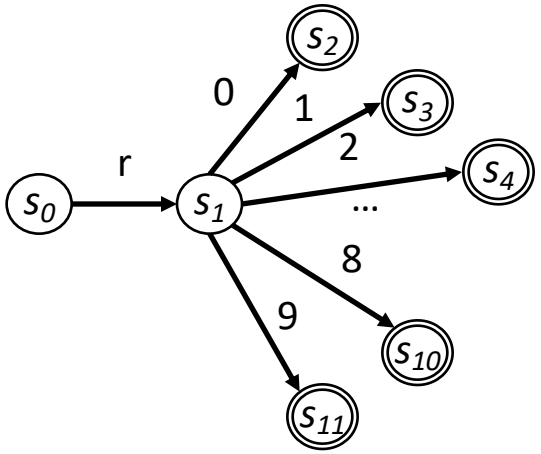


The Cycle of Constructions

To make the example fit, we have eliminated some of the ε-transitions, e.g., between r and 0

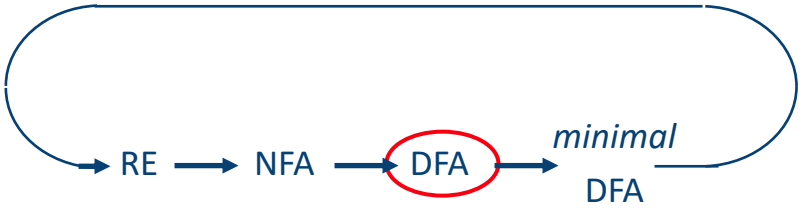
Abbreviated Register Specification

Applying the subset construction yields



This is a **DFA**, but it has a lot of states …

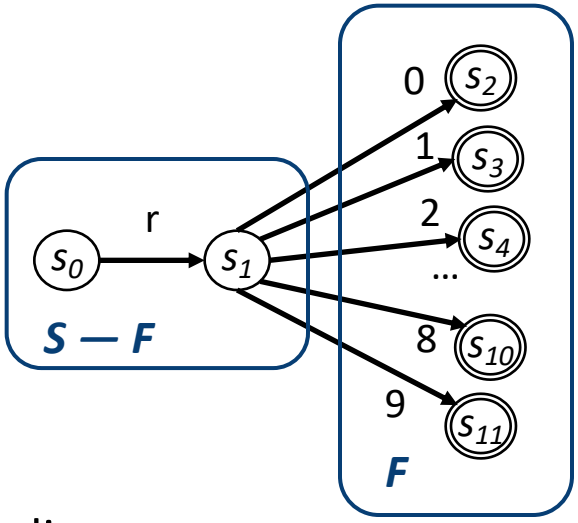
The Cycle of Constructions



Abbreviated Register Specification

Hopcroft's algorithm

Initial sets



F does not split.

Since no transitions leave it, there are no states to split it.

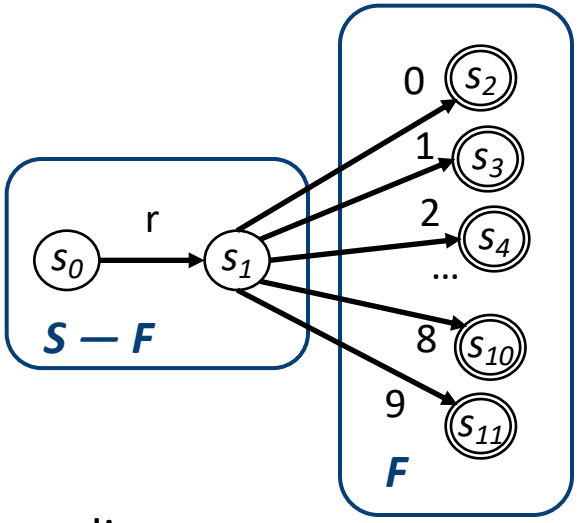
The Cycle of Constructions

Technically, this edge shows up as 10 transitions, which are combined by construction of the character classifier …

Abbreviated Register Specification

Hopcroft's algorithm

Initial sets



{ ***S*** — ***F*** } does split

Any character in Σ will split it into { s_0 }, { s_1 }

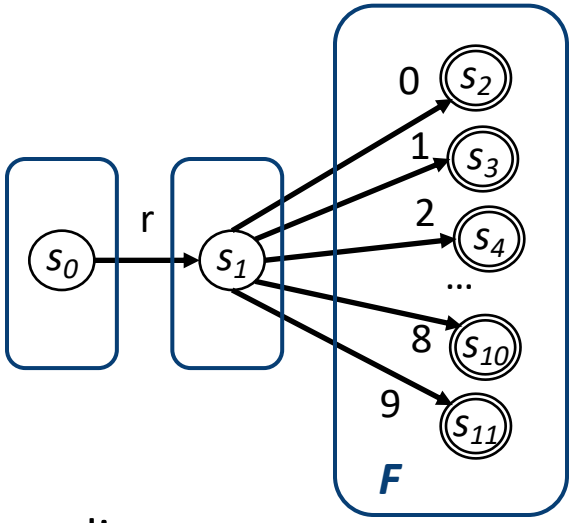
The Cycle of Constructions

Technically, this edge shows up as 10 transitions, which are combined by construction of the character classifier …

Abbreviated Register Specification

Hopcroft's algorithm

Initial sets

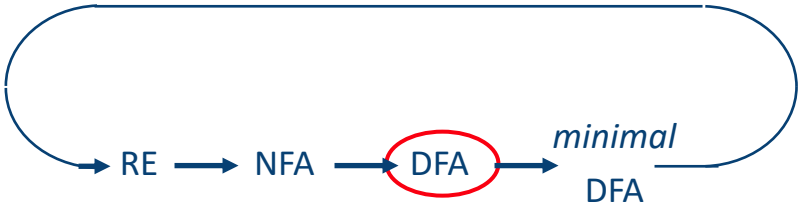


{ ***S*** — ***F*** } does split

Any character in Σ will split it into { s_0 }, { s_1 }

This partition is the final partition

The Cycle of Constructions

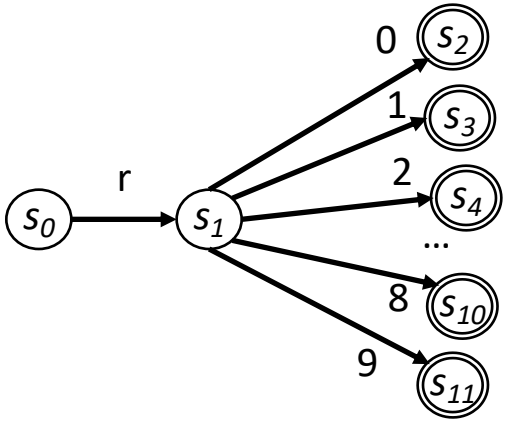


Technically, this edge shows up as 10 transitions, which are combined by construction of the character classifier …

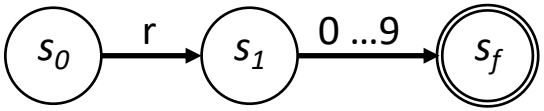
Abbreviated Register Specification

Hopcroft's algorithm

Initial sets



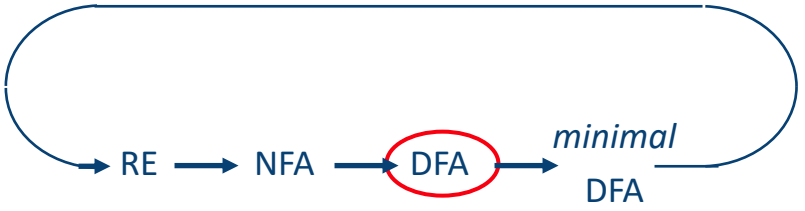
Becomes, through minimization



The Critical Takeaway Points:

- The construction will build a minimal **DFA**
- The size of the **DFA** relates to the language described by the **RE**, not the size of the **RE**
- The result is a **DFA**, so it has **O**(1) cost per character
- The compiler writer can use the most *"natural"* or "*intuitive*" **RE**

The Cycle of Constructions



The Plan for Scanner Construction

RE → NFA *(Thompson's construction)*

- Build an **NFA** for each term in the **RE**
- Combine them in patterns that model the operators

NFA → DFA *(Subset construction)*

- Build a **DFA** that simulates the **NFA**

DFA → Minimal DFA

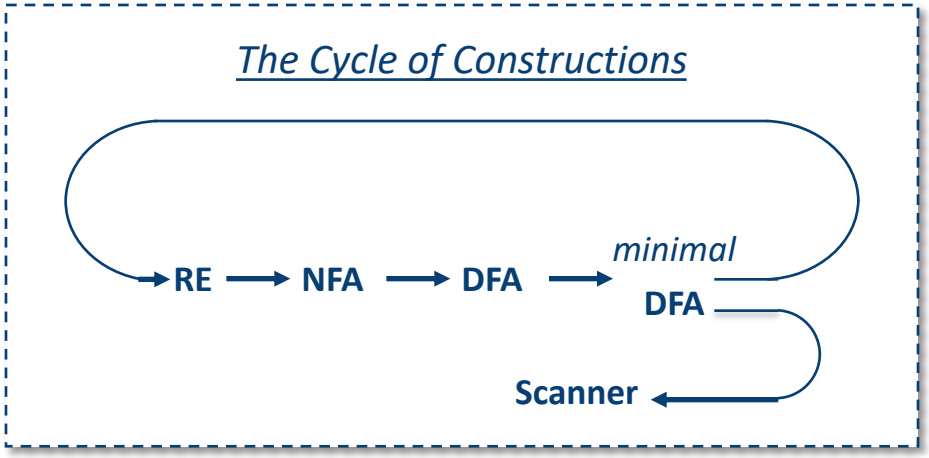
- Hopcroft's algorithm
- Brzozowski's algorithm

Minimal **DFA** → Scanner

- See § 2.5 in EaC2e

DFA → RE

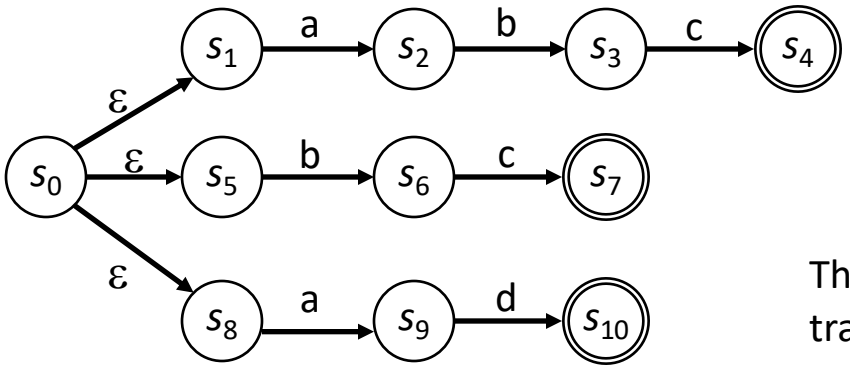
- All pairs, all paths problem
- Union together paths from s_0 to a final state



Brzozowski's Algorithm for **DFA** Minimization

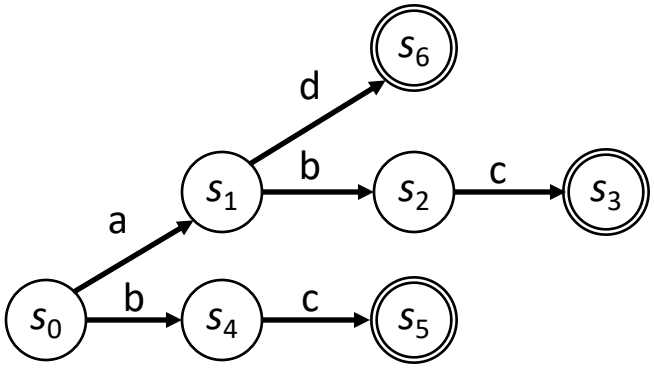
The Intuition

- The subset construction merges prefixes in the **NFA**



abc | bc | ad

Thompson's construction would leave ε-transitions between each single-character automaton



Subset construction eliminates ε-transitions and merges the paths for a. It leaves duplicate tails, such as bc, intact.

Brzozowski's Algorithm

Idea: Use The Subset Construction Twice

- For an **NFA** *N*
  - Let *reverse(N)* be the **NFA** constructed by making initial state final, adding a new start state with an ε-transition to each previously final state, and reversing the other edges
  - Let *subset(N)* be the **DFA** produced by the subset construction on *N*
  - Let *reachable(N)* be *N* after removing any states that are not reachable from the initial state
- Then,

$$reachable(subset(reverse(\ reachable(subset(reverse(N))\ )))$$

is a minimal **DFA** that implements *N* [Brzozowski, 1962]

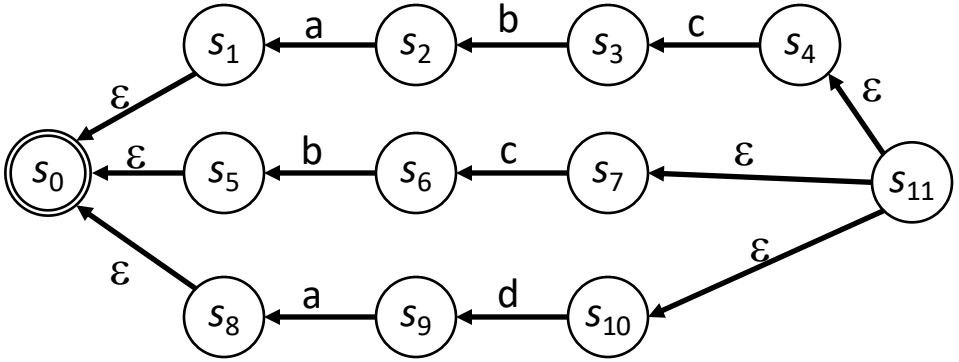
Not everyone finds this result to be intuitive.

Neither algorithm dominates the other.

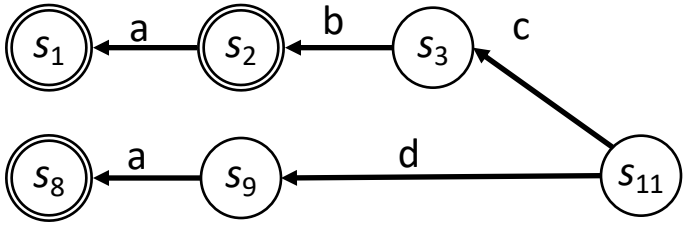
Brzozowski's Algorithm

Step 1

- The subset construction on *reverse(NFA)* merges suffixes in original NFA



Reversed NFA

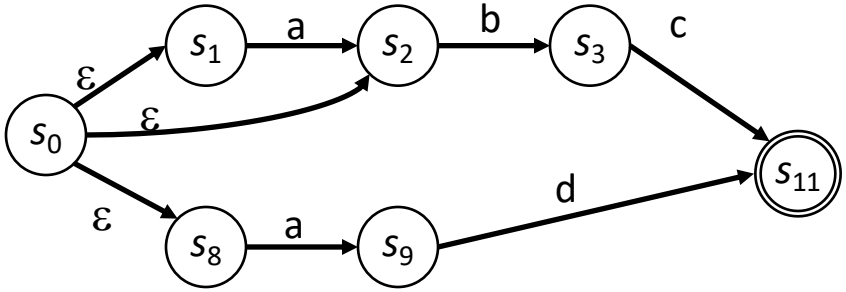


subset(reverse(NFA))

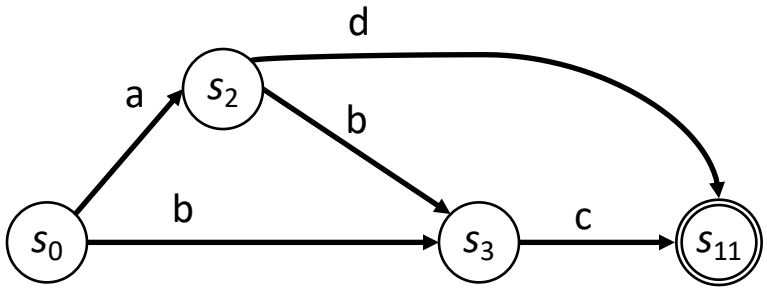
Brzozowski's Algorithm

Step 2

- Reverse it again & use subset to merge prefixes …



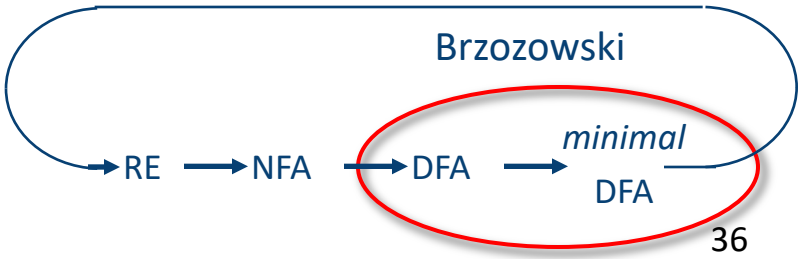
Reverse it, again



Minimal DFA

And subset it, again

The Cycle of Constructions

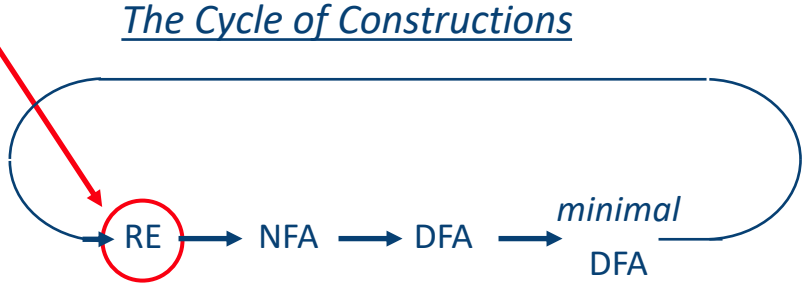


Abbreviated Register Specification

Start with a regular expression

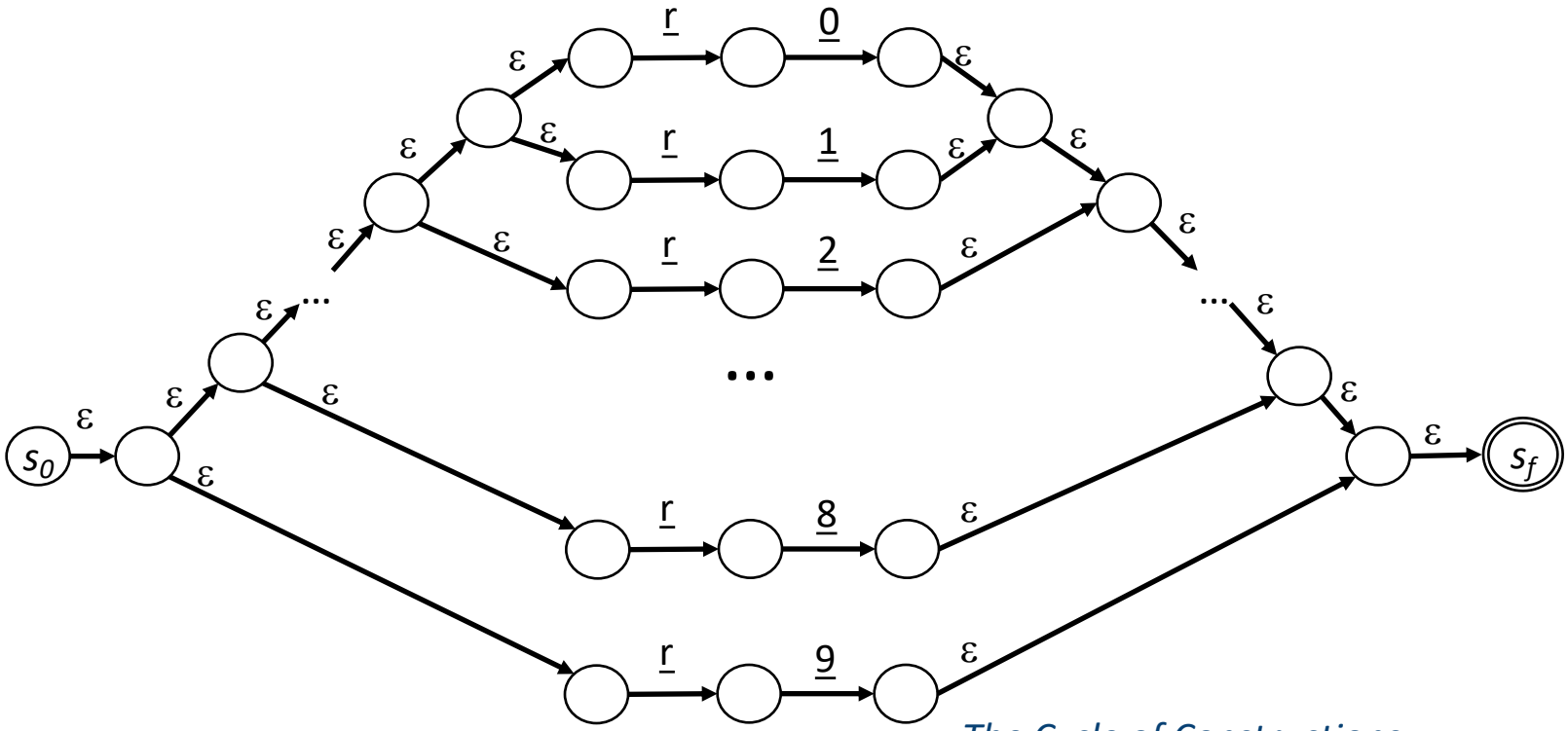
r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7 | r8 | r9

Register names from zero to nine



Abbreviated Register Specification

Thompson's construction produces something along these lines

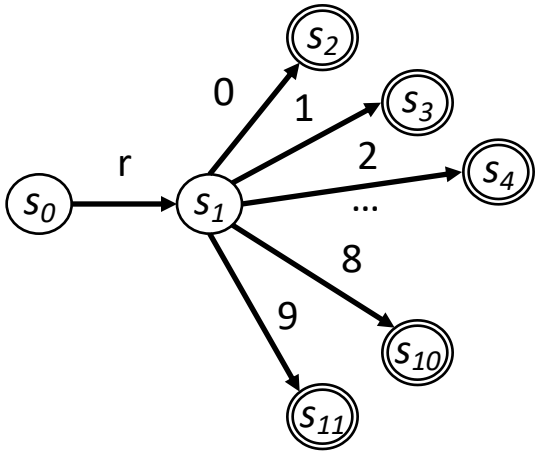


The Cycle of Constructions

To make the example fit, we have eliminated some of the ε-transitions, e.g., between r and 0

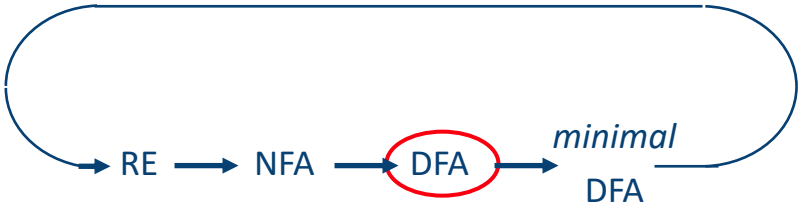
Abbreviated Register Specification

Applying the subset construction yields



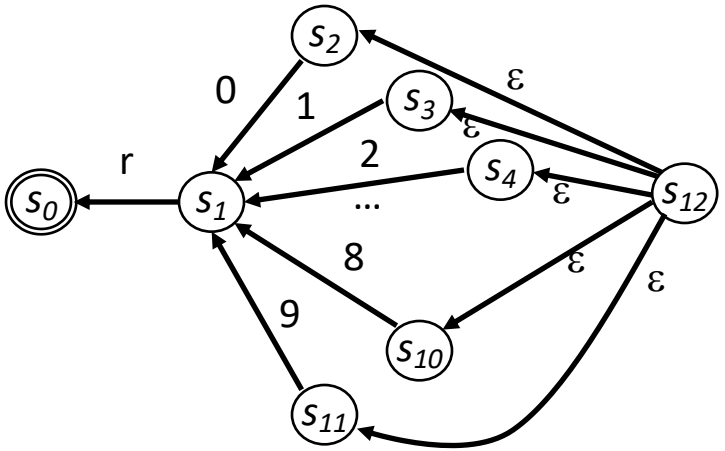
This is a **DFA**, but it has a lot of states …

The Cycle of Constructions

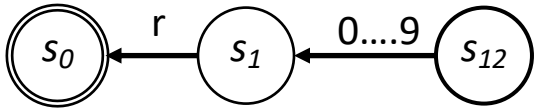


Abbreviated Register Specification

Applying Brzozowski's algorithm, step 1

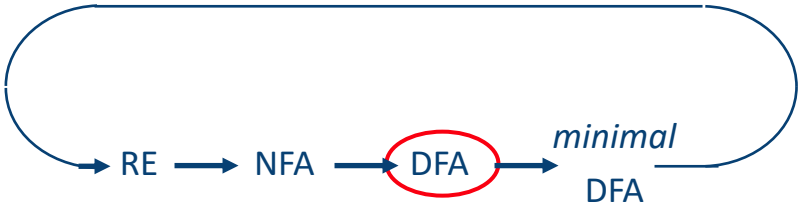


Reversed NFA



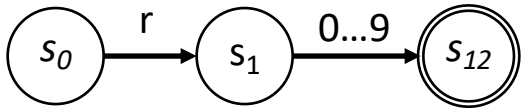
After Subset Construction

The Cycle of Constructions



Abbreviated Register Specification

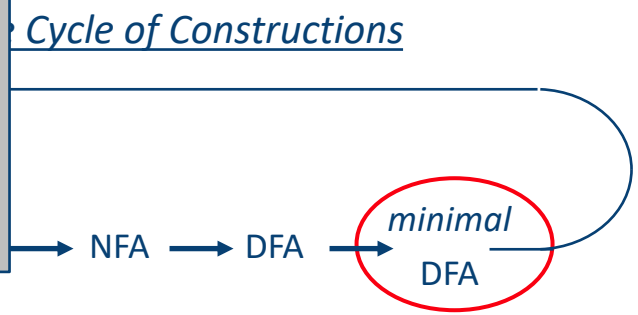
Brzozowski, step 2 reverses that DFA and subsets it again



A skilled human might build this **DFA**

The Critical Point:

- The construction will build a minimal **DFA**
- The size of the **DFA** relates to the language described by the **RE**, not the size of the **RE**
- The result is a **DFA**, so it has **O**(1) cost per character
- The compiler writer can use the "most natural" or "intuitive" **RE**



One Last Algorithm

RE Back to DFA

The Wikipedia page on "Kleene's algorithm" is pretty good. It also contains a link to Kleene's 1956 paper. This form of the algorithm is usually attributed to McNaughton and Yamada in 1960.

Kleene's Construction

```
for i ← 0 to |D| - 1;    // label each immediate path
   for j ← 0 to |D| - 1;
      R⁰ij ← { a | δ(di,a) = dj};
      if (i = j) then
         R⁰ii = R⁰ii | {ε};

for k ← 0 to |D| - 1;    // label nontrivial paths
   for i ← 0 to |D| - 1;
      for j ← 0 to |D| - 1;
         Rᵏij ← Rᵏ⁻¹ik (Rᵏ⁻¹kk)* Rᵏ⁻¹kj | Rᵏ⁻¹ij

L ← {}                   // union labels of paths from
For each final state si  //  s0 to a final state si
   L ← L | R|D|-1 0i
```

R^k_{ij} is the set of paths from i to j that include no state higher than k

Adaptation of all points, all paths, low cost algorithm

The Cycle of Constructions