HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 8: Lexing

Stephen Chong

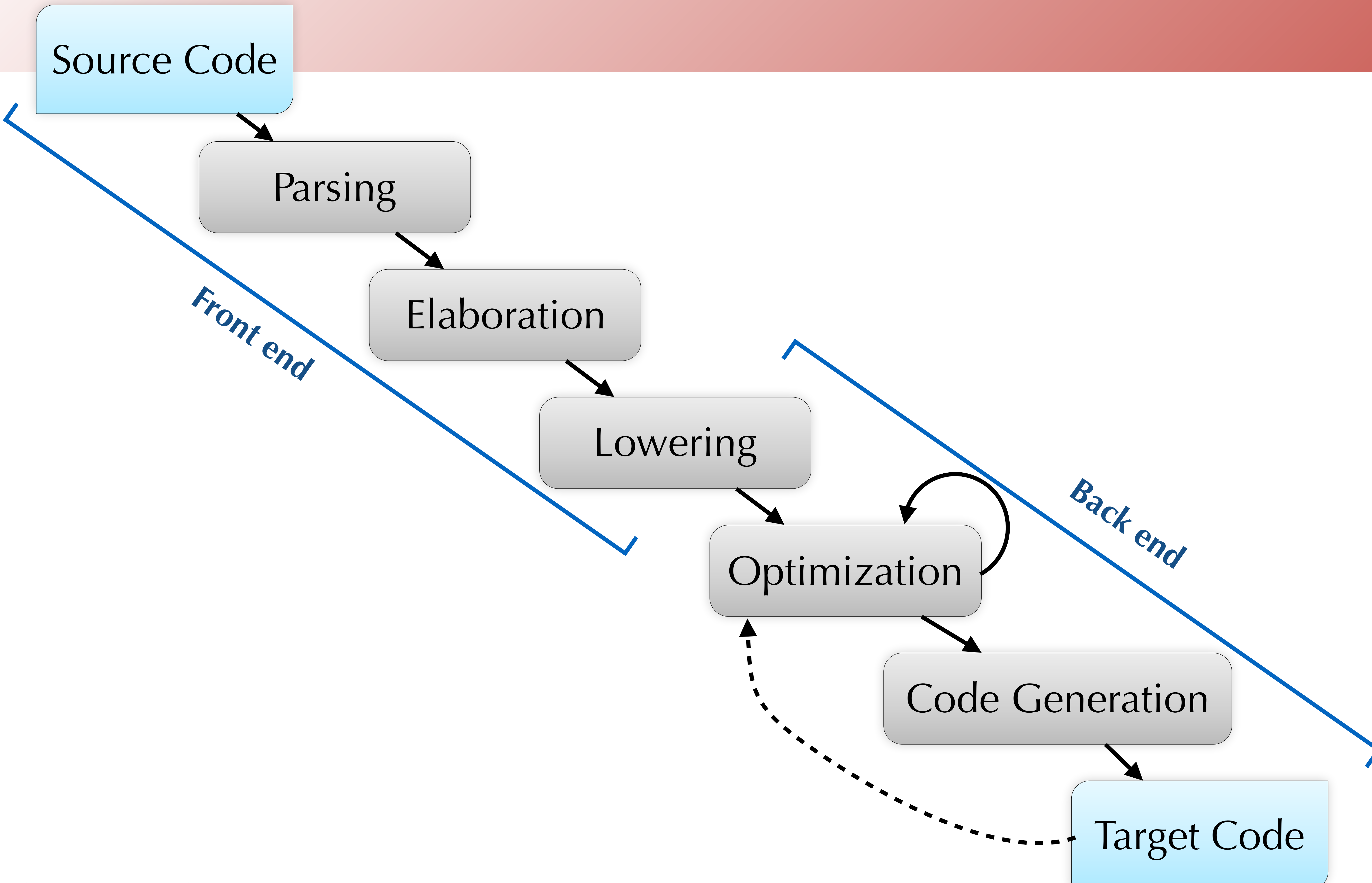
https://www.seas.harvard.edu/courses/cs153

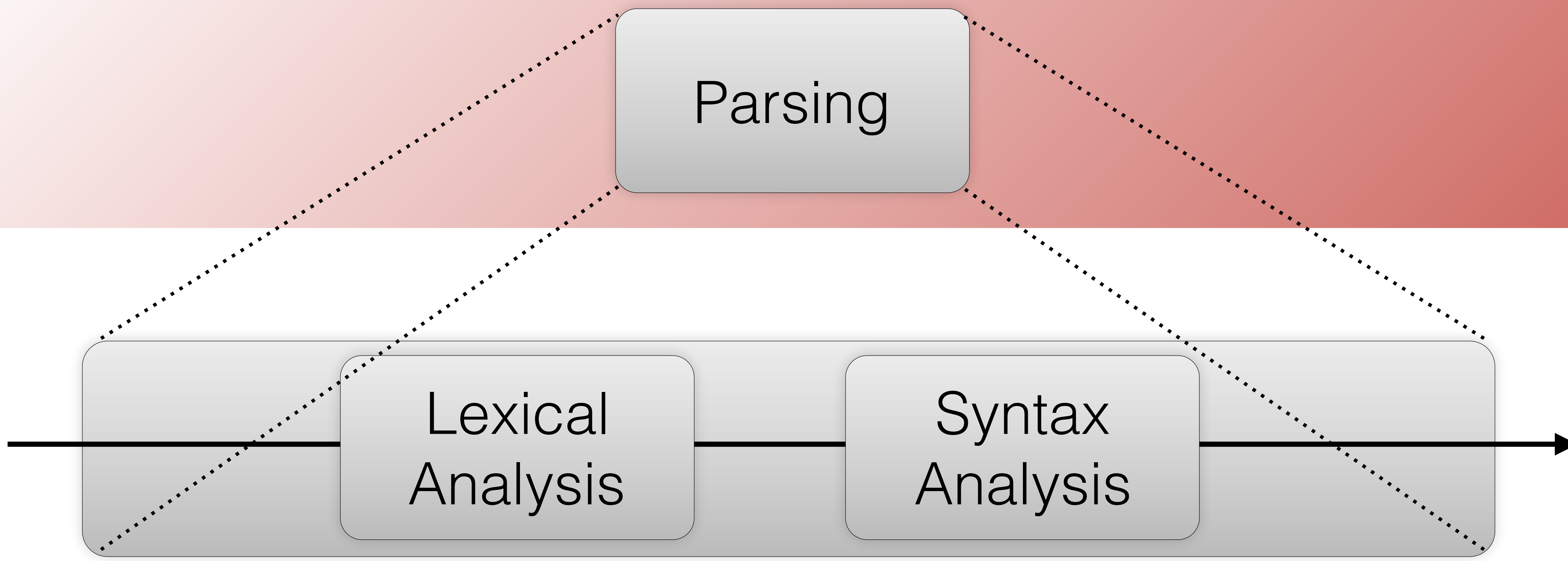
Contains content from lecture notes by Steve Zdancewic and Greg Morrisett

Announcements

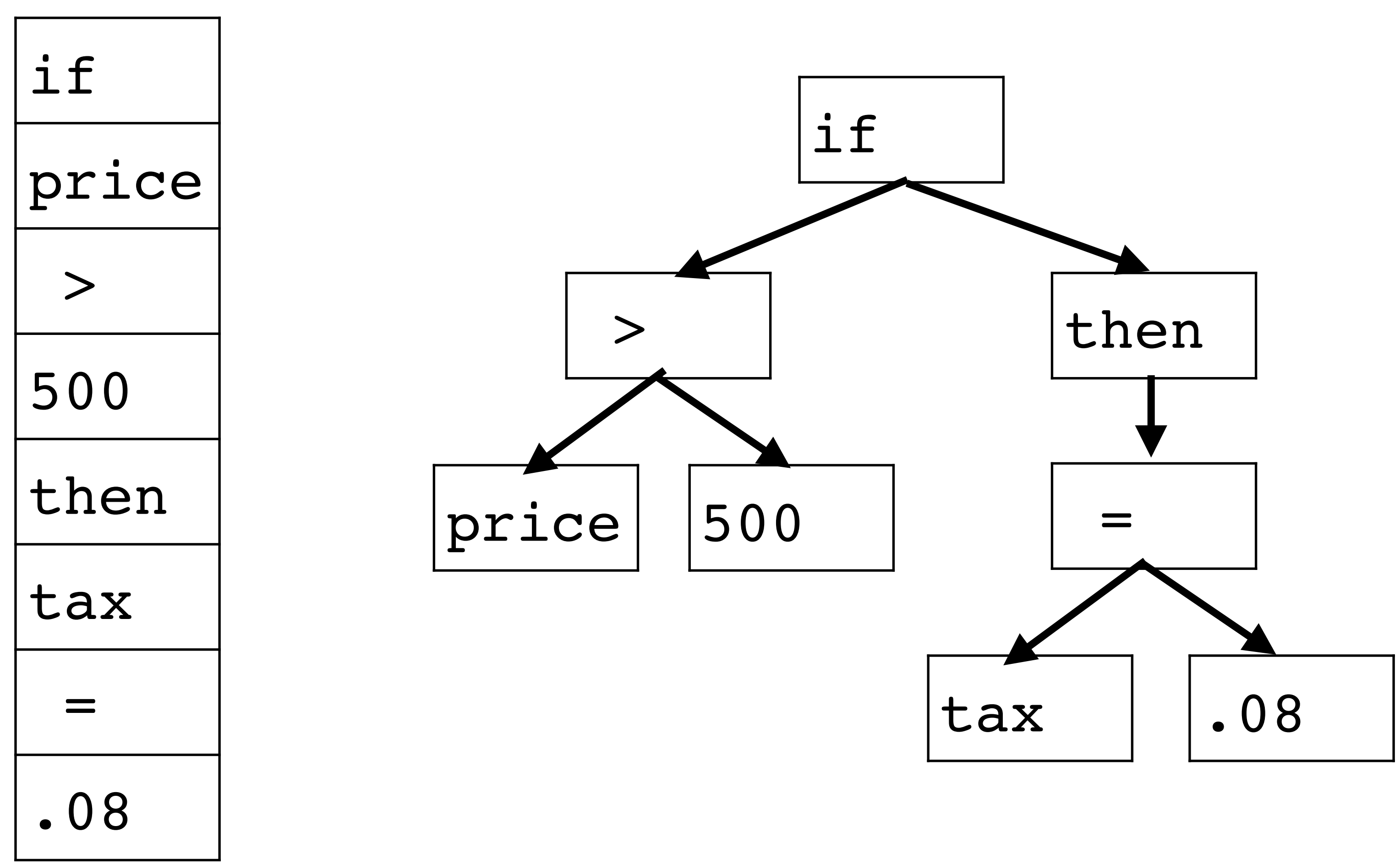
- Homework 3 (LLVMlite) out
  - Due Tuesday Oct 15
  - Start early!!!
    - Challenging assignment; HW4 will be released Oct 8

Basic Architecture: Review

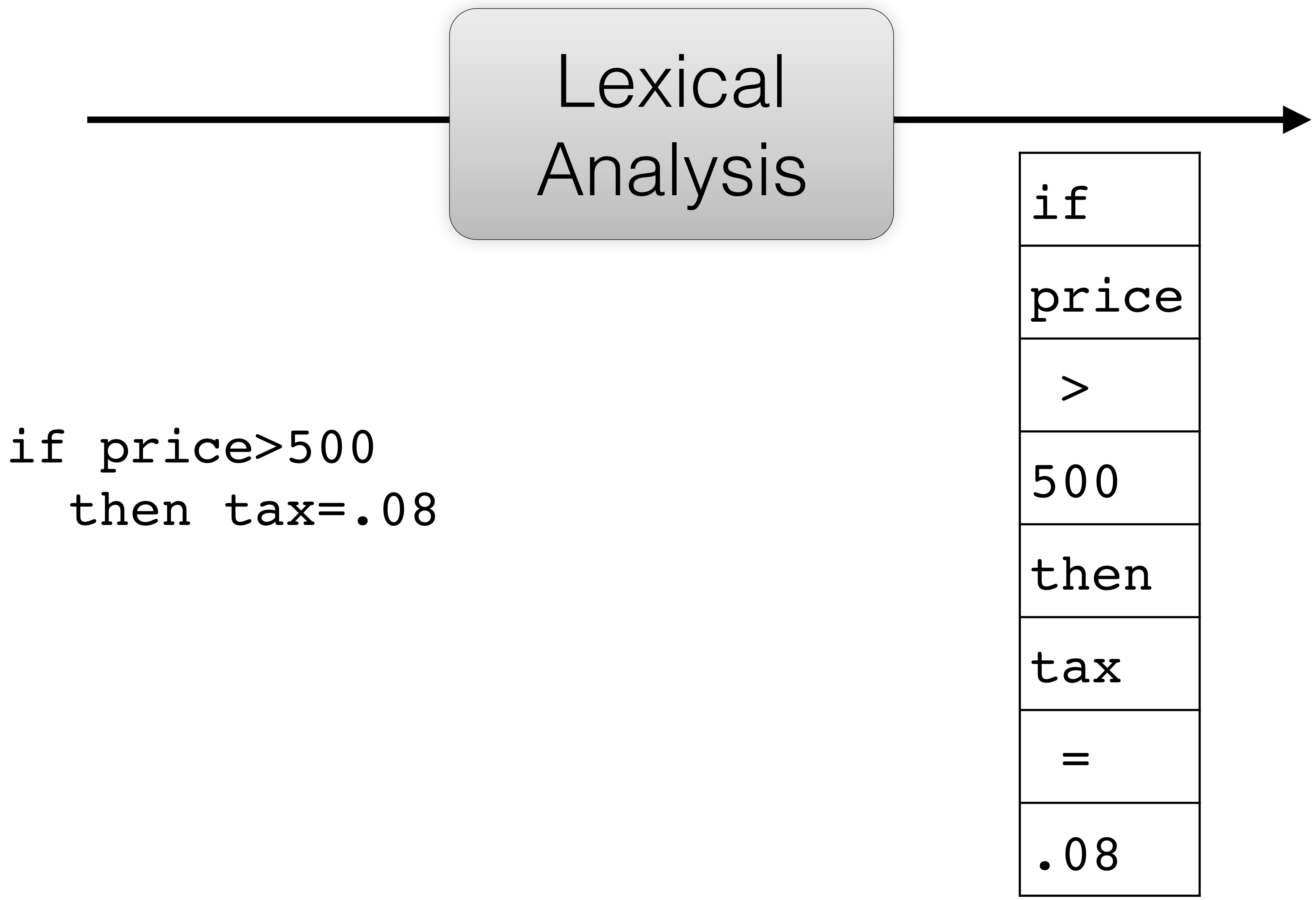





```
if price>500
  then tax=.08
```



Today



- **Lexical analysis**: breaks input sequence of characters into individual words, aka "tokens"

Lexical Tokens

- A language classifies lexical tokens into **token types**

Type	**Examples**
ID	`foo n14 last`
NUM	`73 0 00 515 082`
REAL	`66.1 .5 10. 1e67`
IF	`if`
COMMA	`,`
NOTEQ	`!=`
LPAREN	`(`

  - So, a token type specifies a set of acceptable tokens.
- **Reserved words** are tokens that cannot be used as identifiers
  - E.g., IF, VOID, RETURN

Example 1

- Given a program

```
if (price>500)
    then tax=.08
```

the lexical analysis returns the sequence of tokens

IF LPAREN ID(`price`) GT NUM(`500`) RPAREN THEN ID(`tax`) EQ REAL(`0.08`)

Example 2

- Given a program

```
if (price>500)
    then tax=1xab
```

the lexical analysis returns

ERROR

because 1xab is neither a number nor an identifier.

Type	Examples
ID	`foo n14 last`
NUM	`73 0 00 515 082`
THEN	`then`

- The lexical analysis can help in reporting where an error occurs in the code.
  - By recognizing '\n' as a token and incrementing the line number.

Example 3

Type	Examples
ID	`foo n14 last`
NUM	`73 0 00 515 082`
THEN	`then`

- Given a program

```
if (price>500)
   thn tax=.08
```

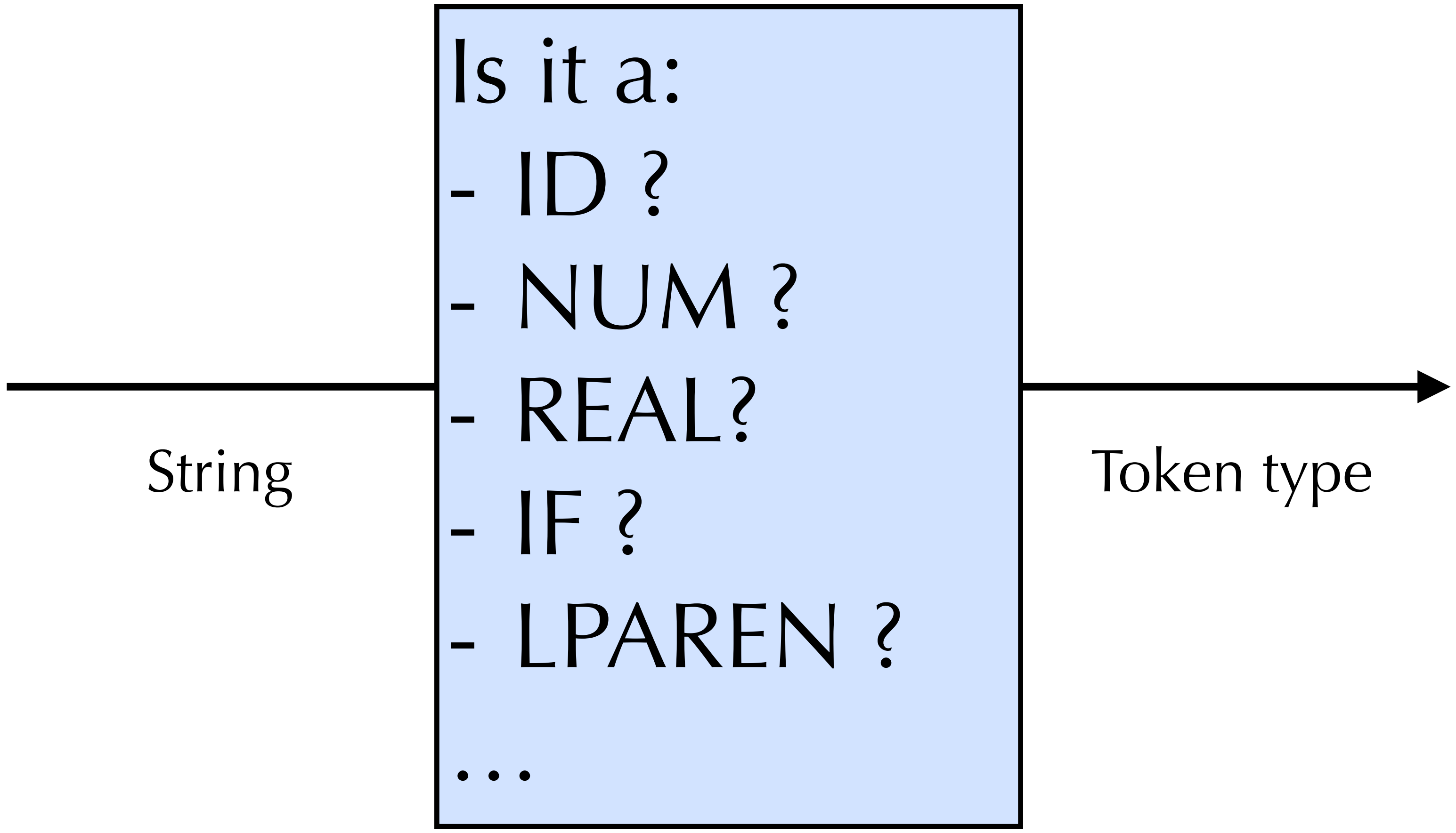
the lexical analysis returns

IF LPAREN ID(`price`) GT NUM(`500`) RPAREN ID(`thn`) ID(`tax`) EQ REAL(`0.08`)

- Is this an error at the level of lexical analysis?
  - No, it is an error at the level of syntax analysis (next lectures)!

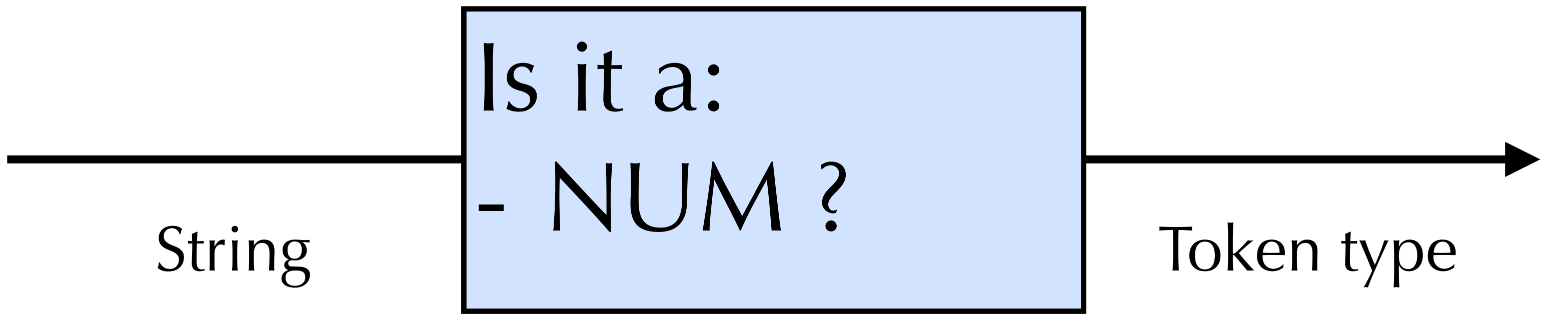
Towards Implementing A Lexical Analysis

- Recall: Lexical analysis breaks input into tokens.
- The lexical analysis needs to decide the token type for a given string (i.e., sequence of characters).



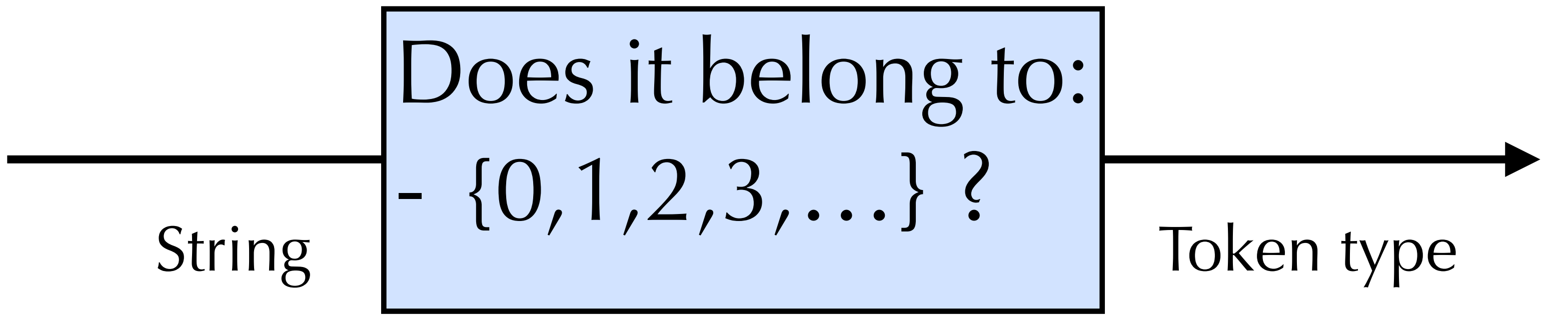
Let's simplify…

- Recall: Lexical analysis breaks input into tokens.
- The lexical analysis needs to decide the token type for a given string (i.e., sequence of characters).



A Set Membership Question

- Recall: a token type specifies a **set** of acceptable tokens (i.e., strings).
  - The set of acceptable tokes for NUM is {0,1,2,3,…}.
- But this set is infinite…



A Set Membership Question

- How can we mechanically decide if a string belongs to a (possibly infinite) set S of strings?
- An approach:
  - Use a finite representation of S.
    - Regular expressions
  - Check whether the string is accepted by such a finite representation.
    - Deterministic finite-state automata

Regular Expressions

- Each regular expression represents a set of strings.
- Examples
  - `( 0 | 1 )* 0`
    - Binary numbers that are multiples of 2
  - `b*(abb*)*(a|ε)`
    - Strings of `a`'s and `b`'s without consecutive `a`'s
  - `(a|b)*aa(a|b)*`
    - Strings of `a`'s and `b`'s with consecutive `a`'s

Regular Expressions (RE)

- Grammar
  - $\varnothing$ (matches no string)
  - ε (epsilon – matches empty string)
  - Literals ('a', 'b', '2', '+', etc.) drawn from alphabet
  - Concatenation ($R_1\ R_2$)
  - Alternation ($R_1 \mid R_2$)
  - Kleene star (R^*)

Set of Strings

- $[\![\varnothing]\!] = \{ \}$
- $[\![\varepsilon]\!] = \{ \text{“”} \}$
- $[\![\text{‘a’}]\!] = \{ \text{“a”} \}$
- $[\![R_1\ R_2]\!] = \{ s \mid s = \alpha\hat{}\beta \text{ and } \alpha \in [\![R_1]\!] \text{ and } \beta \in [\![R_2]\!] \}$
- $[\![R_1 \mid R_2]\!] = \{ s \mid s \in [\![R_1]\!] \text{ or } s \in [\![R_2]\!] \}$
  $= [\![R_1]\!] \cup [\![R_2]\!]$
- $[\![R^*]\!] = [\![\ \varepsilon \mid RR^*\ ]\!]$
  $= \{ s \mid s = \text{“”} \text{ or } s = \alpha\hat{}\beta \text{ and } \alpha \in [\![R]\!]$
  $\text{and } \beta \in [\![R^*]\!] \}$

Syntactic Sugar

- [0-9] shorthand for 0 | 1 | … | 9
- R? shorthand for (R | ε) (i.e., R is optional)
- R+ shorthand for (R R*) (i.e., at least one R)

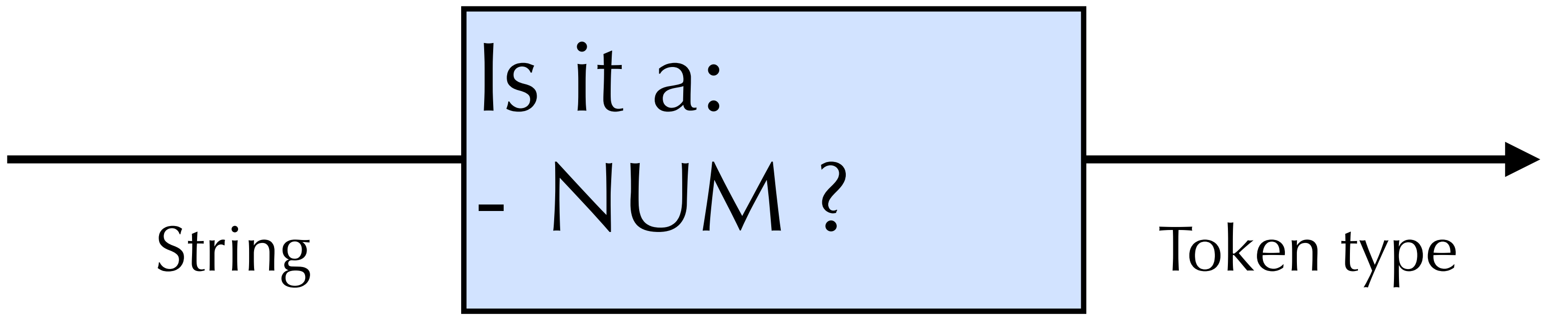
Regular Expressions to Specify Token Types!

Reg Exp	Token Type
`if`	IF
`[a-z][a-z0-9]*`	ID
`[0-9]+`	NUM
`([0-9]+ "." [0-9]*) | ([0-9]* "." [0-9]+)`	REAL

- Question: What is the token type of input `iffy`?
  - We want the token ID(`iffy`) rather than IF.
- In general, we want the longest match:
  - longest initial substring of the input that can match a regular expression is taken as next token

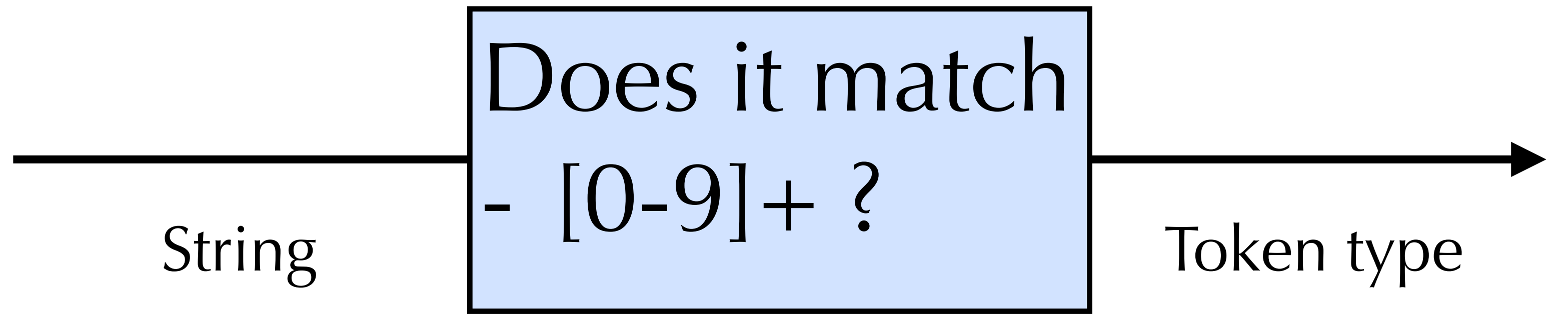
Recall: A Set Membership Question

- Lexical analysis breaks input into tokens.
- The lexical analysis needs to decide the token type for a given string (i.e., sequence of characters).



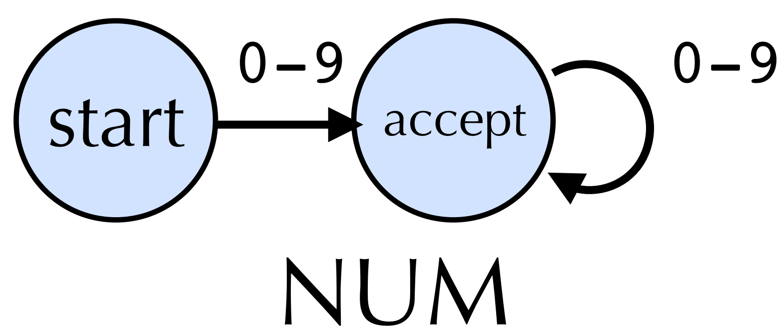
A Matching Question

- Lexical analysis breaks input into tokens.
- The lexical analysis needs to decide the token type for a given string (i.e., sequence of characters).

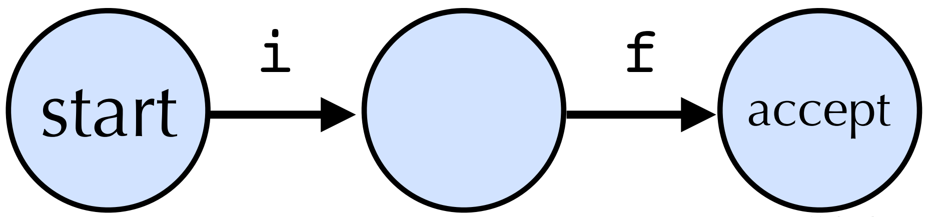


From RE to DFA

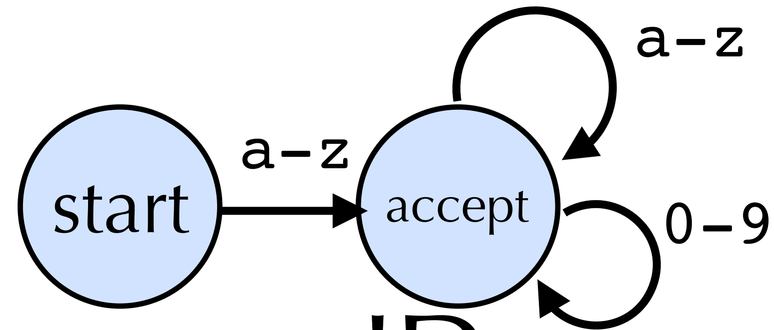
- A Deterministic Finite-state Automaton (DFA) can be used to decide whether an input matches a regular expression.
- Example: DFA for regular expression [0-9]+ :



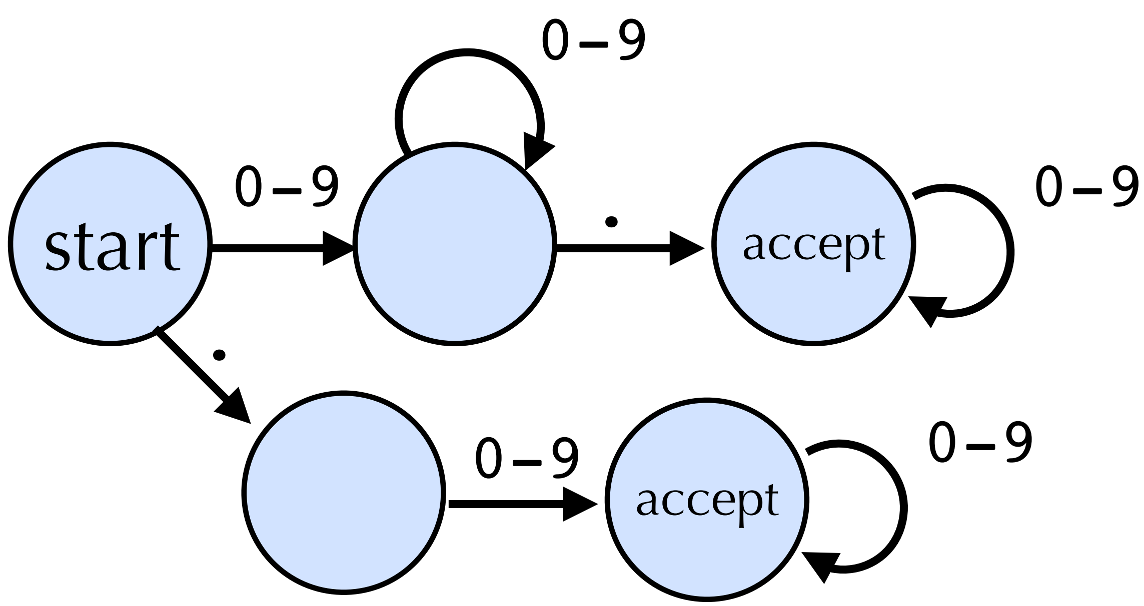
Other DFAs



IF

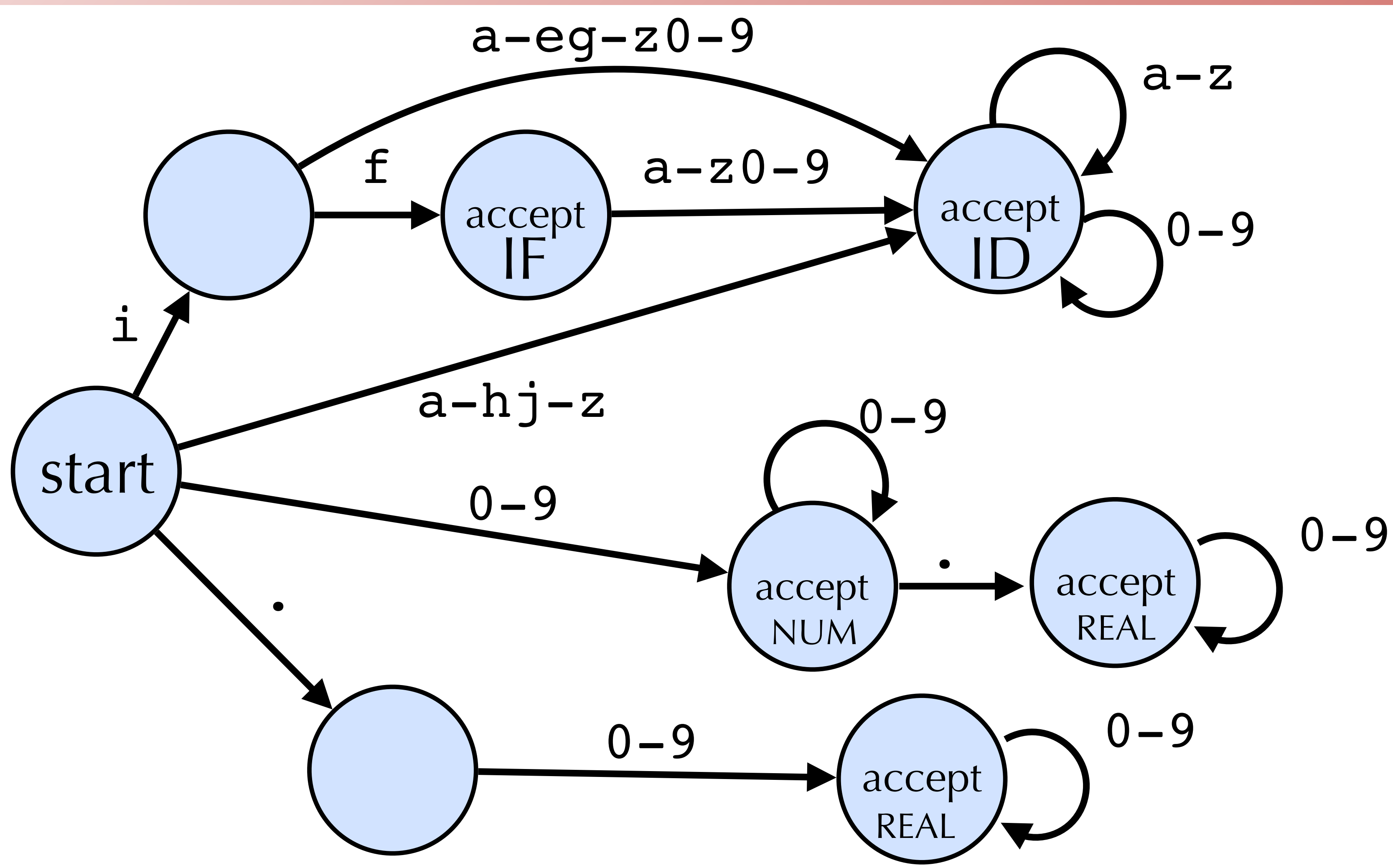


ID



REAL

Combined Finite Automaton



- This DFA takes as an input a sequence of characters and returns a Token Type (if the input is accepted).
    - So, this DFA can be used for Lexical Analysis.

Using DFAs

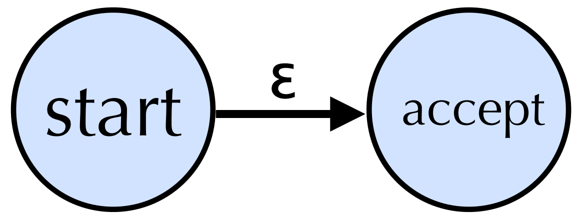
- Usually record transition function as array indexed by state and characters (i.e., transition table)
  - See Appel Chap 2.3 for an example.

How is a RE converted to a DFA?

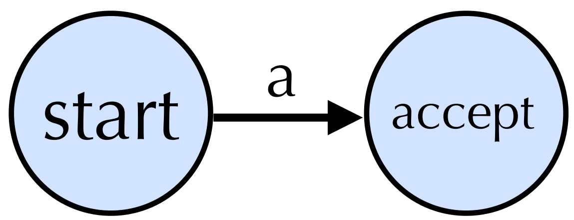
1. Convert RE to a Nondeterministic Finite-state Automaton (NFA).
2. Convert NFA to DFA.

RE to NFA conversion

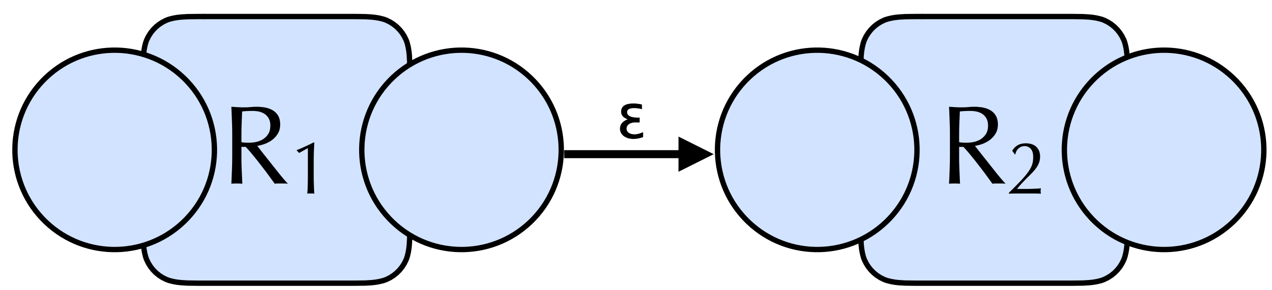
- Epsilon ε



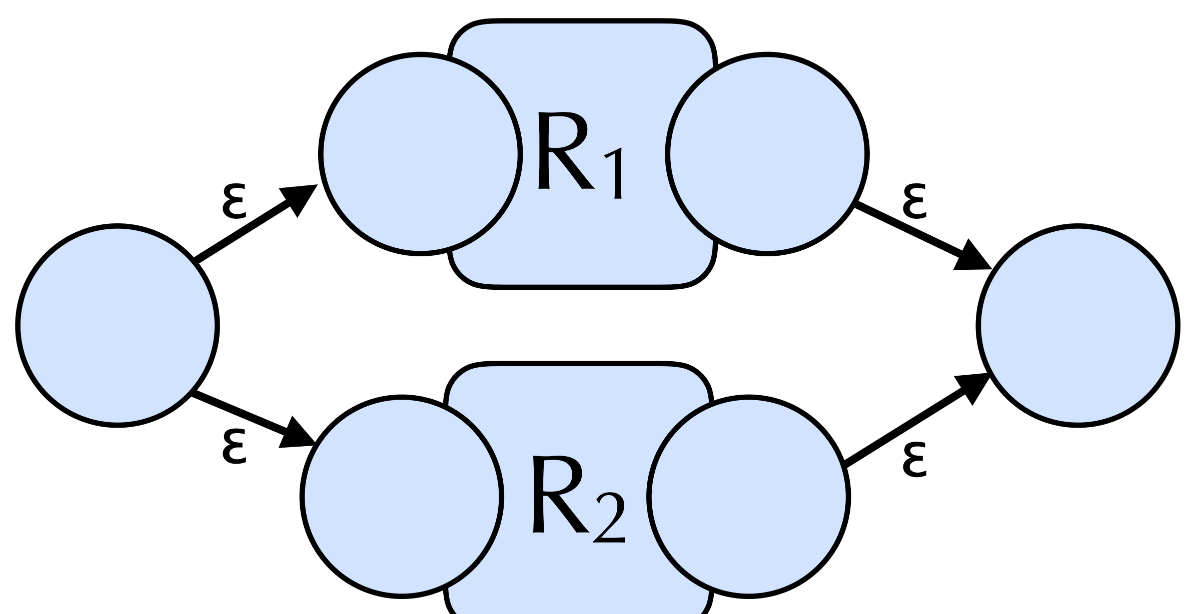
- Literal ‘a’



- Concatenation R_1R_2

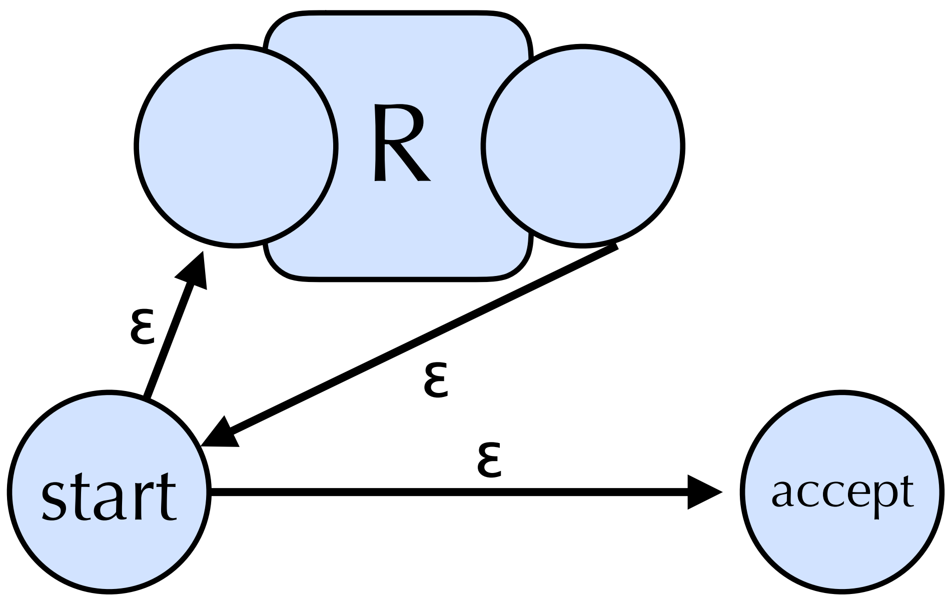


- Alternation $R_1 \mid R_2$



RE to NFA conversion

- Kleene star R*

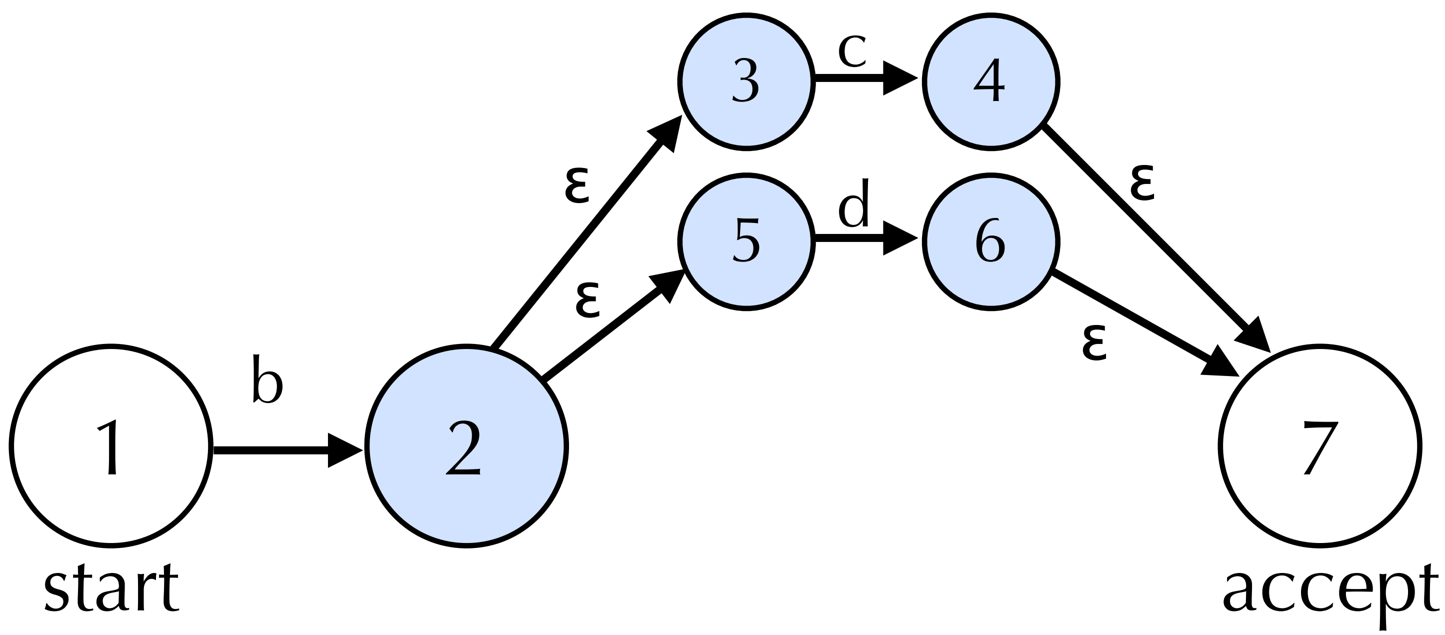


NFA to DFA conversion (intuition)

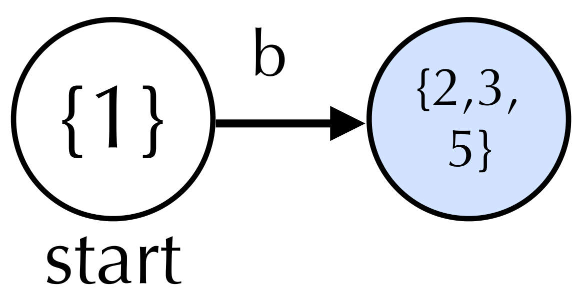
- The NFA of a regular expression R can be easily composed from NFAs of subexpressions of R.
- But executing an NFA under input strings is harder and less efficient than executing a DFA due to the nondeterminism.
- So, we convert NFAs to DFAs.
    - Basic idea: each state in DFA will represent a **set of states** of the NFA.

Example: NFA to DFA

NFA:

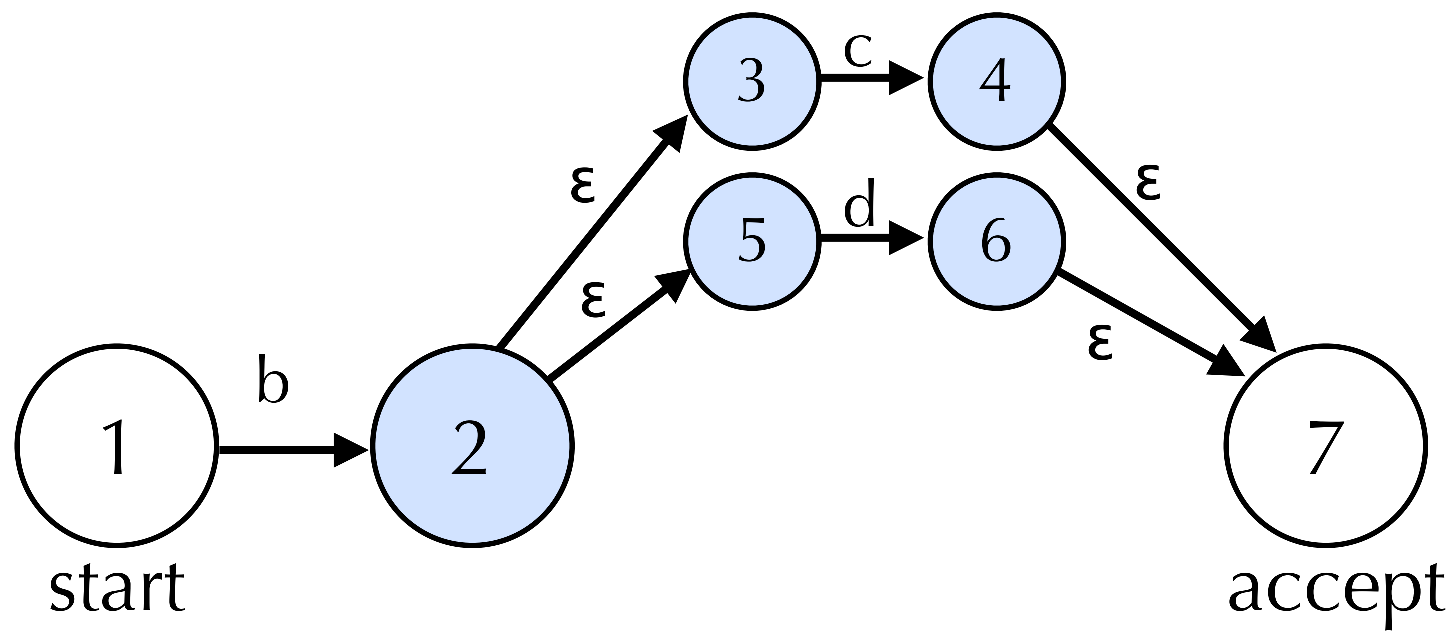


DFA:

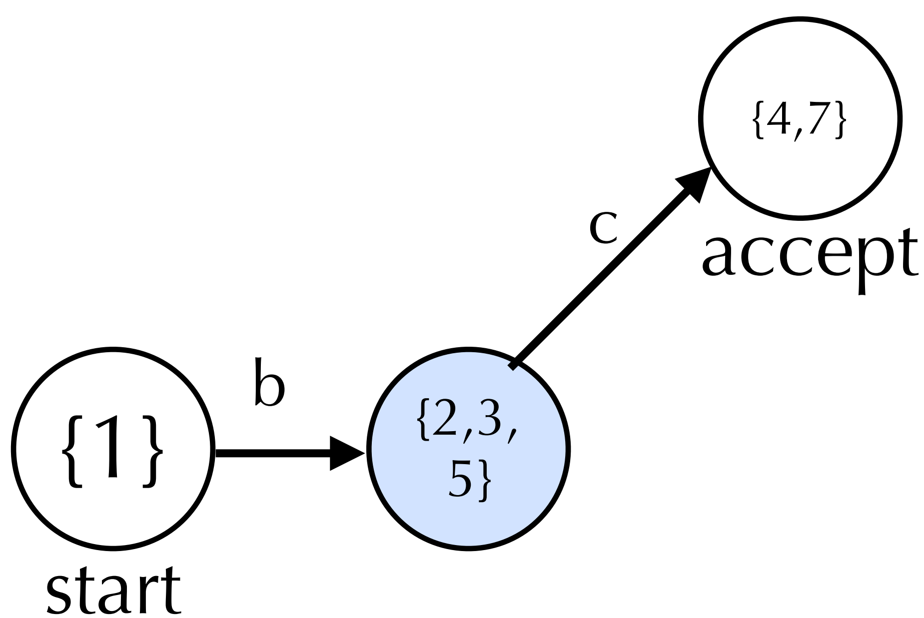


Example: NFA to DFA

NFA:

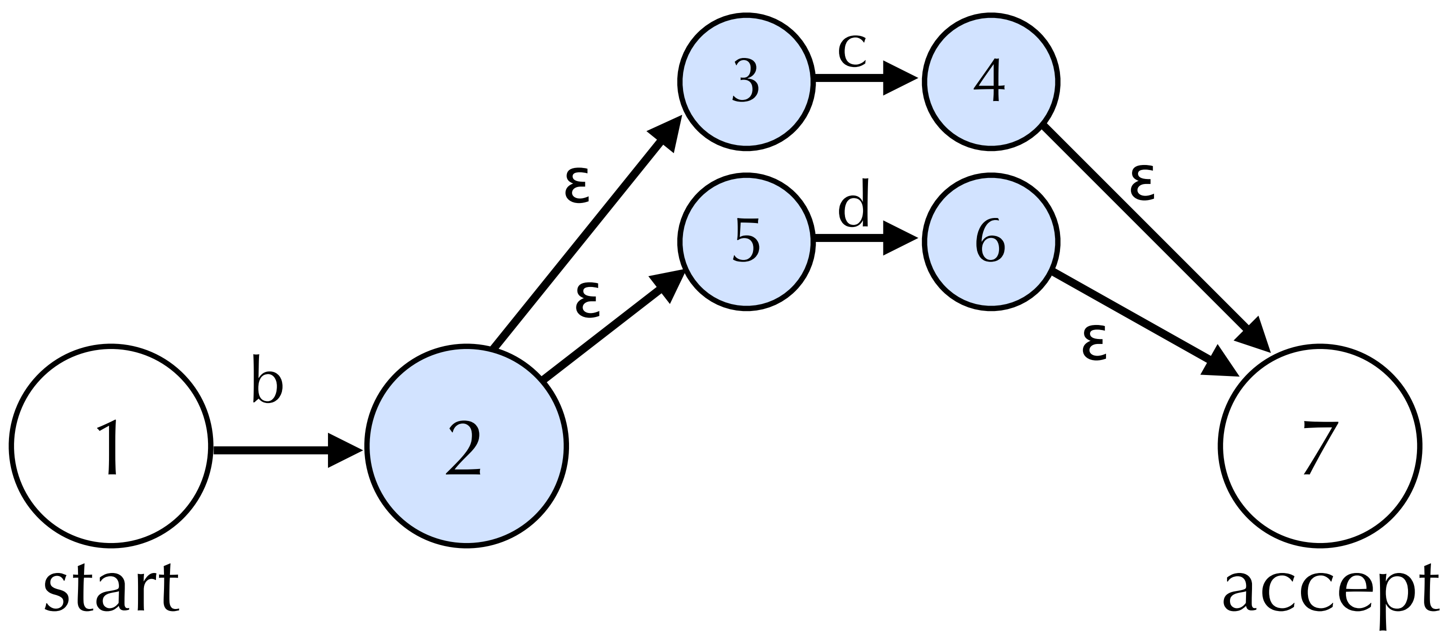


DFA:

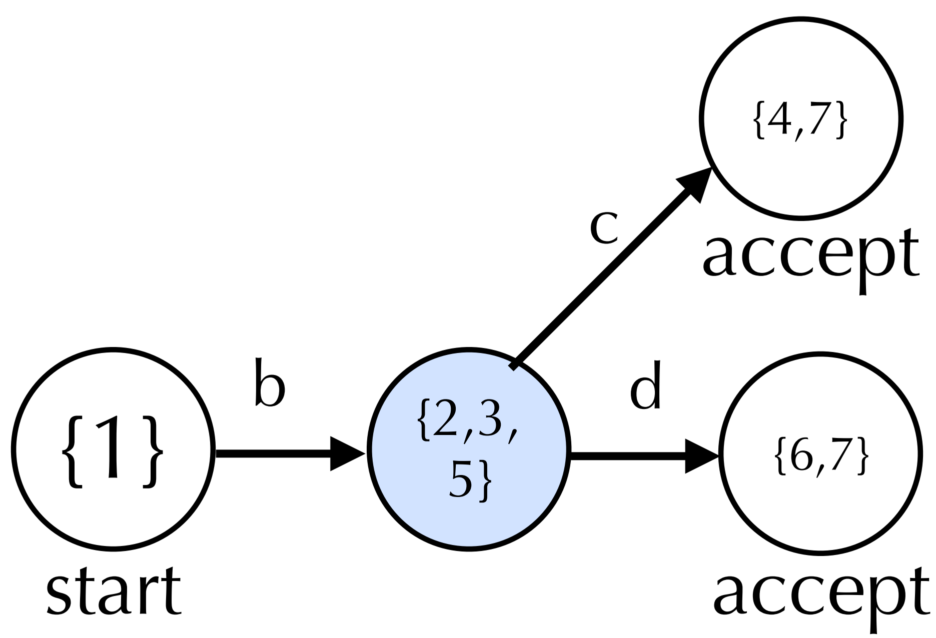


Example: NFA to DFA

NFA:



DFA:



Check that this DFA is, in fact, deterministic!

Lexical Analysis Summary

- Use a regular expression R_i to specify the set strings for each Token Type.
  - Example: [0-9]+ specifies the set of strings for NUM
- Construct the NFA formed by $(R_1 \mid R_2 \mid \ldots \mid R_n)$.
- Construct the DFA for this NFA.
- Produce the transition table for that DFA.
- Implement longest match.

Using a Lexer Generator

- The designer of a lexical analysis follows the first step of the previous slide.
- The remaining steps are automatically performed by the lexer generator!

A Lexer Generator in ML

- Provide regular expressions for token types in file `mllexeg.mll`
- Run lexer generator: `ocamllex mllexeg.mll`
- The lever generator produces the final transition table at file `mllexeg.ml`

Structure of ocamllex File

```
{ header }
let ident = regexp ...
rule entrypoint1 [arg1 ... argn] =
  parse regexp { action }
      | ...
      | regexp { action }
and entrypoint2 [arg1 ... argn] =
  parse ...
and ...
{ trailer }
```

- Header and trailer are arbitrary OCaml code, copied to the output file
- Can define abbreviations for common regular expressions
- Rules are turned into (mutually recursive) functions with `args1 ... argn lexbuf`
  - `lexbuf` is of type `Lexing.lexbuf`
  - Result of function is the result of ml code `action`

A hand-written Lexer

- See file `lexer.ml`