AUTOMATIC LEXICAL ANALYSIS

CMPT 379 Lecture 14a

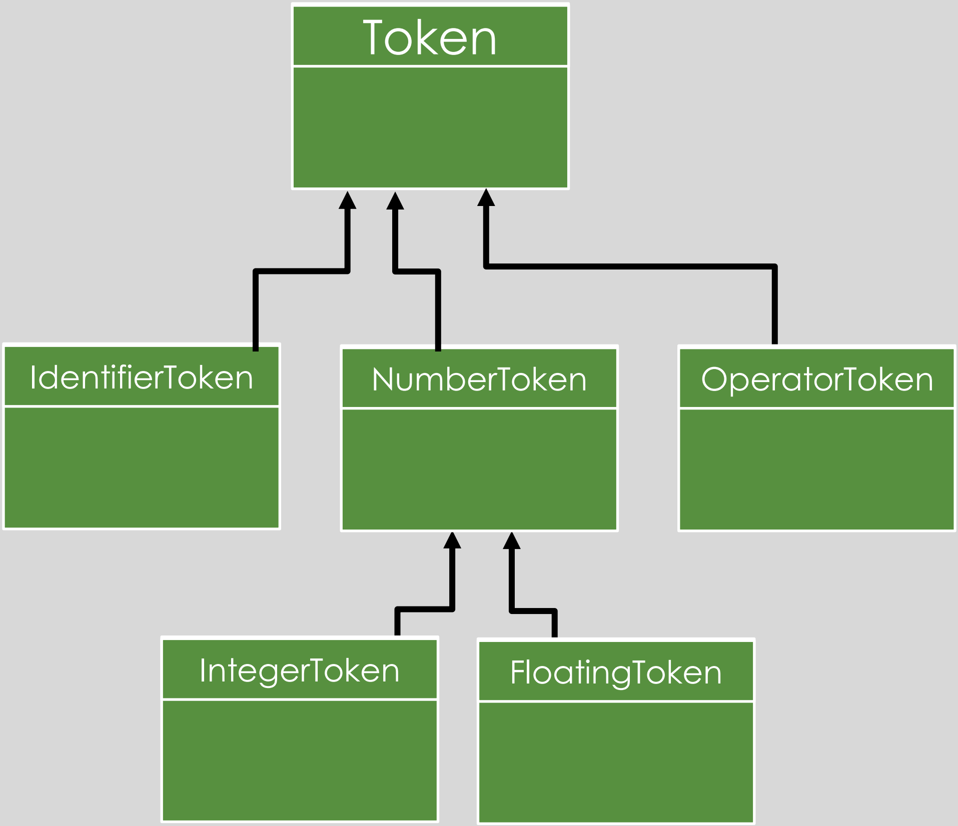
Lecture Overview

- Regular Expressions to NFA to DFA
- Transition Tables
- Table-driven Lexical Analyzers
- Direct-coded Lexical Analyzers
- Hand-coded Lexical Analyzers
- Hopcroft Minimization

Automatically Produced Lexical Analyzers

- There are a number of lexical analyzer generators: lex, flex, and ANTLR for example.
- They accept as input a collection of regular expressions, and some specification of what to do when each regular expression is recognized.
- Some generators allow free-form code to express what to do, while others restrict specification, such as allowing only the token type to be specified.

Token types



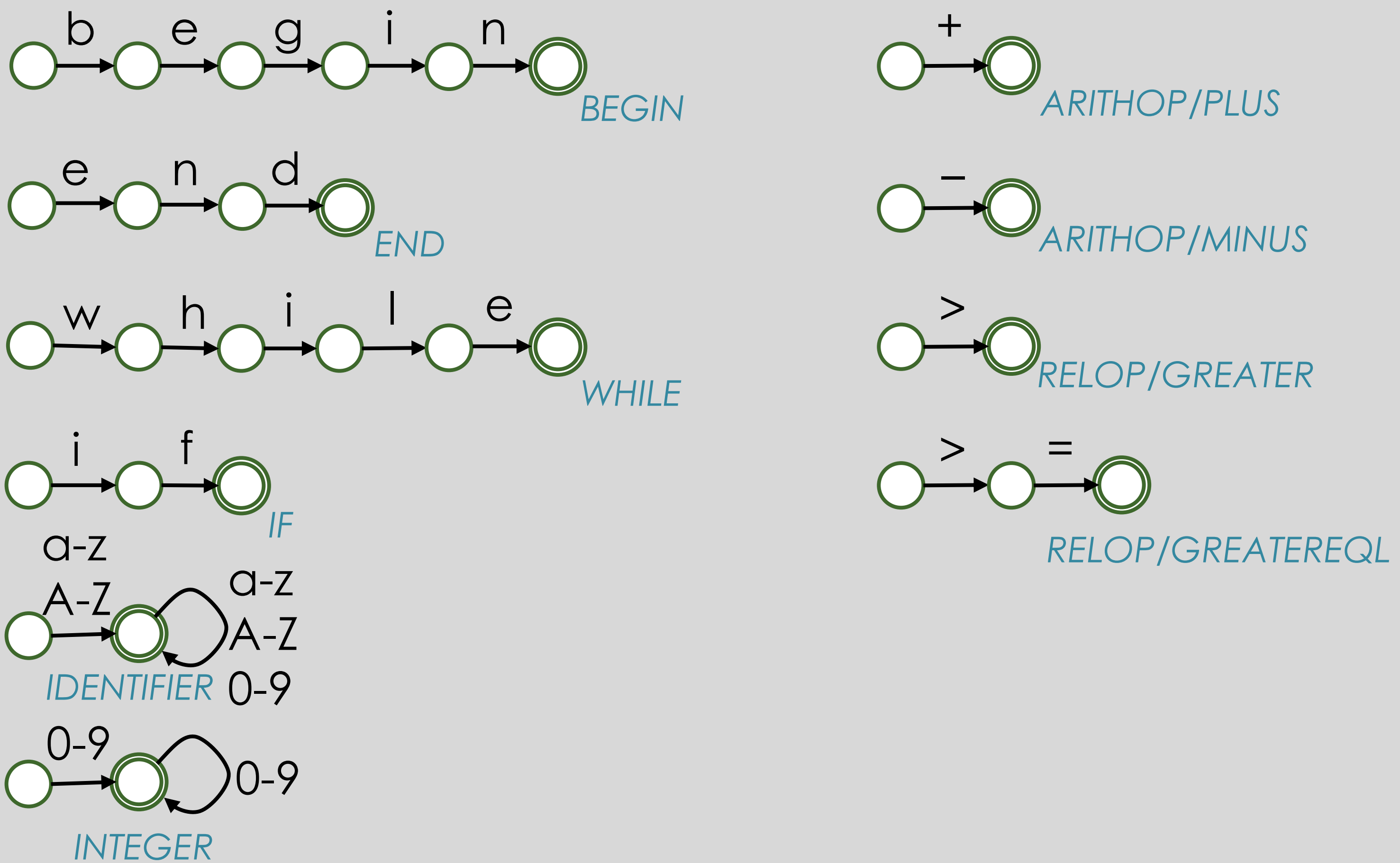
Using types of the compiler language

Using int or enum type field in a generic token

Sample Lex Specification

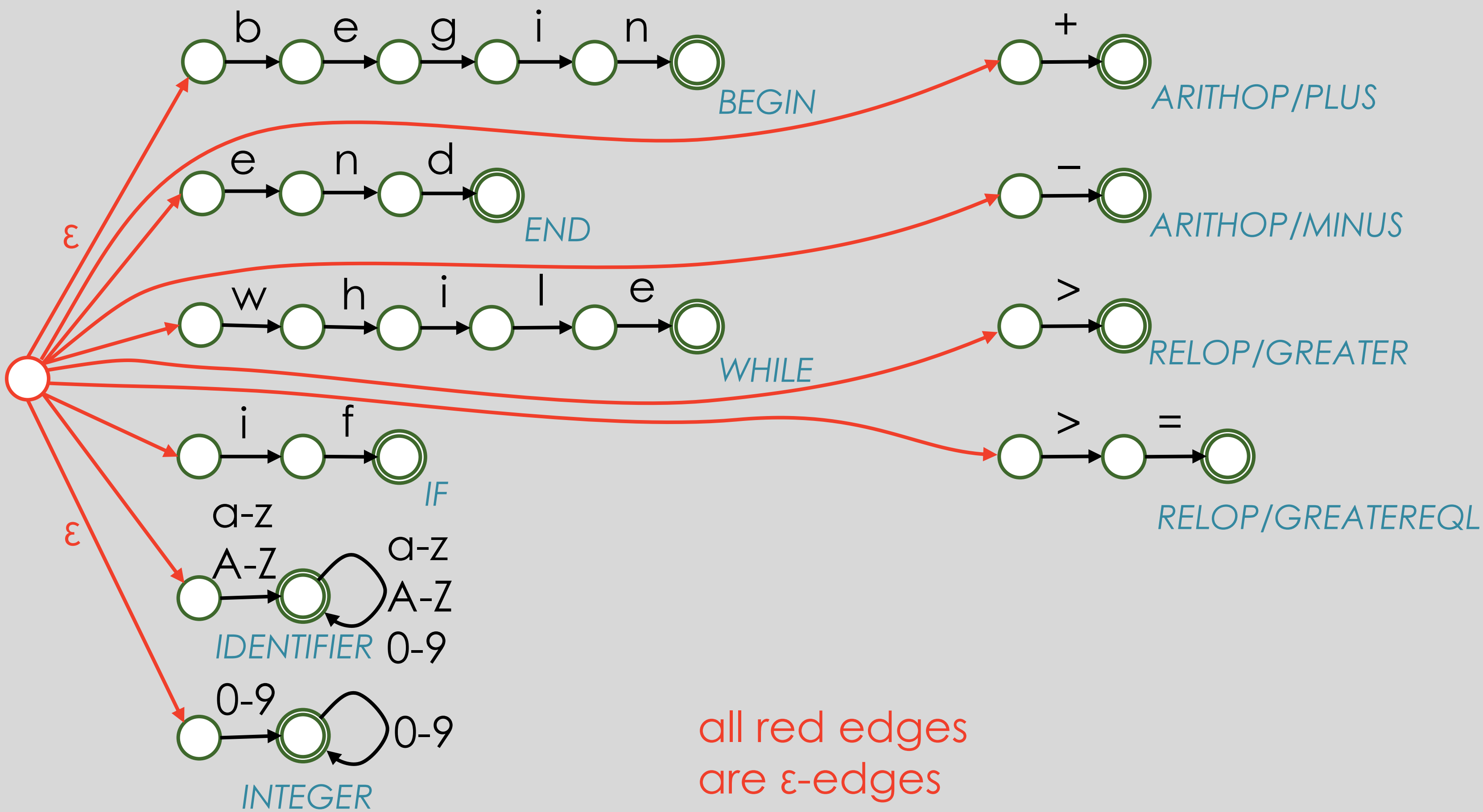
- **begin** **return(BEGIN);**
- **end** **return(END);**
- **while** **return(WHILE);**
- **if** **return(IF);**
- **[a-zA-Z][a-zA-Z0-9]*** **{ tokval = install(); return(IDENTIFIER); }**
- **[0-9]+** **{ tokval = install(); return(INTEGER); }**
- **\+** **{ tokval = PLUS; return(ARITHOP); }**
- **\-** **{ tokval = MINUS; return(ARITHOP); }**
- **>** **{ tokval = GREATER; return(RELOP); }**
- **>=** **{ tokval = GREATEREQL; return(RELOP); }**

DFAs



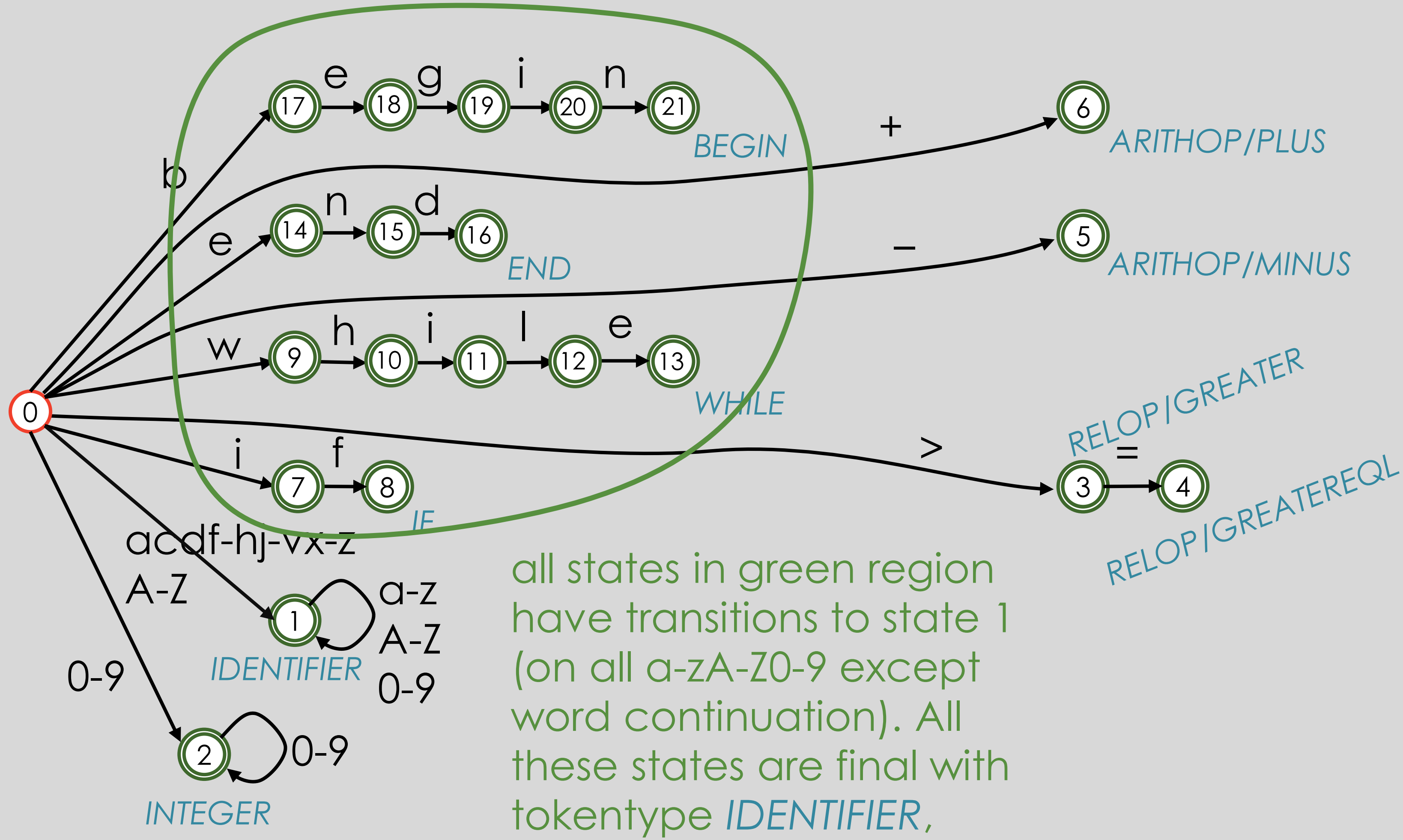
b e g i n
BEGIN
e n d
END
w h i l e
WHILE
i f
IF
a-z
A-Z
a-z
A-Z
0-9
IDENTIFIER
0-9
0-9
INTEGER
+
ARITHOP/PLUS
–
ARITHOP/MINUS
>
RELOP/GREATER
> =
RELOP/GREATEREQL

NFA



all red edges
are ε-edges

Convert NFA to DFA



all states in green region have transitions to state 1 (on all a-zA-Z0-9 except word continuation). All these states are final with tokentype *IDENTIFIER*, unless otherwise shown.

Transition Table

	a	b	c	d	e	f	g	h	i	…	A	…	Z	0	1	2	…	+	–	>	=
s0	s1	s17	s1	s1	s14	s1	s1	s1	s7		s1		s1	s2	s2	s2		s6	s5	s3	D
s1	s1	s1	s1	s1	s1	s1	s1	s1	s1		s1		s1	s1	s1	s1		D	D	D	D
s2	D	D	D	D	D	D	D	D	D		D		D	s2	s2	s2		D	D	D	D
s3	D	D	D	D	D	D	D	D	D		D		D	D	D	D		D	D	D	s4
s4	D	D	D	D	D	D	D	D	D		D		D	D	D	D		D	D	D	D
s5	D	D	D	D	D	D	D	D	D		D		D	D	D	D		D	D	D	D
s6	D	D	D	D	D	D	D	D	D		D		D	D	D	D		D	D	D	D
s7	s1	s1	s1	s1	s1	s8	s1	s1	s1		s1		s1	s1	s1	s1		D	D	D	D
s8	s1	s1	s1	s1	s1	s1	s1	s1	s1		s1		s1	s1	s1	s1		D	D	D	D
s9	s1	s1	s1	s1	s1	s1	s1	s10	s1		s1		s1	s1	s1	s1		D	D	D	D
s10	s1	s1	s1	s1	s1	s1	s1	s1	s11		s1		s1	s1	s1	s1		D	D	D	D
s11	s1	s1	s1	s1	s1	s1	s1	s1	s1		s1		s1	s1	s1	s1		D	D	D	D
s12	s1	s1	s1	s1	s13	s1	s1	s1	s1		s1		s1	s1	s1	s1		D	D	D	D
s13	s1	s1	s1	s1	s1	s1	s1	s1	s1		s1		s1	s1	s1	s1		D	D	D	D
…																					

Token Type Table

	Typecode
s0	–
s1	*IDENTIFIER*
s2	*INTEGER*
s3	*RELOP/GREATER*
s4	*RELOP/GREATEREQL*
s5	*ARITHOP/MINUS*
s6	*ARITHOP/PLUS*
s7	*IDENTIFIER*
s8	*IF*
s9	*IDENTIFIER*
s10	*IDENTIFIER*
s11	*IDENTIFIER*
s12	*IDENTIFIER*
s13	*WHILE*
...	

Table-driven Scanner

```
NextToken()
   state = s0
   lexeme = ""          // empty string
   clear stack S
   S.push(bottom)    // sentinel

   while state ≠ D do
      char = NextChar();
      lexeme = lexeme + char;
      if state is final state then
              clear stack S
      S.push(state);
      state = transition[state, char]
   end
```

```
while state is not final and
        state ≠ bottom  do
   state = S.pop( );
   truncate lexeme;
   RollBack( );
end

if state is final state then
   return Typecode[state]
else
   return invalid
```

Character Categories

- Many times, any of a set of characters may generate the same transitions on every state. (In our example, one such set is the digits, and another is the capital letters.)
- When this is the case, often it is best to clump these characters into a character category.
- By having the transition table indexed by category rather than character, we save table space, which may make the difference between it fitting in cache or not.
- The cost is an extra table lookup in the scanning loop of NextToken( ).

Table-driven Scanner with Character Categories

```
NextToken()
   state = s0
   lexeme = ""          // empty string
   clear stack S
   S.push(bottom)       // sentinel

   while state ≠ D do
      char = NextChar();
      lexeme = lexeme + char;
      if state is final state then
              clear stack S
      S.push(state);
      category = CharCat[char]
      state = transition[state, category]
   end
```

```
while state is not final and
        state ≠ bottom  do
   state = S.pop( );
   truncate lexeme;
   RollBack( );
end

if state is final state then
   return Typecode[state]
else
   return invalid
```

Direct-coded Scanners

- Another type of automated scanner is a direct-coded scanner.
- Direct-coded scanners can do away with the table lookup costs by having separate sections of code for each state and if statements to control transitions (jumps between these separate sections of code).
- See the text for an example. However, they forgot to set *state* in the code in the text.

Fixing the Text's Example

```
...
s0: char = NextChar();
    lexeme = lexeme + char;
    if state is final state then
        clear stack S
    S.push(state);

    if char = 'r'
        then goto s1
        else goto s_out
```

```
s0: state = s0;
    char = NextChar();
    lexeme = lexeme + char;
    if state is final state then
        clear stack S
    S.push(state);

    if char = 'r'
        then goto s1
        else goto s_out
```

We know s0 is not final so we don't need the **if** or the clear. We can eliminate the variable *state*.

```
s0: char = NextChar();
    lexeme = lexeme + char;
    S.push(s0);

    if char = 'r'
        then goto s1
        else goto s_out
```

Do something similar for each state. Only s2 is final and needs the "clear stack".

Hand-coded Scanners

- In our project, we are using a hand-coded scanner.
- Despite there being lexical analyzer generators, many if not most compilers use hand-coded scanners.
- Like direct-coded scanners, hand-coded scanners can eliminate table lookup costs.
- Hand-coded scanners can also reduce the overhead of the interface between the scanner and the rest of the compiler (input handler and parser). This includes things like input rollback, handling input buffering, and producing token lexemes and values.

Handling Keywords

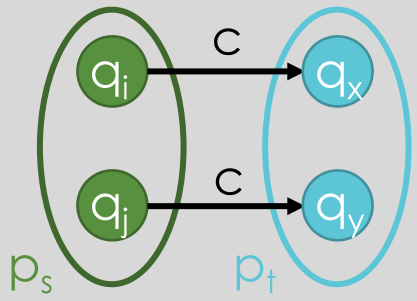
- With an automatically-generated scanner, keywords are most often handled by including regular expressions for them in the description that generates the DFA and recognizer.
- An alternative strategy is to scan keywords and identifiers together, and once scanned, determine if it is a keyword by table lookup. This is usual in hand-coded scanners and is what we do in the project.
- The tradeoff is that it simplifies the DFA and reduces the size of the tables, but it entails an extra lookup at the end of each identifier. The lookup cost can be kept to a minimum by using perfect hashing. Perfect hashing is a technique for creating a hash table with known keys that has no collisions and so a constant worst-case lookup time.

Minimizing DFAs

- The DFA generated from an NFA (or a hand-generated one) may have more states than is necesssary.   By minimizing a DFA we mean to find an equivalent DFA that has a minimum number of states.
- There are two main methods for minimizing DFAs, due to Hopcroft and Brzozowski. We will examine Hopcroft minimization.

Hopcroft's Algorithm

- We construct a partition $P = \{p_1, p_2, \ldots, p_m\}$ of the set of states of the input DFA. This partition groups states together by their behaviour in response to all input characters.
- If q_i and q_j are in partition p_s, and $\delta(q_i, c) = q_x$ and $\delta(q_j, c) = q_y$ for some alphabet character c, then q_x and q_y must both be in some partition p_t.



This is called behavioural equivalence.

Hopcroft's Algorithm

- To minimize the DFA, the partitions should be as large as possible. To construct such partitions, the algorithm starts with a rough partition that does not have behavioural equivalence. This initial partition has two sets, the final states and the nonfinal states.
- It then iteratively refines the partition until behavioural equivalence is satisfied.

Hopcroft's Algorithm

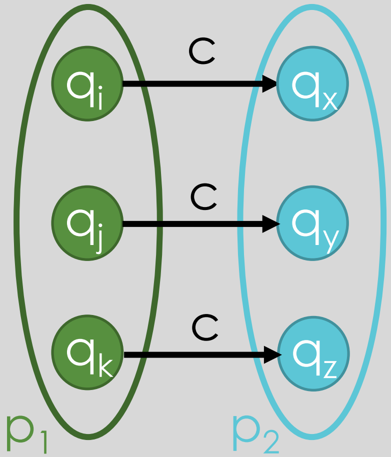
$T = \{F, Q - F\}$
$P = \emptyset$

while $P \neq T$ **do**
    $P = T$
    $T = \emptyset$
    for each set $p \in P$ **do**
        $T = T \cup \text{Split}(p)$
    end
end

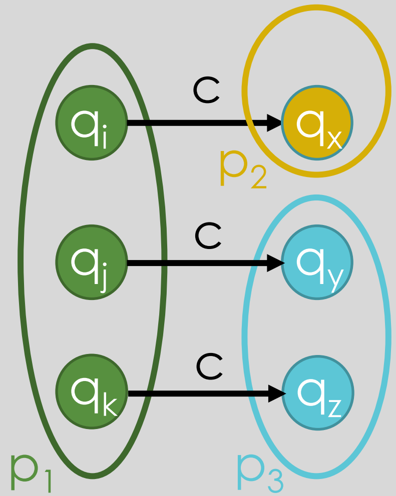
Split(S) **{**
    for each $c \in \Sigma$ **do**
        if c splits S into $S_1 \ldots S_k$
            then return $\{S_1 \ldots S_k\}$
    end
    return S
}

recall that DFA $M = (Q, \Sigma, \delta, q_0, F)$

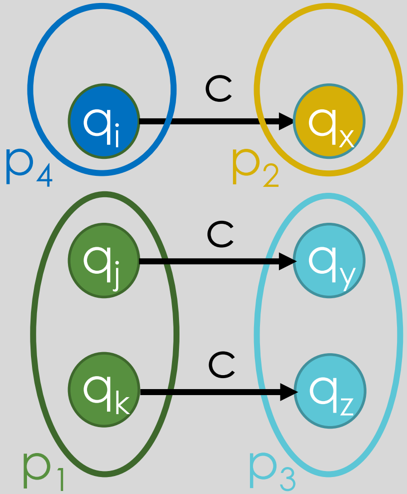
Hopcroft's Algorithm



c does not split p_1



c splits p_1



partitions after split on c

Hopcroft's Algorithm

- To construct the DFA from the final partition, create a single state to represent each set p_i of the partition.
- Add the obvious transitions between these new representative states: if some state q_i of the input DFA has a transition to q_j on character c, then the representative for the partition that q_i is in has a transition to the representative for the partition that q_j is in on the character c.
- The resulting DFA is a minimal equivalent to the input DFA.