# Lecture 4: Lexical analyzer generators, lex/flex

David Hovemeyer

September 13, 2021

601.428/628 Compilers and Interpreters

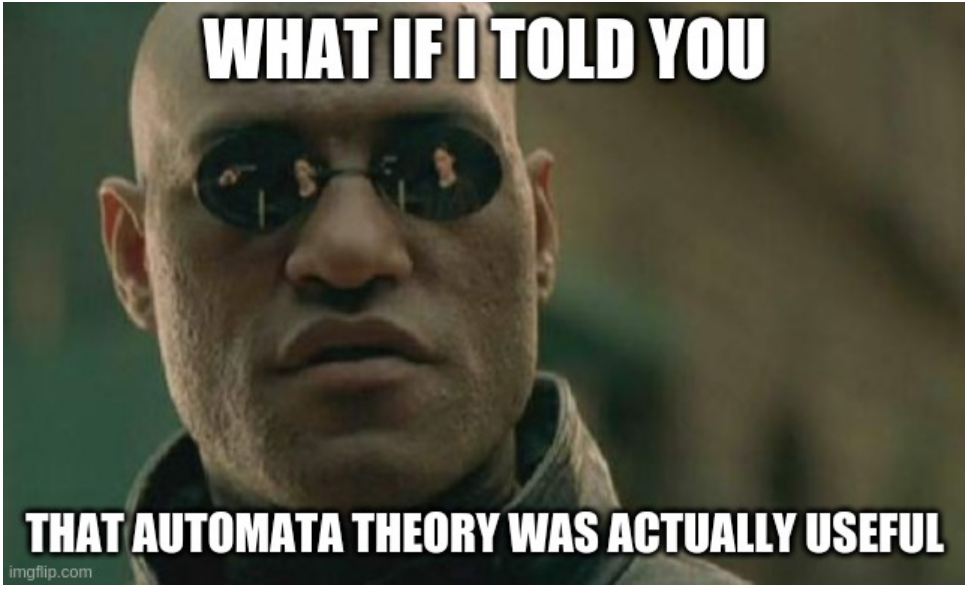
# Today

- Regular expressions
- NFAs and DFAs
- lex and flex

# Lexical analysis and regular languages

# Implementing lexical analyzers

- Lexical analyzers (a.k.a. scanners) break the source text into a sequence of tokens
- We can hand-code these
  - Not terribly difficult, but somewhat tedious
- Is there a better way to implement them?


WHAT IF I TOLD YOU
THAT AUTOMATA THEORY WAS ACTUALLY USEFUL
imgflip.com

# Regular languages!

- For any "reasonable" programming language, the lexemes of legal tokens can be described by a *regular language*
- Basic idea:
  - Each kind of token is described by a *regular expression*
  - Regular expressions can be easily converted to *nondeterministic finite automata* (NFAs)
  - The NFA for each kind of token can be combined into a single NFA which recognizes all of the different kinds of tokens
  - The combined NFA can be converted into a *deterministic finite automaton* (DFA)
  - A DFA can be easily converted into an efficient program to recognize tokens

# Formal languages, regular languages

- A *formal language* is a set of strings
- A *string* is a sequence of symbols
- *Regular languages* are a particular subset of formal languages
  - Which happen to be useful for describing character patterns of tokens in programming languages
- Each string in a regular language is a string of symbols chosen from an *alphabet*
  - For programming languages, these symbols are text characters appearing in the input source code

# Regular expressions

- Regular expressions are one way to specify a *regular language*
- Constructing a regular expression:
  - Sequence of literal symbols: generates a string
  - $\boxed{*}$ operator: means "0 or more"
  - $\boxed{+}$ operator: means "1 or more"
  - $\boxed{|}$ operator: means "or"
  - $\boxed{(}$ and $\boxed{)}$: used for grouping
  - Concatenation: if $X$ and $Y$ are regular expressions, then $XY$ is a regular expression generating all possible strings $xy$ where $x$ is in the language generated by $X$, and $y$ is in the language generated by $Y$

# Regular expressions

Examples of regular expressions:

| Regular expression | Language (set of strings) |
|---|---|
| a | a |
| aa | aa |
| a* | $\epsilon$, a, aa, aaa, ... |
| aa* | a, aa, aaa, ... |
| a+ | a, aa, aaa, ... |
| ba+ | ba, baa, baaa, ... |
| (ba)+ | ba, baba, bababa, ... |
| (a\|b) | a, b |
| a\|b* | a, $\epsilon$, b, bb, bbb, ... |
| (a\|b)* | $\epsilon$, a, b, aa, ab, ba, bb, ... |
| aa(ba)*bb | aabb, aababb, aababababb, ... |

# Insta-quiz!

Which of the following strings is *not* generated by the regular expression $(ab)^*|(ba)^*$?

A. abab
B. bababa
C. abba
D. babab
E. All of the above strings are generated

# Extended regular expression syntax

- "Basic" regular expressions are a bit limited
- For example, the regular expression for "lowercase letter" is

  (a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z)
- "Extended" regular expressions can specify *character classes*, e.g.
  - [a-z]
  - [A-Za-z]
  - [0123456789]
  - [0-9]
- Regular expression for C identifiers:

  [A-Za-z_][A-Za-z_0-9]*

# NFAs and DFAs

# Finite automata

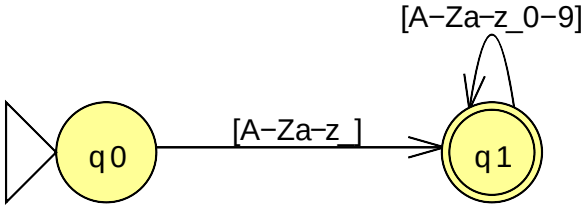
- A *finite automaton* is another way to specify a regular language
- Acts as a *recognizer* for strings in a regular language
  - If it accepts a string, it's in the language
  - If it rejects a string, it's not in the language

# Finite automata concepts

- Has *states* and *transitions*
- One state is designated as the *start state*
- At least one state is designated as a *final state*
- Each transition is labeled with one symbol
- Recognition process:
  - Start in start state
  - Following a (non-epsilon) transition consumes one symbol from the candidate string
  - If the current state is a final state when end of string is reached, it's in the language
  - Otherwise, string is not in the language

# Finite automata

Finite automaton recognizing C identifiers:




**Important**: for simplicity, we're labeling transitions with character classes; it's important to understand that this is just a shorthand notation for multiple transitions

- For example, [A-Za-z_] matches 53 characters, so the arrow from q0 to q1 is really 53 distinct transitions

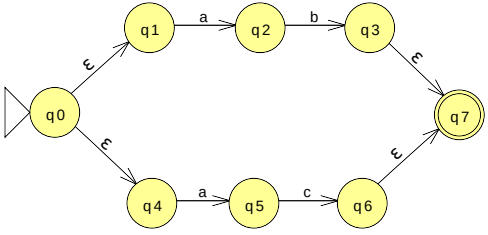
# Deterministic finite automata

- The example finite automaton on the previous slide is a *deterministic* finite automaton (DFA)
- "Deterministic" means that
  - In any state, there aren't multiple outgoing transitions (to different "destination" states) labeled with the same symbol, and
  - There aren't any epsilon transitions
- As a DFA processes a candidate string, there is always a single current state

# Nondeterministic finite automata

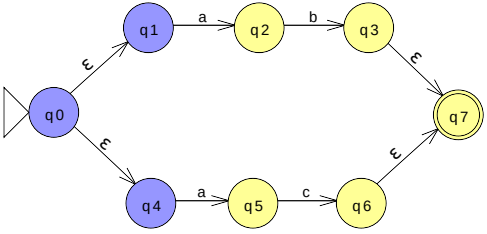
- A *nondeterministic* finite automaton (NFA) has
  - States with multiple outgoing transitions on the same symbol, and/or
  - One or more epsilon transitions
- An epsilon transition does not consume an symbol from the input string
- When an NFA processes a candidate string, it can be in multiple states at the same time
- Candidate string is accepted if, when end of string is reached, current set of states contains any accepting state

# Example NFA



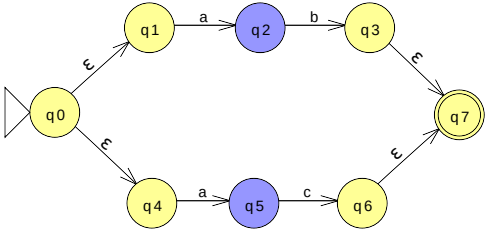

| States | Candidate string |
|---|---|
| | $\wedge$ab |

# Example NFA



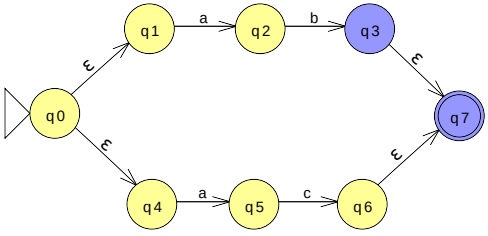

| States | Candidate string |
|---|---|
| { q0, q1, q4 } | ∧ab |

# Example NFA




| States | Candidate string |
|---|---|
| { q0, q1, q4 } | $_\wedge$ab |
| { q2, q5 } | a$_\wedge$b |

# Example NFA



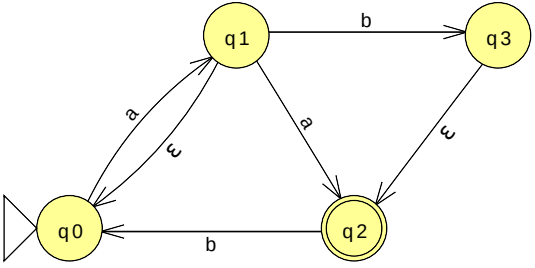

| States | Candidate string |
|---|---|
| { q0, q1, q4 } | $_\wedge$ab |
| { q2, q5 } | a$_\wedge$b |
| { q3, q7 } | ab$_\wedge$ |

When end of string is reached, the current set of states contains a final state (q7), so the string is accepted

# Insta-quiz!

What set of states is reached when the NFA on the right recognizes the string aab?

A. { q0 }
B. { q0, q3 }
C. { q1, q3 }
D. { q0, q2, q3 }
E. None of the above



# Eliminating nondeterminism

- Nondeterminism can always be eliminated!
- I.e., for any NFA, we can create a DFA that recognizes the same language
  - NFA with $n$ states could yield a DFA with $2^n$ states, but that's not likely to occur in practice
- Basic idea: simulate behavior of all possible inputs to the NFA, map each reachable set of NFA states to a corresponding DFA state
- We'll show an example of how this works soon

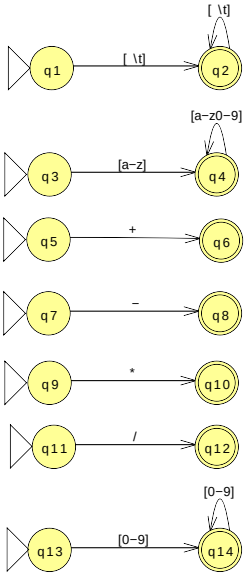
# Example language

Regular expressions for tokens in a simple programming language:

| Token kind | Regular expression | Note |
|---|---|---|
| Whitespace | [␣\t]+ | Not a token per se, but does need to be recognized by the lexer |
| Identifier | [a-z][a-z0-9]* | |
| Addition | \+ | Literal plus symbol, not "1 or more" |
| Subtraction | - | |
| Multiplication | * | Literal asterisk |
| Division | / | |
| Number | [0-9]+ | |

# Example language: per-token FAs

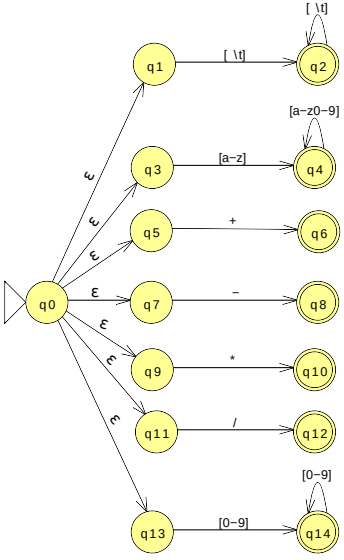
Translate each regular expression into a DFA (this can be automated)



# Example language: unified NFA

Combine individual token FAs into a single NFA

NFA recognizes union of all lexemes (for all kinds of tokens)

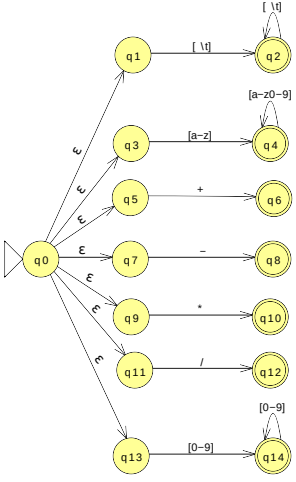


# Example language: conversion to DFA

- Now, let's convert the unified NFA into a DFA
- For each reachable set of states in NFA, create corresponding state in DFA
- Add transitions to DFA corresponding to transitions between reachable NFA state sets
- See textbook for full algorithm

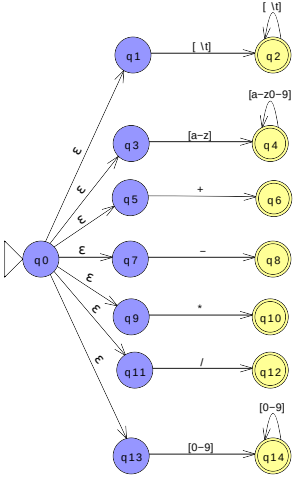
# NFA to DFA conversion

| NFA states | DFA state |
|---|---|



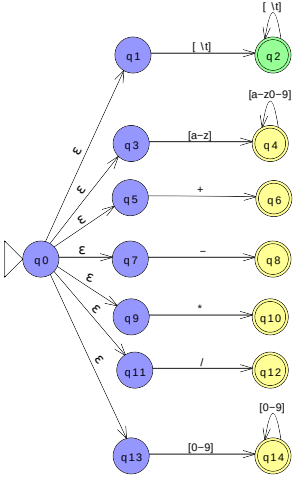
# NFA to DFA conversion

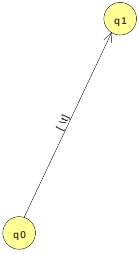
| NFA states | DFA state |
|---|---|
| { 0,1,3,5,7,9,11,13 } | 0 |




# NFA to DFA conversion

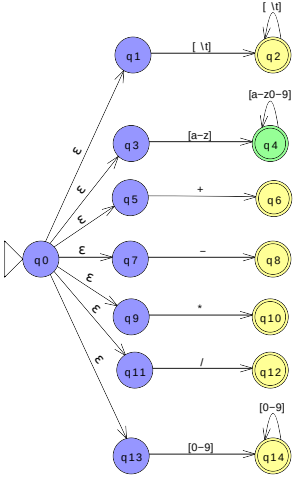
| NFA states | DFA state |
|---|---|
| { 0,1,3,5,7,9,11,13 } | 0 |
| { 2 } | 1 |

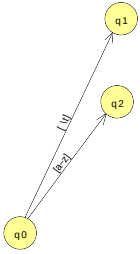


# NFA to DFA conversion

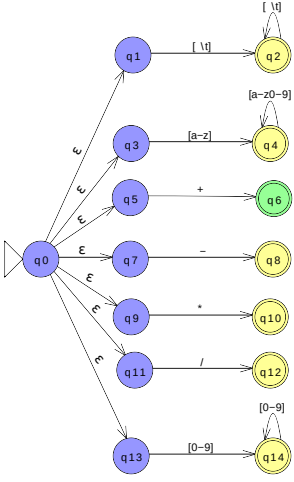
| NFA states | DFA state |
|---|---|
| { 0,1,3,5,7,9,11,13 } | 0 |
| { 2 } | 1 |
| { 4 } | 2 |

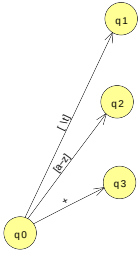


# NFA to DFA conversion

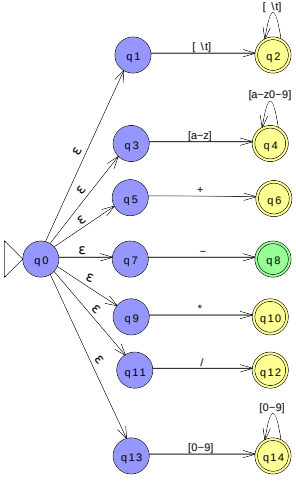
| NFA states | DFA state |
|---|---|
| { 0,1,3,5,7,9,11,13 } | 0 |
| { 2 } | 1 |
| { 4 } | 2 |
| { 6 } | 3 |

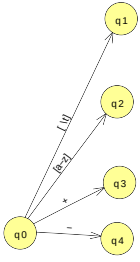


# NFA to DFA conversion

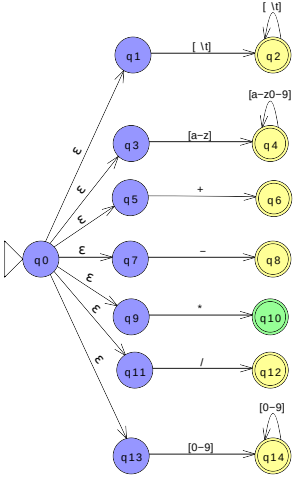
| NFA states | DFA state |
|---|---|
| { 0,1,3,5,7,9,11,13 } | 0 |
| { 2 } | 1 |
| { 4 } | 2 |
| { 6 } | 3 |
| { 8 } | 4 |

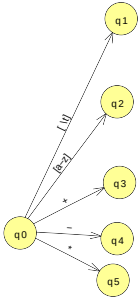


# NFA to DFA conversion

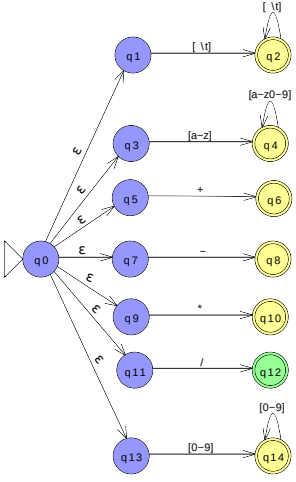
| NFA states | DFA state |
|---|---|
| { 0,1,3,5,7,9,11,13 } | 0 |
| { 2 } | 1 |
| { 4 } | 2 |
| { 6 } | 3 |
| { 8 } | 4 |
| { 10 } | 5 |

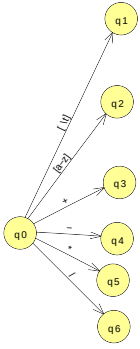


# NFA to DFA conversion

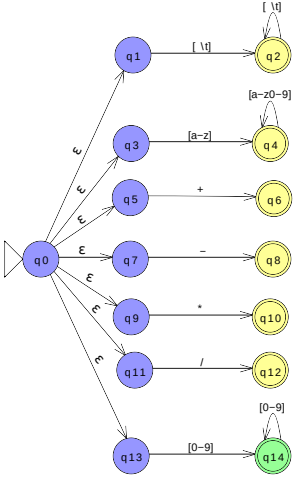
| NFA states | DFA state |
|---|---|
| { 0,1,3,5,7,9,11,13 } | 0 |
| { 2 } | 1 |
| { 4 } | 2 |
| { 6 } | 3 |
| { 8 } | 4 |
| { 10 } | 5 |
| { 12 } | 6 |

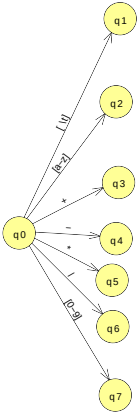


# NFA to DFA conversion

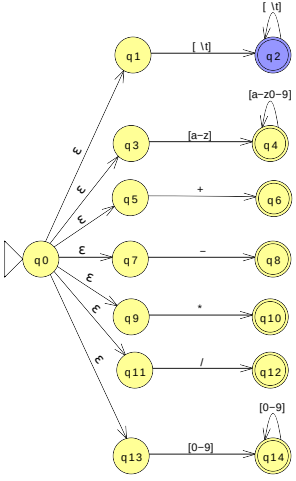
| NFA states | DFA state |
|---|---|
| { 0,1,3,5,7,9,11,13 } | 0 |
| { 2 } | 1 |
| { 4 } | 2 |
| { 6 } | 3 |
| { 8 } | 4 |
| { 10 } | 5 |
| { 12 } | 6 |
| { 14 } | 7 |

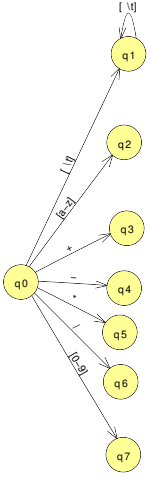


# NFA to DFA conversion

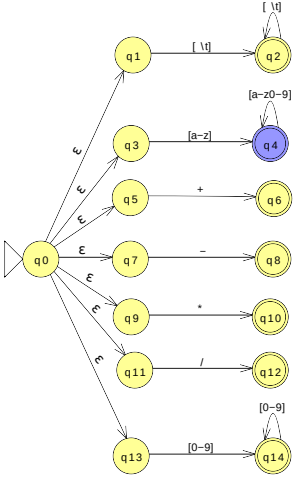
| NFA states | DFA state |
|---|---|
| { 0,1,3,5,7,9,11,13 } | 0 |
| { 2 } | 1 |
| { 4 } | 2 |
| { 6 } | 3 |
| { 8 } | 4 |
| { 10 } | 5 |
| { 12 } | 6 |
| { 14 } | 7 |

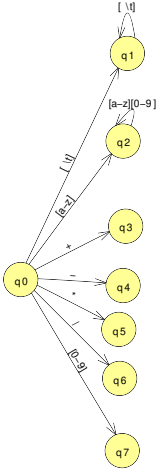


# NFA to DFA conversion

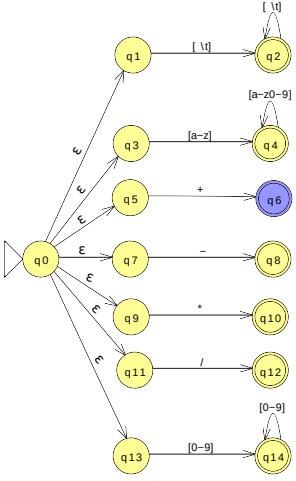
| NFA states | DFA state |
|---|---|
| { 0,1,3,5,7,9,11,13 } | 0 |
| { 2 } | 1 |
| { 4 } | 2 |
| { 6 } | 3 |
| { 8 } | 4 |
| { 10 } | 5 |
| { 12 } | 6 |
| { 14 } | 7 |

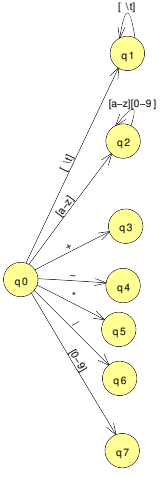


# NFA to DFA conversion

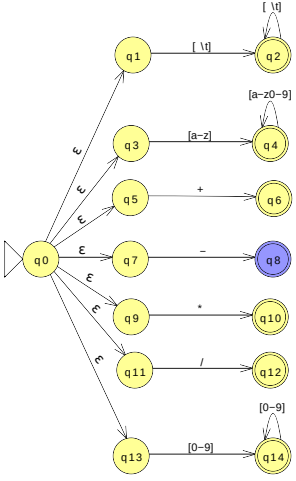
| NFA states | DFA state |
|---|---|
| { 0,1,3,5,7,9,11,13 } | 0 |
| { 2 } | 1 |
| { 4 } | 2 |
| { 6 } | 3 |
| { 8 } | 4 |
| { 10 } | 5 |
| { 12 } | 6 |
| { 14 } | 7 |

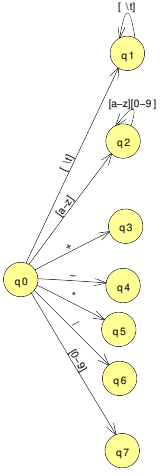


# NFA to DFA conversion

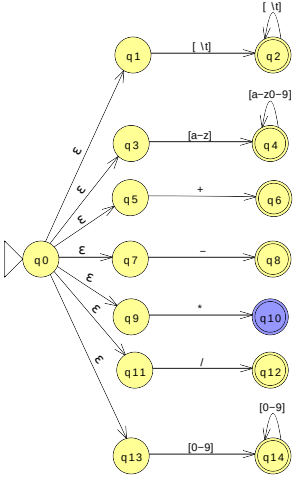
| NFA states | DFA state |
|---|---|
| { 0,1,3,5,7,9,11,13 } | 0 |
| { 2 } | 1 |
| { 4 } | 2 |
| { 6 } | 3 |
| { 8 } | 4 |
| { 10 } | 5 |
| { 12 } | 6 |
| { 14 } | 7 |

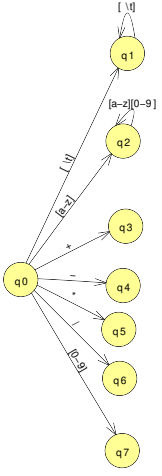


# NFA to DFA conversion

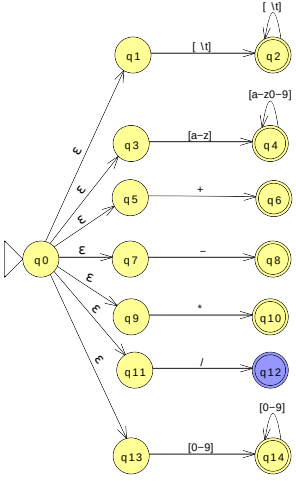
| NFA states | DFA state |
|---|---|
| { 0,1,3,5,7,9,11,13 } | 0 |
| { 2 } | 1 |
| { 4 } | 2 |
| { 6 } | 3 |
| { 8 } | 4 |
| { 10 } | 5 |
| { 12 } | 6 |
| { 14 } | 7 |

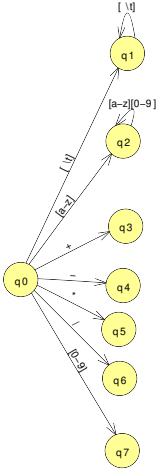


# NFA to DFA conversion

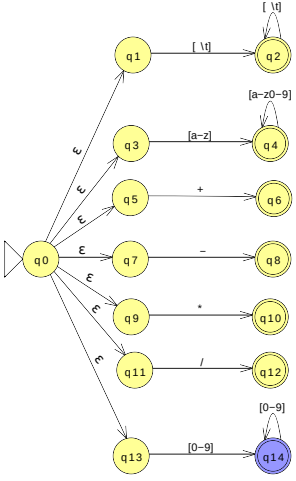
| NFA states | DFA state |
|---|---|
| { 0,1,3,5,7,9,11,13 } | 0 |
| { 2 } | 1 |
| { 4 } | 2 |
| { 6 } | 3 |
| { 8 } | 4 |
| { 10 } | 5 |
| { 12 } | 6 |
| { 14 } | 7 |

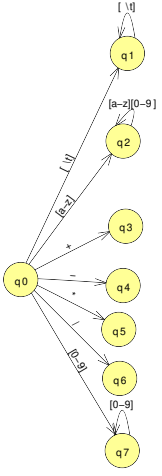


# NFA to DFA conversion

| NFA states | DFA state |
|---|---|
| { 0,1,3,5,7,9,11,13 } | 0 |
| { 2 } | 1 |
| { 4 } | 2 |
| { 6 } | 3 |
| { 8 } | 4 |
| { 10 } | 5 |
| { 12 } | 6 |
| { 14 } | 7 |



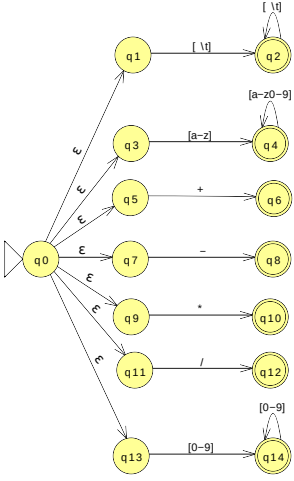



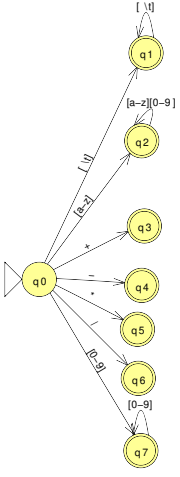
# NFA to DFA conversion

| NFA states | DFA state |
|---|---|
| { 0,1,3,5,7,9,11,13 } | 0 |
| { 2 } | 1 |
| { 4 } | 2 |
| { 6 } | 3 |
| { 8 } | 4 |
| { 10 } | 5 |
| { 12 } | 6 |
| { 14 } | 7 |

Final steps:

- Make q0 of DFA the start state
- Each NFA state set containing a final state has its corresponding DFA state marked as final



# Table-driven recognition

Any DFA can be represented as a table indicating, for each DFA state, which transitions to other DFA states exist

Given a table, it's trivial to create a program to recognize the language

Basic idea: repeatedly

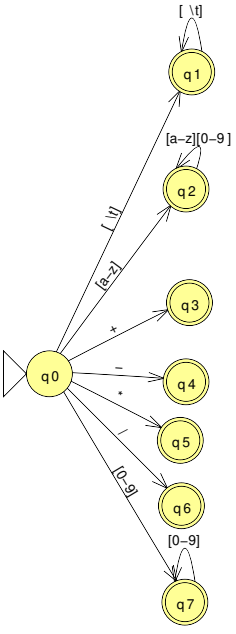
- Read an input character
- See if there is a transition to another state

When we reach EOF, or if there's no transition available, see if we're in a final state

- Which one we're in tells us what kind of token we've recognized

# DFA transition table

| State | [ \t] | [a-z] | + | - | * | / | [0-9] |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 1 | – | – | – | – | – | – |
| 2 | – | 2 | – | – | – | – | 2 |
| 3 | – | – | – | – | – | – | – |
| 4 | – | – | – | – | – | – | – |
| 5 | – | – | – | – | – | – | – |
| 6 | – | – | – | – | – | – | – |
| 7 | – | – | – | – | – | – | 7 |



# Some details

A few issues required to make this work:

- NFA to DFA conversion algorithm doesn't guarantee a *minimal DFA*
  - Can use DFA minimization algorithm
- A final DFA state could correspond to multiple NFA final states
  - For example, keywords are generally matched by the same regular expression pattern as identifiers
  - For example, if a keyword is recognized, the NFA will also be in the final state for identifiers
  - Solution is to prioritize kinds of tokens
    - E.g., keywords take priority over identifiers

# Can we put this into practice?

Is this a basis for implementing practical lexical analyzers?

It would be very time-consuming to build NFAs and DFAs by hand. For example, the notation "[a-z]" is really 26 different characters requiring 26 different FA transitions, 26 columns in the DFA table, etc.

But, could we automate this process?

# lex and flex

# lex and flex

lex and flex are *lexical analyzer generators*

- lex: developed at AT&T Bell Labs, distributed with Unix, not really used any more
- flex: modern open-source replacement for lex

They automate the process we've just covered

And, they're surprisingly easy to use

# flex lexer specification

```
%{
C preamble (includes, definitions, global vars)
%}

flex options

%%

patterns and actions

%%

C functions
```

# Example flex program

```
%{
#include <stdio.h>

enum TokenKind {
  TOK_IDENTIFIER = 1,
  TOK_PLUS,
  TOK_MINUS,
  TOK_TIMES,
  TOK_DIVIDE,
  TOK_NUMBER,
};
%}

%option noyywrap

%%

[ \t\n]+          { /* whitespace, ignore */ }
[a-z][a-z0-9]*    { return TOK_IDENTIFIER; }
"+"               { return TOK_PLUS; }
"-"               { return TOK_MINUS; }
"*"               { return TOK_TIMES; }
"/"               { return TOK_DIVIDE; }
[0-9]+            { return TOK_NUMBER; }

%%
```

```
int main(void) {
  yyin = stdin;
  int kind;
  while ((kind = yylex()) != 0) {
    printf("%d:%s\n", kind, yytext);
  }
  return 0;
}
```

Source code in `lexdemo.zip` linked from course website

# Running the example program

User input in **bold**:

```
$ ./lexdemo
foo + bar * 42
1:foo
2:+
1:bar
4:*
6:42
```

# How flex programs work

Basic idea:

- Sequence of *patterns* and *actions*
- When a pattern is recognized, the corresponding action is executed
  - If input matches multiple patterns, the pattern appearing earliest takes priority
- Action can return control to parser, or continue recognizing more input
  - If action has a `return` statement, it indicates to the parser what kind of token was recognized

# yylex() function

The `yylex()` function reads input until both

- A pattern is matched, and
- The pattern's action executes a `return`

The value returned by the action is the return value of `yylex()`

Returns 0 when end of input is reached

- Token kind values should thus be non-zero

# yyin, yytext

`yyin`: A `FILE*` variable from which input will be read

`yytext`: This is a (nul terminated) C character string containing the lexeme of the recognized pattern

# yylval

A variable of the union type `YYSTYPE` (usually declared by the parser)

Members of this union allow different grammar symbols to have different kinds of values associated with them

- Lexer actions can assign to one of the fields
- We'll see how this works when we cover yacc/bison