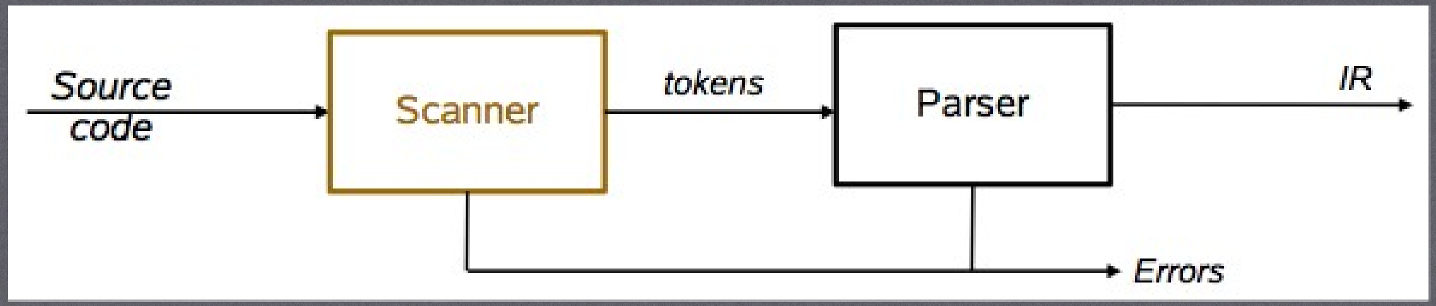


Compiling Techniques

Lecture 3: Introduction to Lexical Analysis

Christophe Dubach

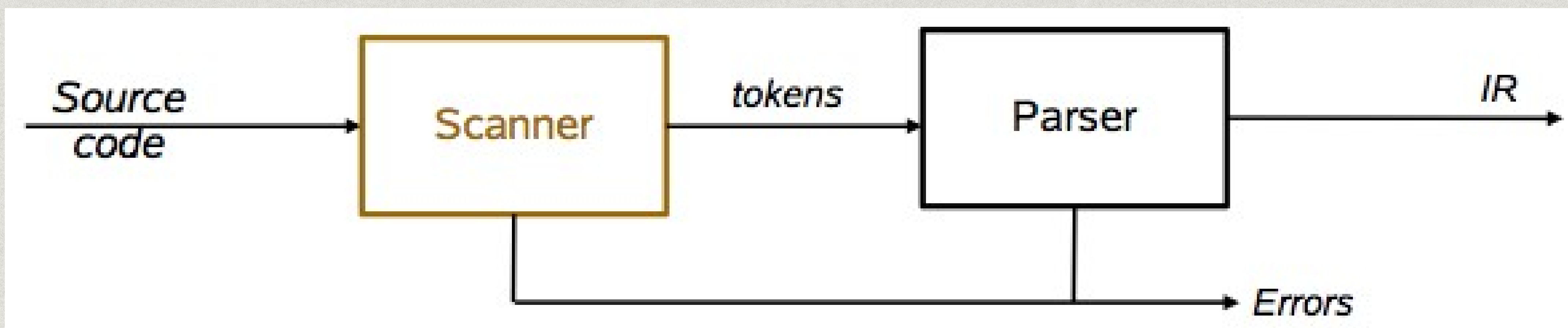
Overview

- Tutorials
- The Big Picture
- Regular Expressions
- DFAs and NFAs
- Automating Scanner Construction

Tutorials

- Monday 1:10pm - AT 4.07 (Christophe Dubach)
  Monday 1:10pm - AT 4.14A (Björn Franke)
  Thursday 1:10pm - AT 4.07 (Christophe Dubach)
- Tutorials start next week
- Group allocation on course website
- Online Forum: https://piazza.com/
  - action: enrol today!

Scanner

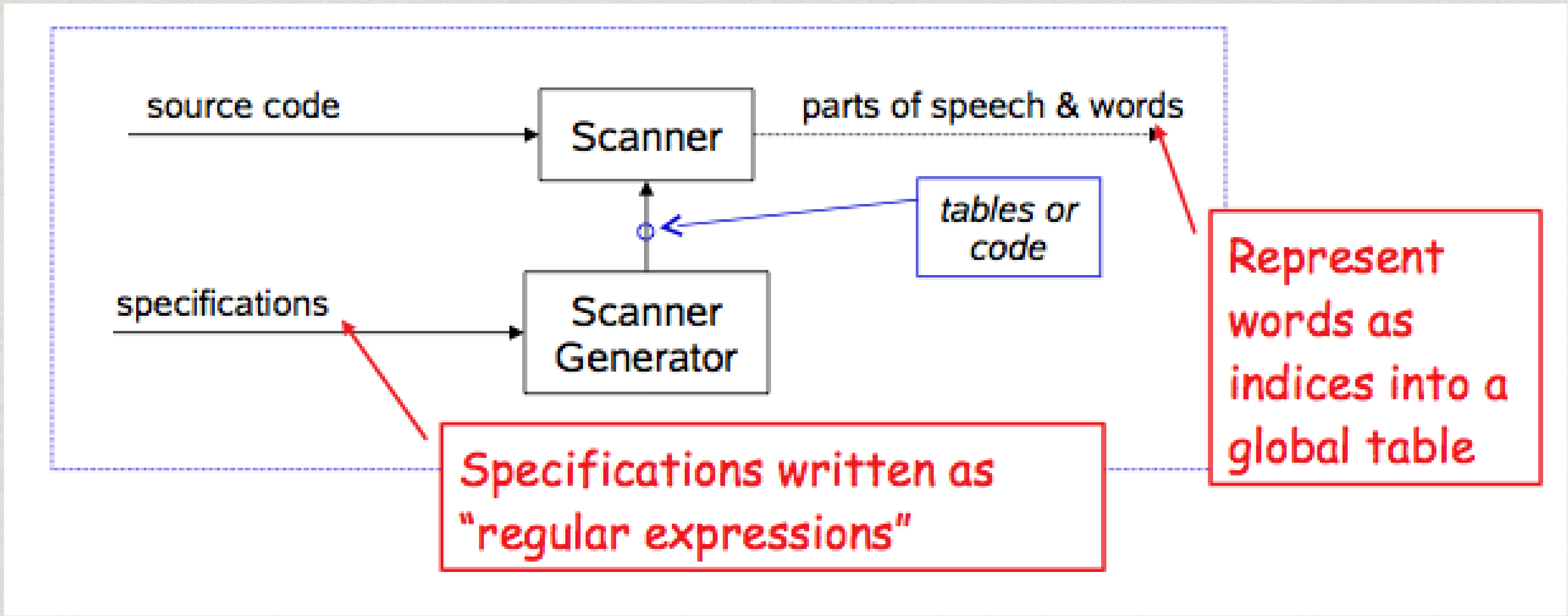


- Maps character stream into words—the basic unit of syntax
- Produces pairs — a word & its part of speech
  - x=x+y; becomes<id,x>=<id,x>+<id,y>;
  - word ≅ lexeme, part of speech ≅ token type
  - In casual speech, we call the pair a token
- Typical tokens include number, identifier, +, –, new, while, if
- Scanner eliminates white space (including comments)
- Speed is important

The Big Picture

- Why study lexical analysis?
  - We want to avoid writing scanners by hand
  - We want to harness the theory from other classes
- Goals:
  - To simplify specification & implementation of scanners
  - To understand the underlying techniques and technologies

The Big Picture



Regular Expressions

- Lexical patterns form a regular language
  - Any finite language is regular
  - Regular expressions (REs) describe regular languages
- Regular Expression (over alphabet **Σ**)
  - **ε** is a RE denoting the set {**ε**}
  - If **a** is in Σ, then **a** is a RE denoting {**a**}
  - If **x** and **y** are REs denoting L(**x**) and L(**y**) then
    - **x**|**y** is an RE denoting L(**x**) ∪ L(**y**)
    - **xy** is an RE denoting L(**x**)L(**y**)
    - **x*** is an RE denoting L(**x**)*
- Precedence is closure, then concatenation, then alternation

Set Operations

Operation	Definition
Union of L and M *Written $L \cup M$*	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
Concatenation of L and M *Written LM*	$LM = \{st \mid s \in L \text{ and } t \in M\}$
Kleene closure of L *Written L^**	$L^* = \cup_{0 \le i \le \infty} L^i$
Positive Closure of L *Written L^+*	$L^+ = \cup_{1 \le i \le \infty} L^i$

Example

Identifiers:

Letter → ($\underline{a}$|$\underline{b}$|$\underline{c}$| … |$\underline{z}$|$\underline{A}$|$\underline{B}$|$\underline{C}$| … |$\underline{Z}$)

Digit → ($\underline{0}$|$\underline{1}$|$\underline{2}$| … |$\underline{9}$)

Identifier → *Letter* (*Letter* | *Digit*)*

Numbers:

Integer → ($\underline{+}$|$\underline{-}$|ε) ($\underline{0}$| ($\underline{1}$|$\underline{2}$|$\underline{3}$| … |$\underline{9}$)(*Digit* *))

Decimal → *Integer* $\underline{.}$ *Digit* *

Real → (*Integer* | *Decimal*) $\underline{E}$ ($\underline{+}$|$\underline{-}$|ε) *Digit* *

Complex → $\underline{(}$ *Real* $\underline{,}$ *Real* $\underline{)}$

NUMBERS CAN GET MORE COMPLICATED!

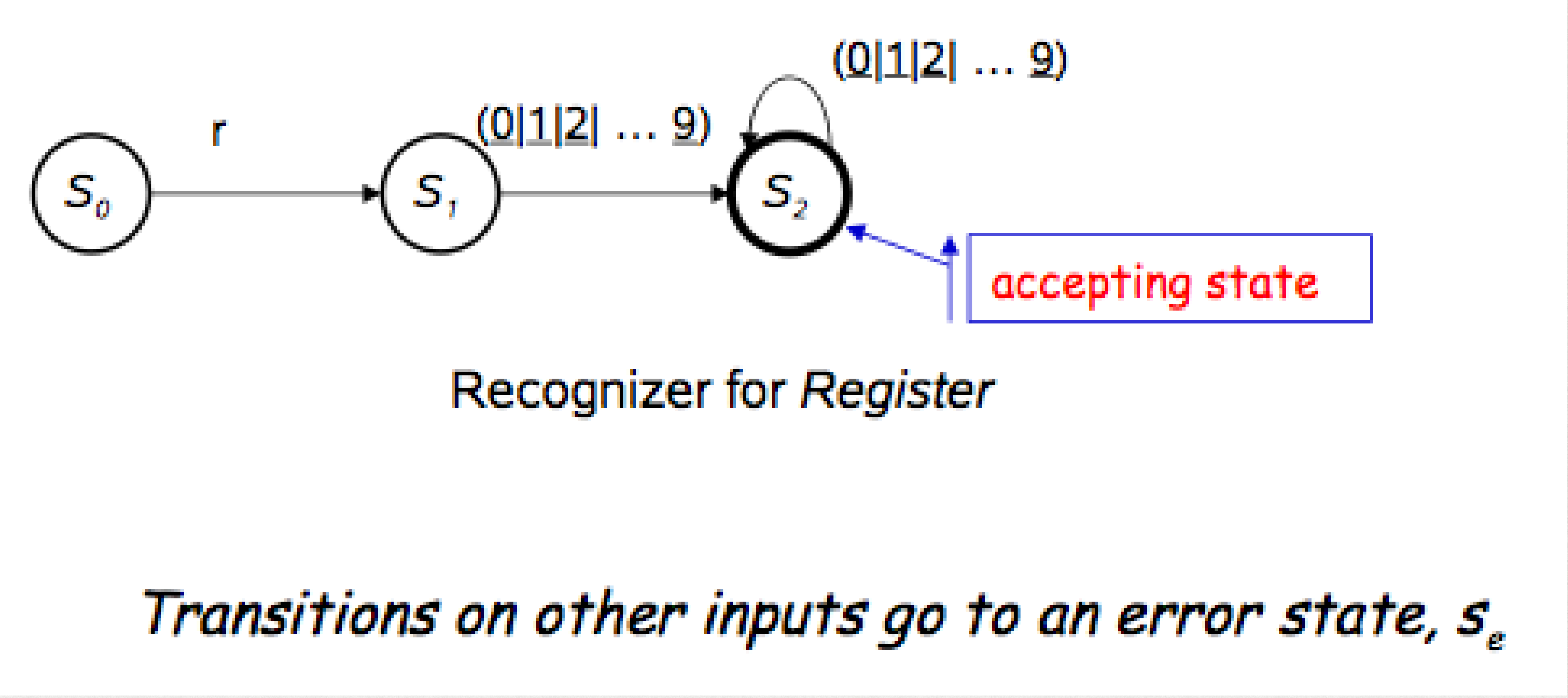
Scanners & Regular Expressions

- *Regular expressions can be used to specify the words to be translated to parts of speech by a lexical analyser*
- Using results from automata theory and theory of algorithms, we can automatically build recognisers from regular expressions
    - Some of you may have seen this construction for string pattern matching
- We study REs and associated theory to **automate scanner construction**!

Example

- Consider the problem of recognising register names
  - Register → r (0|1|2| … | 9) (0|1|2| … | 9)*
  - Allows registers of arbitrary number
  - Requires at least one digit

RE corresponds to a recogniser (or DFA, Deterministic Finite Automaton)



Example (continued)

- DFA operation
  - Start in state s0 & take transitions on each input character
  - DFA accepts a word **x** iff **x** leaves it in a final state (s2)
- So,
  - **r17** takes it through s0, s1, s2 and accepts
  - **r** takes it through s0, s1 and fails
  - **a** takes it straight to failure

Example (continued)

- To be useful, recogniser must turn into code

```
Char ← next character
State ← s0

while (Char ≠ EOF)
    State ← δ(State,Char)
    Char ← next character

if (State is a final state)
    then report success
    else report failure
```

Skeleton recognizer

δ	r	0,1,2,3,4, 5,6,7,8,9	*All others*
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Table encoding RE

Example (continued)

- To be useful, recogniser must turn into code

```
Char ← next character
State ← s0

while (Char ≠ EOF)
    State ← δ(State,Char)
    perform specified action
    Char ← next character

if (State is a final state)
    then report success
    else report failure
```

Skeleton recognizer

δ	r	0,1,2,3,4, 5,6,7,8,9	*All others*
s_0	s_1 *start*	s_e *error*	s_e *error*
s_1	s_e *error*	s_2 add	s_e *error*
s_2	s_e *error*	s_2 add	s_e *error*
s_e	s_e *error*	s_e *error*	s_e *error*

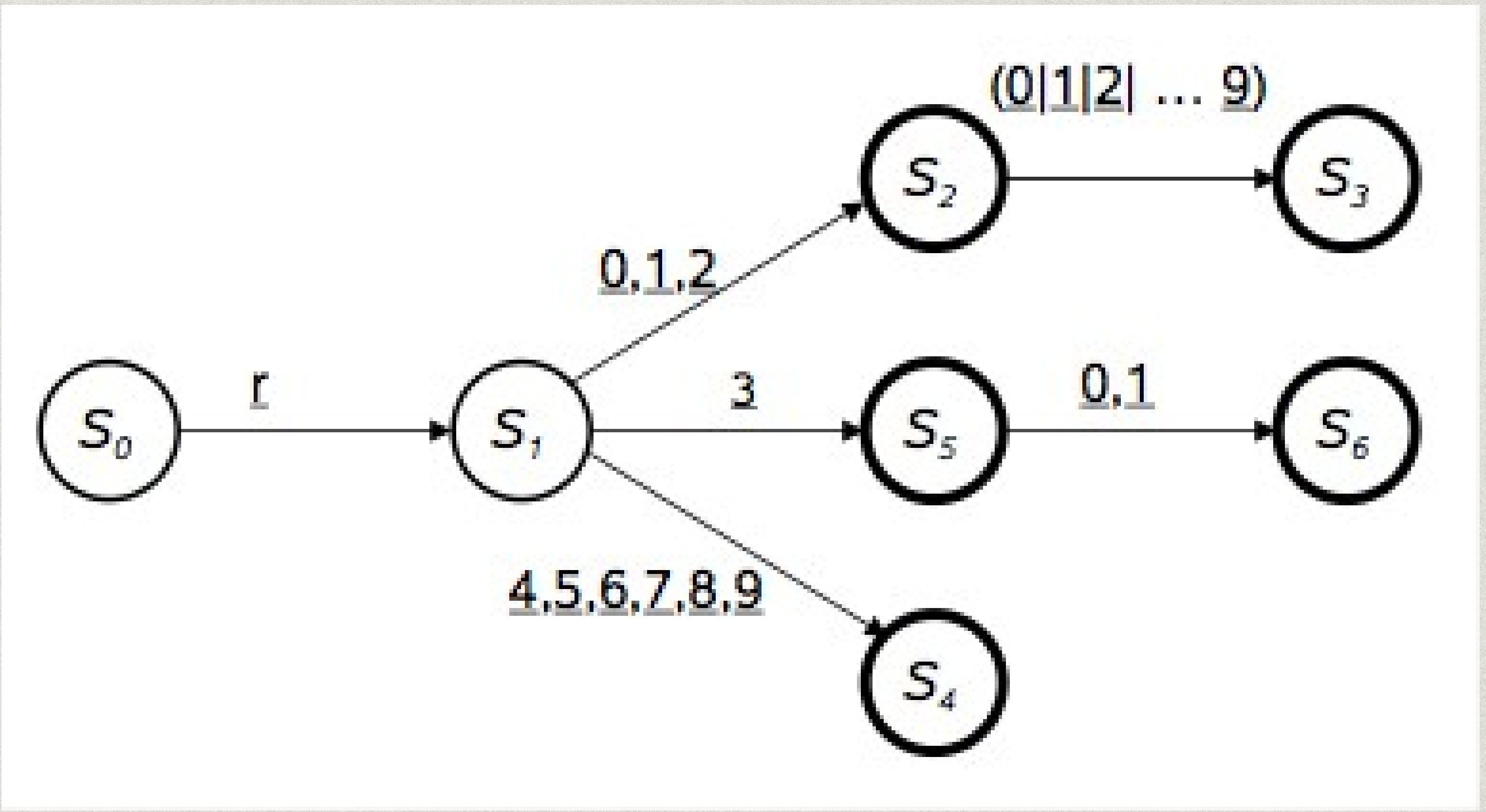
Table encoding RE

Extended Example

- What if we need a tighter specification?
- r Digit Digit* allows arbitrary numbers
  - Accepts r00000
  - Accepts r99999
  - What if we want to limit it to r0 through r31?
- Write a tighter regular expression
  - Register → r ((0|1|2) (Digit | ε) | (4|5|6|7|8|9) | (3|30|31))
  - Register → r0|r1|r2| ... |r31|r00|r01|r02| ... |r09
- Produces a more complex DFA
  - Has more states
  - Same cost per transition
  - Same basic implementation

Extended Example (continued)

- The DFA for Register → r ((0|1|2) (Digit | ε) **|** (4|5|6|7|8|9) **|** (3|30|31))



- Accepts a more constrained set of registers
- Same set of actions, more states

Extended Example (continued)

δ	r	0,1	2	3	4-9	All others
s_0	s_1	s_e	s_e	s_e	s_e	s_e
s_1	s_e	s_2	s_2	s_5	s_4	s_e
s_2	s_e	s_3	s_3	s_3	s_3	s_e
s_3	s_e	s_e	s_e	s_e	s_e	s_e
s_4	s_e	s_e	s_e	s_e	s_e	s_e
s_5	s_e	s_6	s_e	s_e	s_e	s_e
s_6	s_e	s_e	s_e	s_e	s_e	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e

Table encoding RE for the tighter register specification

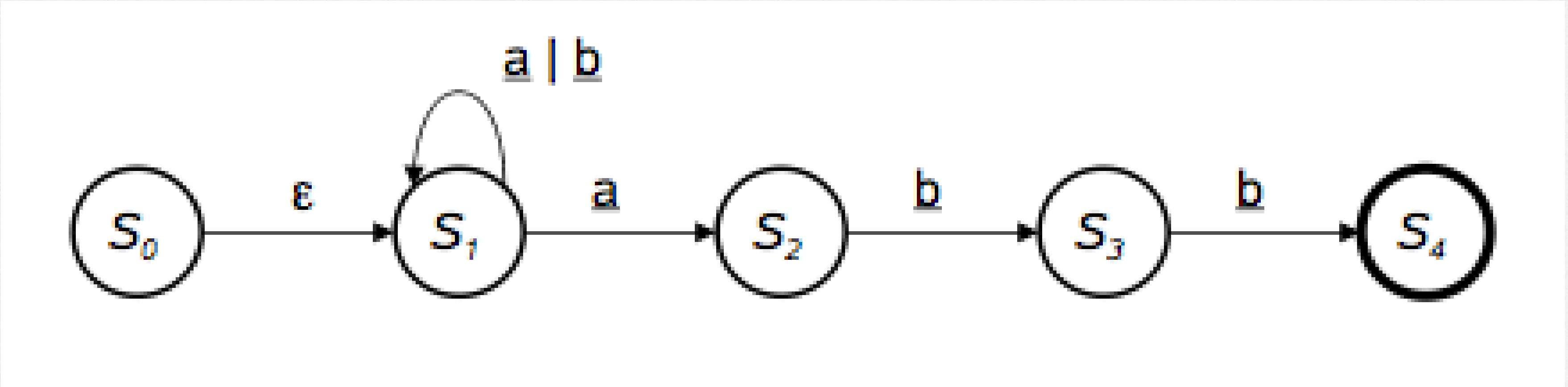
RUNS IN THE SAME SKELETON RECOGNISER!

Goal

- We will show how to construct a finite state automaton to recognise any RE
- Overview:
  - Direct construction of a **nondeterministic finite automaton (NFA)** to recognise a given RE
    - Requires ε-transitions to combine regular subexpressions
  - Construct a **deterministic finite automaton (DFA)** to simulate the NFA
    - Use a set-of-states construction
  - **Minimise** the number of states
    - Hopcroft state minimisation algorithm
  - **Generate** the scanner code
    - Additional specifications needed for details

Non-Deterministic Finite Automata

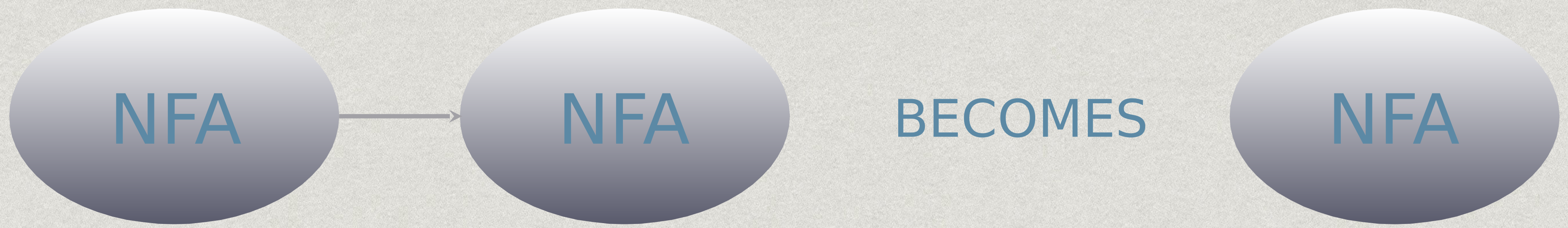
- Each RE corresponds to a deterministic finite automaton (DFA)
- May be hard to directly construct the right DFA
  - What about an RE such as **(a|b)*abb**?



- This is a little different
  - S0 has a transition on **ε**
  - S1 has two transitions on **a**
  - This is a **non-deterministic finite automaton (NFA)**

Non-Deterministic Finite Automata

- An NFA accepts a string **x** iff ∃ a path though the transition graph from s0 to a final state such that the edge labels spell **x**
- Transitions on **ε** consume no input
- To “run” the NFA, start in s0 and guess the right transition at each step
  - Always guess correctly
  - If some sequence of correct guesses accepts **x** then accept
- Why study NFAs?
  - They are the key to automating the RE→DFA construction
  - We can paste together NFAs with ε-transitions



Relationship between NFAs and DFAs

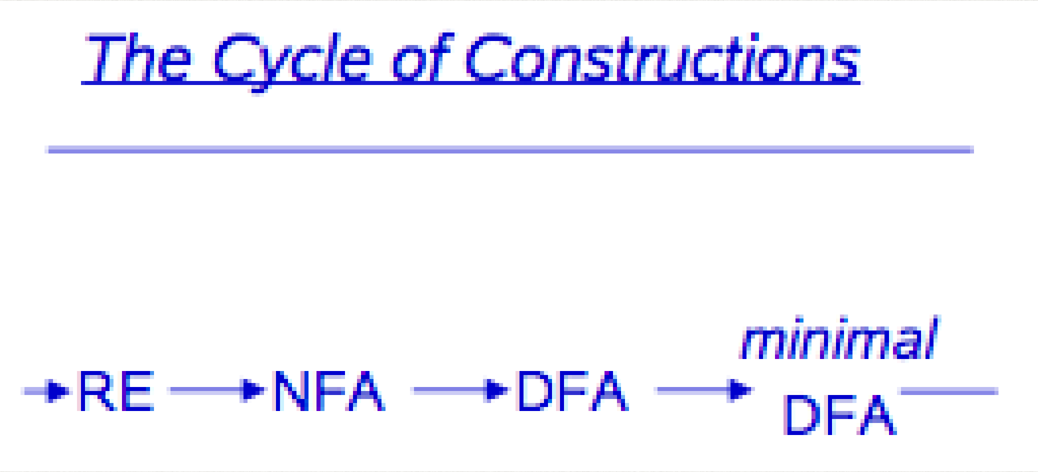
- DFA is a special case of an NFA
  - DFA has no ε transitions
  - DFA’s transition function is single-valued
  - Same rules will work
- DFA can be simulated with an NFA
  - Obviously
- NFA can be simulated with a DFA
  - Less obvious
  - Simulate sets of possible states
  - Possible exponential blowup in the state space
  - Still, one state per character in the input stream

Automating Scanner Construction

- To convert a specification into code:
  1. Write down the RE for the input language
  2. Build a big NFA
  3. Build the DFA that simulates the NFA
  4. Systematically shrink the DFA
  5. Turn it into code
- Scanner generators
  - Lex and Flex work along these lines
  - Algorithms are well-known and well-understood
  - Key issue is interface to parser (define all parts of speech)
  - You could build one in a weekend!

Automating Scanner Construction

- RE→ NFA (Thompson's construction)
    - Build an NFA for each term
    - Combine them with ε-moves
- NFA → DFA (subset construction)
    - Build the simulation
- DFA → Minimal DFA
    - Hopcroft's algorithm
- DFA →RE (Not part of the scanner construction)
    - All pairs, all paths problem



Preview

- Constructing a Scanner from Regular Expressions