Lexical Analysis: DFA Minimization & Wrap Up

Automating Scanner Construction

RE→NFA (*Thompson's construction*) ☐

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

The Cycle of Constructions

NFA →DFA (*subset construction*) ☐

- Build the simulation

DFA →Minimal DFA *(today)*

- Hopcroft's algorithm

DFA →RE *(not really part of scanner construction)*

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

DFA Minimization

The Big Picture

- Discover sets of equivalent states
- Represent each such set with just one state

DFA Minimization

The Big Picture

- Discover sets of equivalent states
- Represent each such set with just one state

Two states are equivalent if and only if:

- The set of paths leading to them are equivalent
- $\forall\ \alpha \in \Sigma$, transitions on α lead to equivalent states (DFA)
- α-transitions to distinct sets $\Rightarrow$ states must be in distinct sets

DFA Minimization

The Big Picture

- Discover sets of equivalent states
- Represent each such set with just one state

Two states are equivalent if and only if:

- The set of paths leading to them are equivalent
- $\forall\ \alpha \in \Sigma$, transitions on α lead to equivalent states (DFA)
- α-transitions to distinct sets $\Rightarrow$ states must be in distinct sets

A partition P of S

- Each $s \in S$ is in exactly one set $p_i \in P$
- The algorithm iteratively partitions the DFA's states

DFA Minimization

Details of the algorithm

- Group states into maximal size sets, *optimistically*
- Iteratively subdivide those sets, as needed
- States that remain grouped together are equivalent

Initial partition, P_0 , has two sets: {F} & {Q-F}   ($D =(Q,\Sigma,\delta,q_0,F)$)

Splitting a set ("partitioning a set by <u>a</u>")

- Assume q_a, & $q_b \in s$, and $\delta(q_a,\underline{a}) = q_x$, & $\delta(q_b,\underline{a}) = q_y$
- If q_x & q_y are not in the same set, then *s* must be split
  - → q_a has transition on a, q_b does not ⇒ <u>a</u> splits *s*
- One state in the final DFA cannot have two transitions on <u>a</u>

DFA Minimization

The algorithm

```
P ← { F, {Q-F}}
while ( P is still changing)
    T ← { }
    for each set S ∈ P
        for each α ∈ Σ
            partition S by α
                into S_1, and S_2
            T ← T ∪ S_1 ∪ S_2
    if T ≠ P then
        P ← T
```

This is a fixed-point algorithm!

Why does this work?

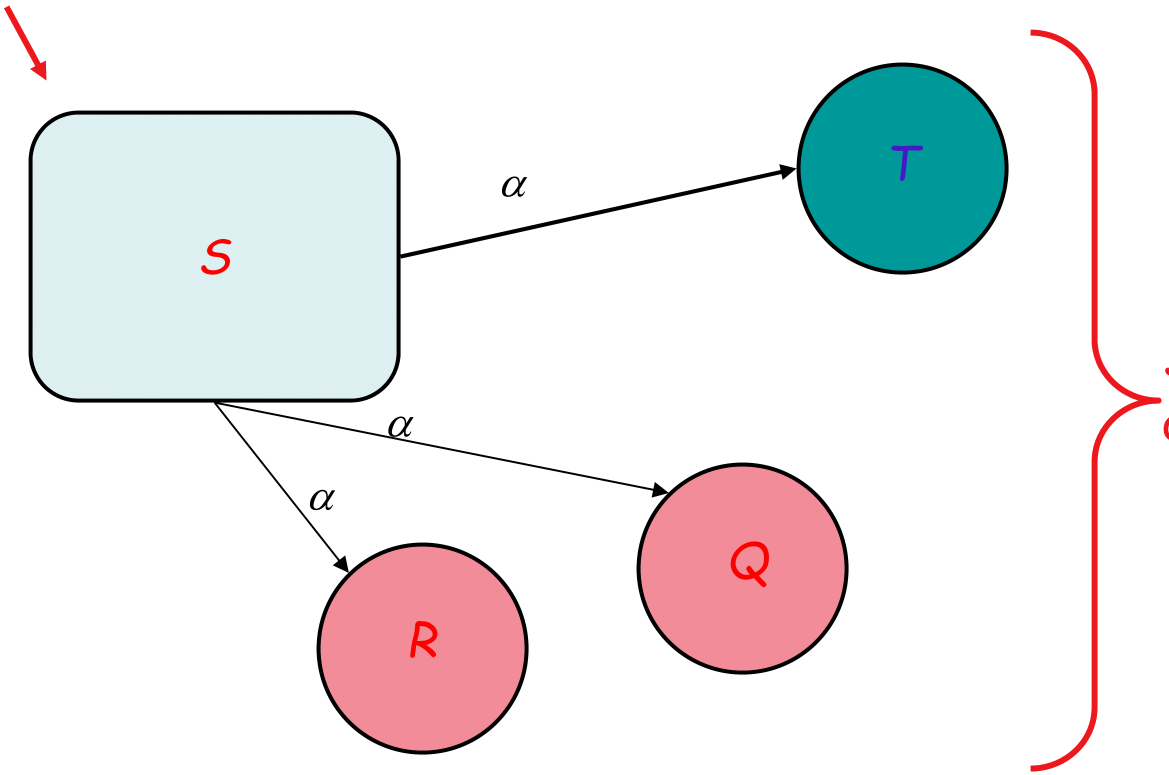
- Partition $P \in 2^Q$
- Start off with 2 subsets of Q {F} and {Q-F}
- While loop takes $P_i \to P_{i+1}$ by splitting 1 or more sets
- P_{i+1} is at least one step closer to the partition with |Q| sets
- Maximum of |Q | splits

Note that

- Partitions are <u>never</u> combined
- Initial partition ensures that final states are intact

Key Idea: Splitting S around α

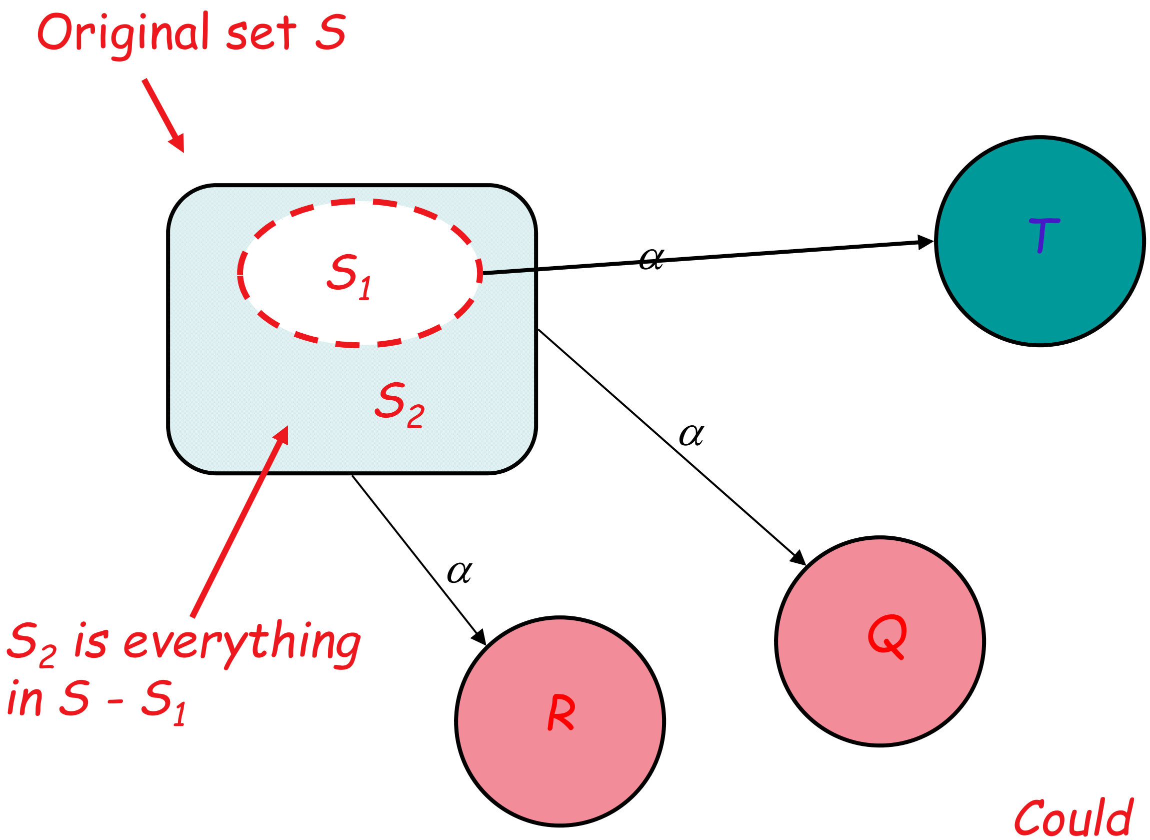
Original set S



S has transitions on α to R, Q, & T

The algorithm partitions S around α

Key Idea: Splitting S around α



Original set S

S_2 is everything in $S - S_1$

Could we split S_2 further?

Yes, but it does not help asymptotically

DFA Minimization

Refining the algorithm

- As written, it examines every $S \in P$ on each iteration
  - → This does a lot of unnecessary work
  - → Only need to examine S if some T, reachable from S, has split
- Reformulate the algorithm using a "worklist"
  - → Start worklist with initial partition, F and Q-F
  - → When it splits S into S_1 and S_2, place S_2 on worklist

This version looks at each $S \in P$ many fewer times

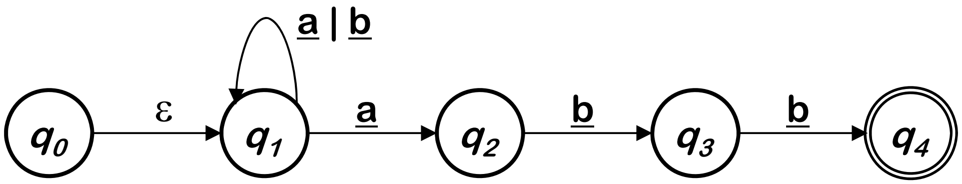
⇒ Well-known, widely used algorithm due to John Hopcroft

Hopcroft's Algorithm

```
$W \leftarrow \{F, Q-F\}$;  $P \leftarrow \{F, Q-F\}$;  // W is the worklist, P the current partition
while ( $W$ is not empty ) do begin
    select and remove $S$ from $W$;  // $S$ is a set of states
    for each $\alpha$ in $\Sigma$ do begin
        let $I_\alpha \leftarrow \delta_\alpha^{-1}(S)$;  // $I_\alpha$ is set of all states that can reach S on $\alpha$
        for each $R$ in $P$ such that $R \cap I_\alpha$ is not empty
            and $R$ is not contained in $I_\alpha$ do begin
            partition $R$ into $R_1$ and $R_2$ such that $R_1 \leftarrow R \cap I_\alpha$; $R_2 \leftarrow R - R_1$;
            replace $R$ in P with $R_1$ and $R_2$;
            if $R \in W$ then replace $R$ with $R_1$ in $W$ and add $R_2$ to $W$;
            else if $|| R_1 || \leq || R_2 ||$
                then add add $R_1$ to $W$;
                else add $R_2$ to $W$;
        end
    end
end
```

A Detailed Example

Remember (a | b)* abb ? *(from last lecture)*



Our first NFA

Applying the subset construction:

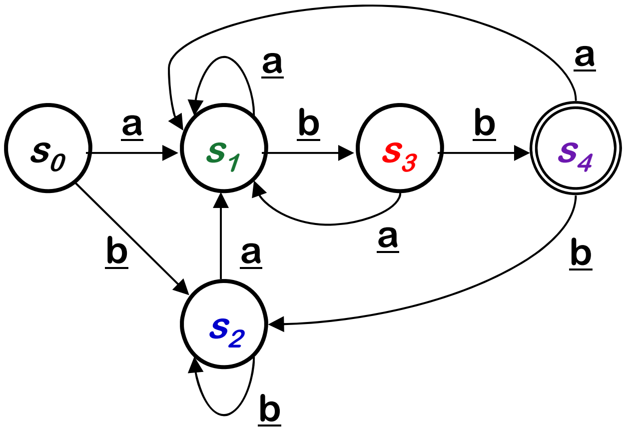
Iter.	State	Contains	ε-closure(move(s_i,a))	ε-closure(move(s_i,b))
0	s_0	q_0, q_1	q_1, q_2	q_1
1	s_1	q_1, q_2	q_1, q_2	q_1, q_3
	s_2	q_1	q_1, q_2	q_1
2	s_3	q_1, q_3	q_1, q_2	q_1, q_4
3	s_4	q_1, q_4	q_1, q_2	q_1

Iteration 3 adds nothing to *S*, so the algorithm halts

contains q_4 (final state)

A Detailed Example

The DFA for (a | b)* abb



δ	a	b
s_0	s_1	s_2
s_1	s_1	s_3
s_2	s_1	s_2
s_3	s_1	s_4
s_4	s_1	s_2

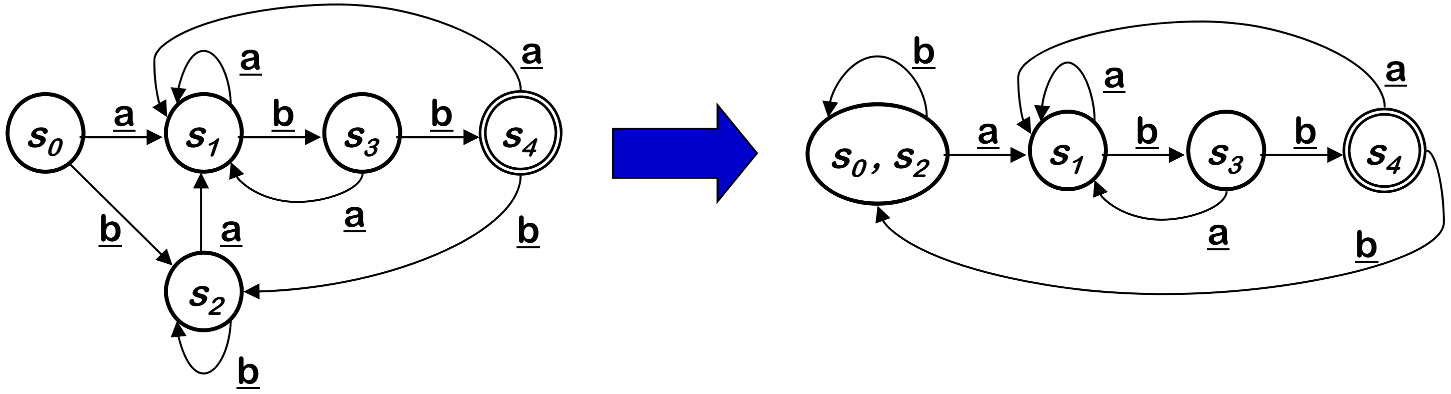
- Not much bigger than the original
- All transitions are deterministic
- Use same code skeleton as before

A Detailed Example

Applying the minimization algorithm to the DFA

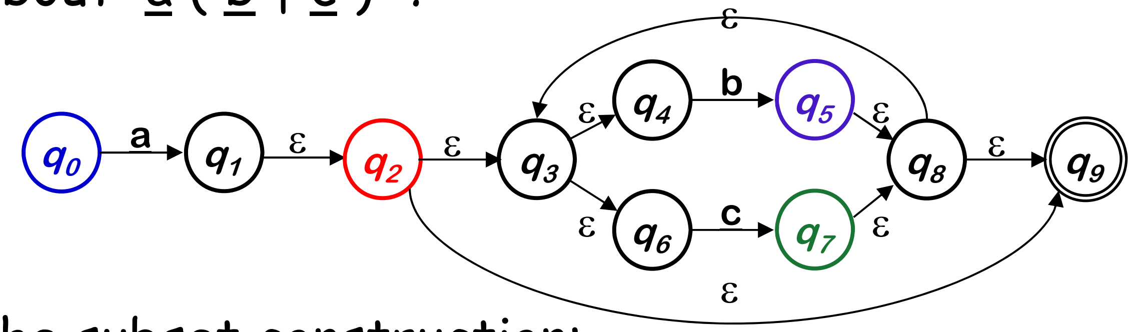
	Current Partition	*Worklist*	*s*	*Split on a*	*Split on b*
P_0	$\{s_4\}$ $\{s_0,s_1,s_2,s_3\}$	$\{s_4\}$ $\{s_0,s_1,s_2,s_3\}$	$\{s_4\}$	none	$\{s_0, s_1, s_2\}$ $\{s_3\}$
P_1	$\{s_4\}$ $\{s_3\}$ $\{s_0,s_1,s_2\}$	$\{s_0,s_1,s_2\}$ $\{s_3\}$	$\{s_3\}$	none	$\{s_0, s_2\}\{s_1\}$
P_2	$\{s_4\}$ $\{s_3\}$ $\{s_1\}$ $\{s_0,s_2\}$	$\{s_0,s_2\}\{s_1\}$	$\{s_1\}$	none	none

final state



DFA Minimization

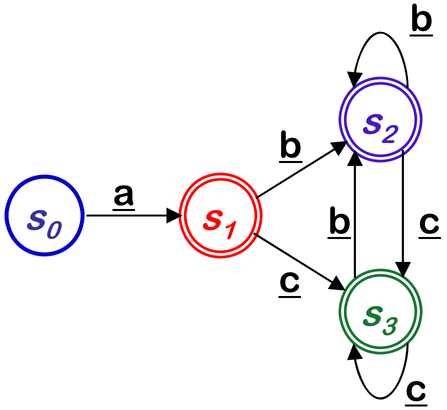
What about $\underline{a}\,(\,\underline{b} \mid \underline{c}\,)^*$?



First, the subset construction:

		ε-closure(move(s,*))		
	NFA states	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	*none*	*none*
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	*none*	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
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

Final states

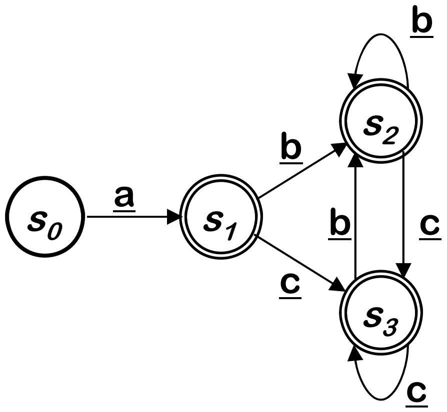


DFA Minimization

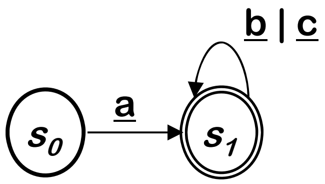
Then, apply the minimization algorithm

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

final states



To produce the minimal DFA



In lecture 5, we observed that a human would design a simpler automaton than Thompson's construction & the subset construction did.

Minimizing that DFA produces the one that a human would design!

Abbreviated Register Specification

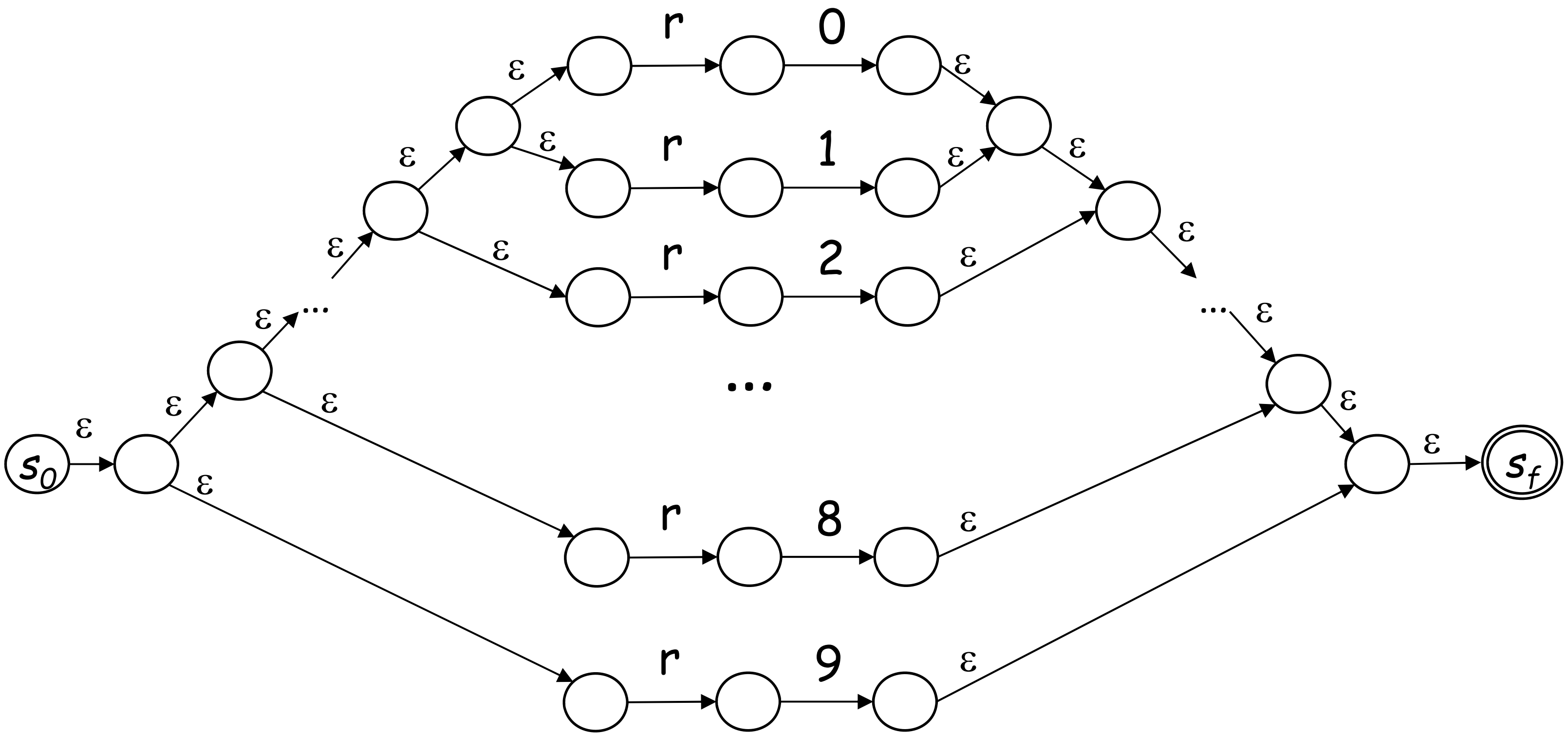
Start with a regular expression

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

The Cycle of Constructions

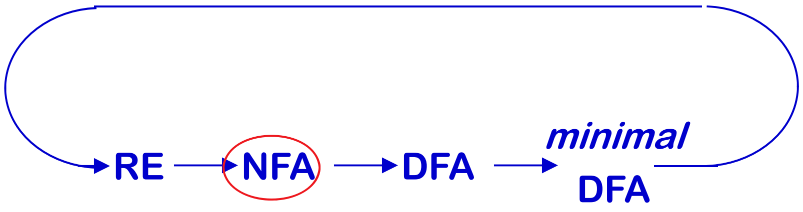
Abbreviated Register Specification

Thompson's construction produces



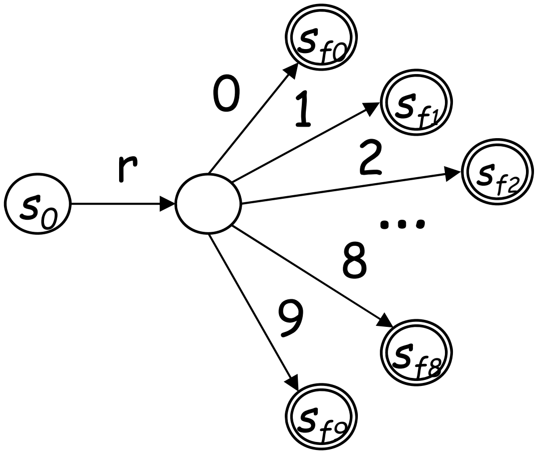
To make it fit, we've eliminated the ε-transition between "r" and "0".

The Cycle of Constructions


RE → NFA → DFA → *minimal* DFA

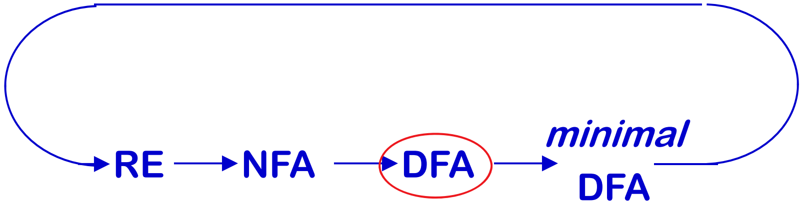
Abbreviated Register Specification

The subset construction builds



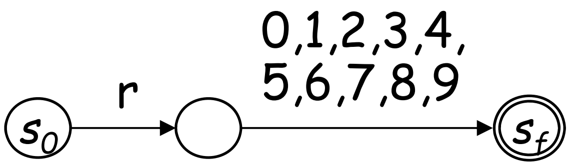
This is a DFA, but it has a lot of states …

The Cycle of Constructions

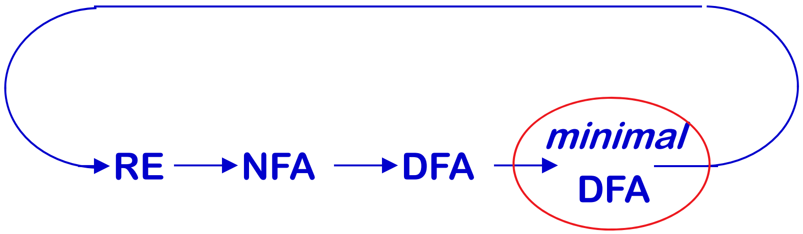


Abbreviated Register Specification

The DFA minimization algorithm builds



This looks like what a skilled compiler writer would do!



The Cycle of Constructions

Limits of Regular Languages

Advantages of Regular Expressions

- Simple & powerful notation for specifying patterns
- Automatic construction of fast recognizers
- Many kinds of syntax can be specified with REs

Example — an expression grammar

$$\begin{array}{lll} \mathit{Term} & \rightarrow & [\text{a-zA-Z}]\ ([\text{a-zA-z}] \mid [\underline{0}\text{-}\underline{9}])^* \\ \mathit{Op} & \rightarrow & \underline{+} \mid \underline{-} \mid \underline{*} \mid \underline{/} \\ \mathit{Expr} & \rightarrow & (\ \mathit{Term}\ \mathit{Op}\ )^*\ \mathit{Term} \end{array}$$

Of course, this would generate a DFA …

If REs are so useful …

Why not use them for everything?

Limits of Regular Languages

Not all languages are regular

$\text{RL's} \subset \text{CFL's} \subset \text{CSL's}$

You cannot construct DFA's to recognize these languages

- $L = \{ p^k q^k \}$ *(parenthesis languages)*
- $L = \{ wcw^r \mid w \in \Sigma^* \}$

Neither of these is a regular language *(nor an RE)*

But, this is a little subtle. You can construct DFA's for

- Strings with alternating 0's and 1's
  $(\varepsilon \mid 1) (01)^* (\varepsilon \mid 0)$
- Strings with an even number of 0's and 1's

RE's can count bounded sets and bounded differences

What can be so hard?

Poor language design can complicate scanning

- Reserved words are important

  if then then then = else; else else = then (PL/I)

- Insignificant blanks (Fortran & Algol68)

  do 10 i = 1,25

  do 10 i = 1.25

- String constants with special characters (C, C++, Java, …)

  newline, tab, quote, comment delimiters, …

- Finite closures (Fortran 66 & Basic)
  - → Limited identifier length
  - → Adds states to count length

What can be so hard? (Fortran 66/77)

```
        INTEGERFUNCTIONA
        PARAMETER(A=6,B=2)
        IMPLICIT CHARACTER*(A-B)(A-B)
        INTEGER FORMAT(10), IF(10), DO9E1
 100    FORMAT(4H)=(3)
 200    FORMAT(4   )=(3)
        DO9E1=1
        DO9E1=1,2
   9    IF(X)=1
        IF(X)H=1
        IF(X)300,200
 300    CONTINUE
        END
C       THIS IS A “COMMENT CARD”
     $  FILE(1)
        END
```

How does a compiler scan this?

- First pass finds & inserts blanks
- Can add extra words or tags to create a scannable language
- Second pass is normal scanner

Example due to Dr. F.K. Zadeck

Building Faster Scanners from the DFA

Table-driven recognizers waste effort

- Read (& classify) the next character
- Find the next state
- Assign to the state variable
- Trip through case logic in *action()*
- Branch back to the top

We can do better

- Encode state & actions in the code
- Do transition tests locally
- Generate ugly, spaghetti-like code
- Takes (many) fewer operations per input character

```
char ← next character;
state ← s0 ;
call action(state,char);
while (char ≠ eof)
  state ← δ(state,char);
  call action(state,char);
  char ← next character;

if T(state) = final then
  report acceptance;
else
  report failure;
```

Building Faster Scanners from the DFA

A direct-coded recognizer for r *Digit Digit**

```
        goto $s_0$;
   $s_0$: word ← Ø;
        char ← next character;
        if (char = 'r')
           then goto $s_1$;
           else goto $s_e$;
   $s_1$: word ← word + char;
        char ← next character;
        if ('0' ≤ char ≤ '9')
           then goto $s_2$;
           else goto $s_e$;
   s2: word ← word + char;
        char ← next character;
        if ('0' ≤ char ≤ '9')
           then goto $s_2$;
           else if (char = eof)
              then report success;
              else goto $s_e$;
   $s_e$: print error message;
        return failure;
```

- Many fewer operations per character
- Almost no memory operations
- Even faster with careful use of fall-through cases

Building Faster Scanners

Hashing keywords versus encoding them directly

- Some *(well-known)* compilers recognize keywords as identifiers and check them in a hash table
- Encoding keywords in the DFA is a better idea
  - → O(1) cost per transition
  - → Avoids hash lookup on each identifier

It is hard to beat a well-implemented DFA scanner

Building Scanners

The point

- All this technology lets us automate scanner construction
- Implementer writes down the regular expressions
- Scanner generator builds NFA, DFA, minimal DFA, and then writes out the (table-driven or direct-coded) code
- This reliably produces fast, robust scanners

For most modern language features, this works

- You should think twice before introducing a feature that defeats a DFA-based scanner
- The ones we've seen (e.g., insignificant blanks, non-reserved keywords) have not proven particularly useful or long lasting

Extra Slides Start Here

Some Points of Disagreement with EAC

- Table-driven scanners are not fast
  - → EaC doesn't say they are slow; it says you can do better
  - → Scanning is a small part of the work in a compiler, so in most cases it cannot make a large % improvement: decide where to spend your effort!
- Faster code can be generated by embedding scanner in code
  - → This was shown for both LR-style parsers and for scanners in the 1980s; flex and its derivatives are an example
- Hashed lookup of keywords is slow
  - → EaC doesn't say it is slow. It says that the effort can be folded into the scanner so that it has no extra cost. Compilers like GCC use hash lookup. A word must fail in the lookup to be classified as an identifier. With collisions in the table, this can add up. At any rate, the cost is unneeded, since the DFA can do it for O(1) cost per character. But again, what % of total cost to compile is this?

Building Faster Scanners from the DFA

Table-driven recognizers waste a lot of effort

- index Read (& classify) the next character
- index Find the next state
- Assign to the state variable
- index Trip through case logic in *action()*
- Branch back to the top

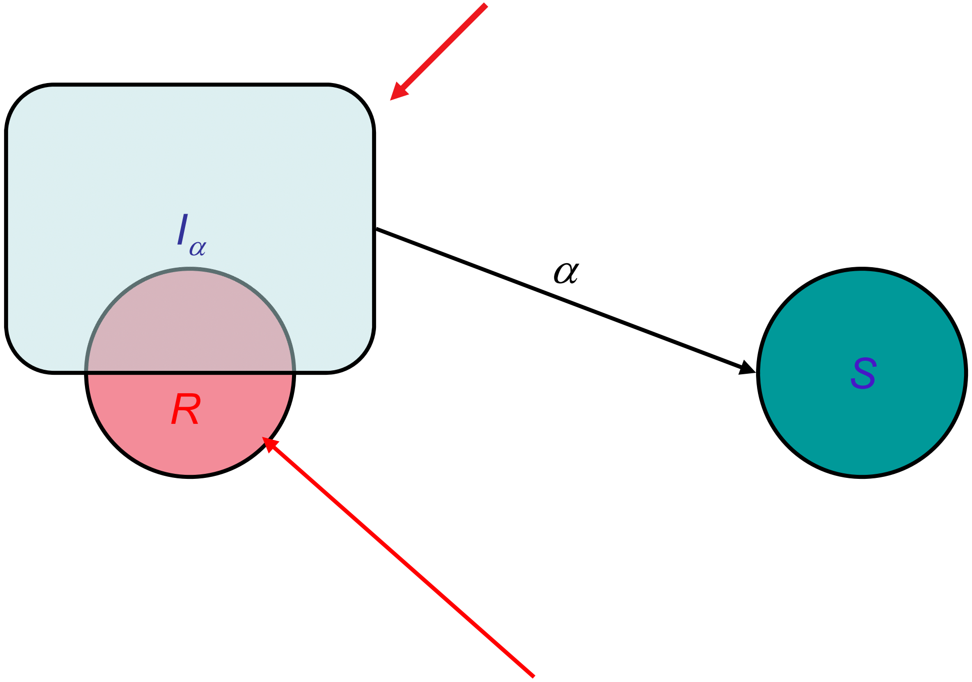
We can do better

- Encode state & actions in the code
- Do transition tests locally
- Generate ugly, spaghetti-like code
- Takes (many) fewer operations per input character

```
char $\leftarrow$ next character;
state $\leftarrow$ $s_0$;
call action(state,char);
while (char $\neq$ eof)
  state $\leftarrow$ $\delta$(state,char);
  call action(state,char);
  char $\leftarrow$ next character;

if $\mathcal{T}$(state) = final then
  report acceptance;
else
  report failure;
```

Key Idea: Splitting S around α



Find partition I that has an α-transition into S

This part must have an a-transition to some other state!