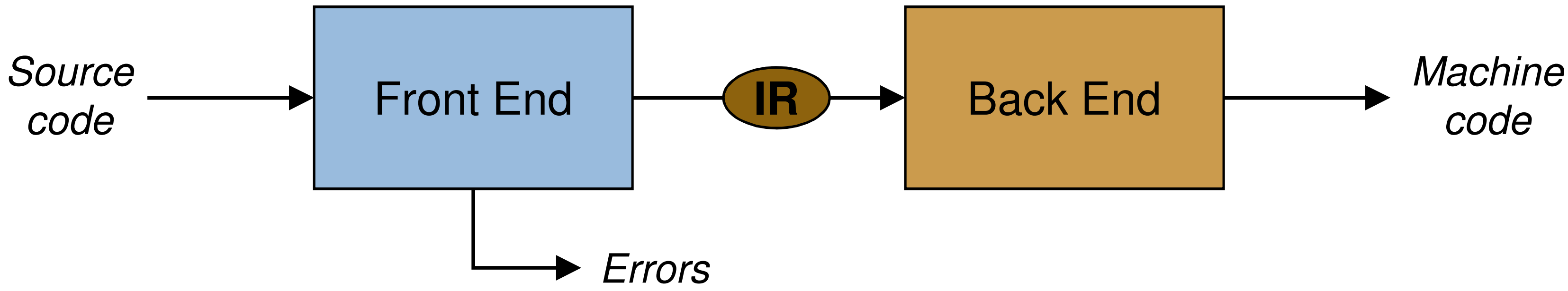
Compilers

Lecture 3

Lexical analysis

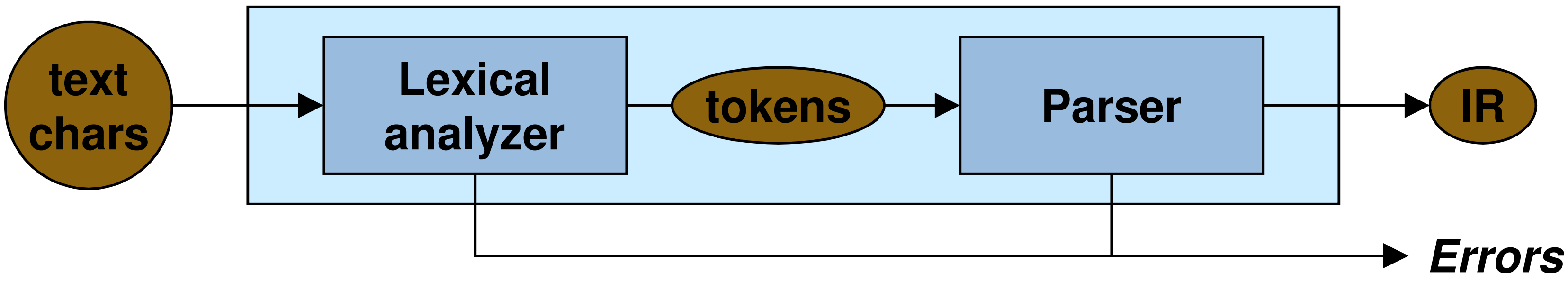
Yannis Smaragdakis, U. Athens
(original slides by Sam Guyer@Tufts)

Big picture



- Front end responsibilities
  - Check that the input program is legal
    - Check syntax and semantics
    - Emit meaningful error messages
  - Build IR of the code for the rest of the compiler

Front end design



- Two part design
  - Scanner (a.k.a. lexer)
    - Reads in characters
    - Classifies sequences into words or ***tokens***
  - Parser
    - Checks sequence of tokens against grammar
    - Creates a representation of the program (AST)

Lexical analysis

- The input is just a sequence of characters.
  Example:

```
if (i == j)
  z = 0;
else
  z = 1;
```

- More accurately, the input is:
  `\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;`

- **Goal**: Partition input string into substrings
  And classify them according to their role

Scanner

- Responsibilities
  - Read in characters
  - Produce a stream of tokens

i	f		(	i	=	=	j	)	{	…

<key,if>	<(>	<id,i>	<op,==>	<id,j>	…

  - Token has a type and a value

Hand-coded scanner

- Explicit test for each token
  - Read in a character at a time
  - Example: recognizing keyword “if”

Hand-coded scanner

- What about other tokens?

  Example: “if” is a keyword, “if0” is an identifier

```
c = readchar();
if (c != `i') { other tokens… }
else {
  c = readchar();
  if (c != `f') { other tokens… }
  else {
    c = readchar();
    if (c not alpha-numeric) {
      putback(c);
      return IF_TOKEN; }
    while (c alpha-numeric) { build identifier }
```

Hand-coded scanner

Problems:

- Many different kinds of tokens
  - Fixed strings (keywords)
  - Special character sequences (operators)
  - Tokens defined by rules (identifiers, numbers)

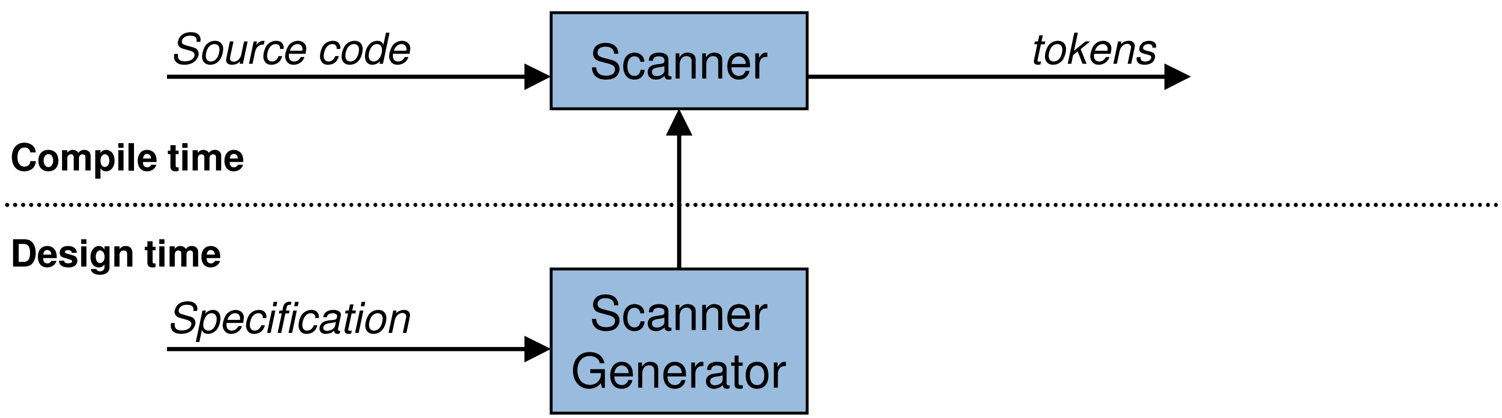
- Tokens overlap
  - “if” and “if0” example
  - “=“ and “==“

- Coding this by hand is too painful!

  Getting it right is a serious concern

Scanner construction

- **Goal**: automate process
  - Avoid writing scanners by hand
  - Leverage the underlying theory of languages



Outline

Problems we need to solve:

- Scanner description
  - How to describe parts of the input language
- The scanning mechanism
  - How to break input string into tokens
- Scanner generator
  - How to translate from (1) to (2)
- Ambiguities
  - The need for ***lookahead***

Problem 1: Describing the scanner

- We want a high-level language **D** that
  1. Describes lexical components, and
  2. Maps them to tokens (determines type)
  3. ***But*** doesn't describe the scanner algorithm itself !

- Part 3 is important
  - Allows focusing on ***what***, not on ***how***
  - Therefore, **D** is sometimes called a ***specification language***, not a programming language

- Part 2 is easy, so let's focus on Parts 1 and 3

Specifying tokens

- Many ways to specify them
- **Regular expressions** are the most popular
  - REs are a way to specify *sets of strings*
  - Examples:
    - <u>a</u> – denotes the set {“a”}
    - <u>a|b</u> – denotes the set {“a”, “b”}
    - <u>ab</u> – denotes the set {“ab”}
    - <u>ab*</u> – denotes the set {“a”, “ab”, “abb”, “abbb”, … }

May specify sets of infinite size

- Why regular expressions?
  - Easy to understand
  - Strong underlying theory
  - Very efficient implementation

Formal languages

- ***Def***: a language is a set of strings
  - Alphabet Σ : the character set
  - Language is a set of strings over alphabet
- Each regular expression denotes a language
  - If ***A*** is a regular expression, then ***L(A)*** is the set of strings denoted by ***A***
  - Examples: given Σ = {‘a’, ‘b’}
    - A = a   L(A) = {“a”}
    - A = a|b   L(A) = {“a”, “b”}
    - A = ab   L(A) = {“ab”}
    - A = ab*   L(A) = {“a”, “ab”, “abb”, “abbb”, … }

Building REs

- Regular expressions over Σ

- Atomic REs
  - ε is an RE denoting empty set
  - if <u>a</u> is in Σ, then a is an RE for {<u>a</u>}

- Compound REs
  - if *x* and *y* are REs then:
    - *xy* is an RE for *L(x)L(y)* — *Concatentation*
    - *x|y* is an RE for $L(x) \cup L(y)$ — *Alternation*
    - *x** is an RE for *L(x)** — *Kleene closure*

Outline

Problems we need to solve:

- Scanner specification language ***DONE***
  - How to describe parts of the input language
- The scanning mechanism
  - How to break input string into tokens
- Scanner generator
  - How to translate from (1) to (2)
- Ambiguities
  - The need for ***lookahead***

Overview of scanning

- How do we recognize strings in the language?

  Every RE has an equivalent finite state automaton that recognizes its language

  (Often more than one)

  - **Idea**: scanner simulates the automaton
    - Read characters
    - Transition automaton
    - Return a token if automaton accepts the string

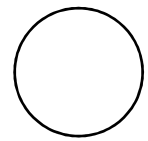
Finite Automata

- Regular expressions = specification
- Finite automata = implementation

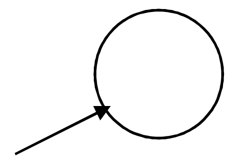
- A finite automaton consists of
  - An input alphabet Σ
  - A set of states S
  - A start state n
  - A set of accepting states $F \subseteq S$
  - A set of transitions state $\rightarrow^{\text{input}}$ state

Finite Automata State Graphs

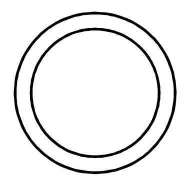
- A state



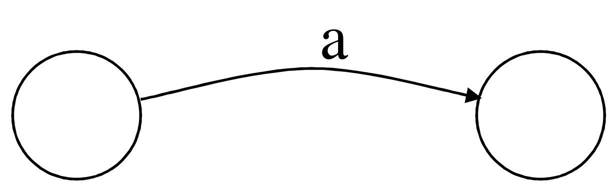
- *The start state*



- *An accepting state*



- *A transition*

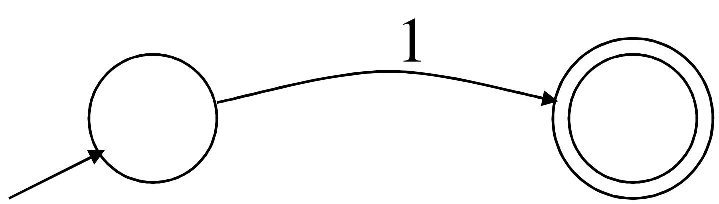


FA Example

- Transition $s_1 \rightarrow^a s_2$
- Is read *In state s_1 on input "a" go to state s_2*

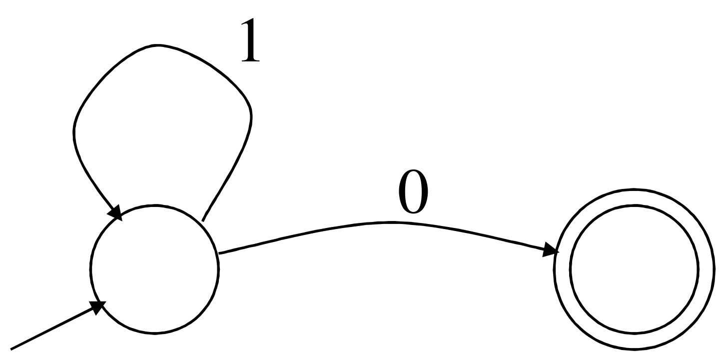
- FA accepts a string if we can follow transitions labeled with the characters in the string from the start to an accepting state
  - What if we run out of characters?

- A finite automaton that accepts only "1"



Another Simple Example

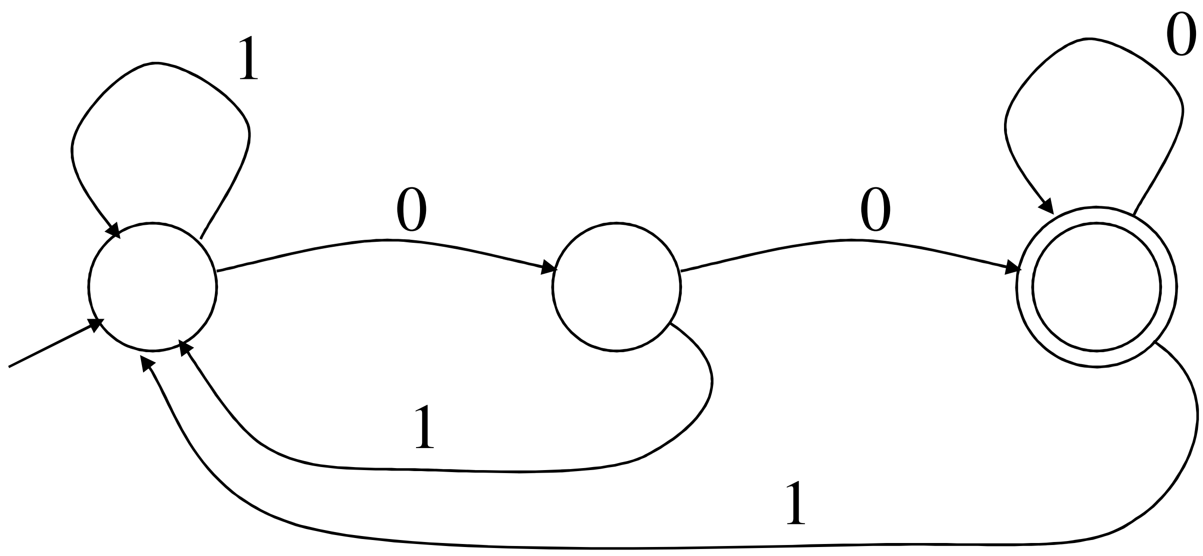
- FA accepts any number of 1's followed by a single 0
- Alphabet: {0,1}



- Check that "1110" is accepted but "1101…" is not

And Another Example

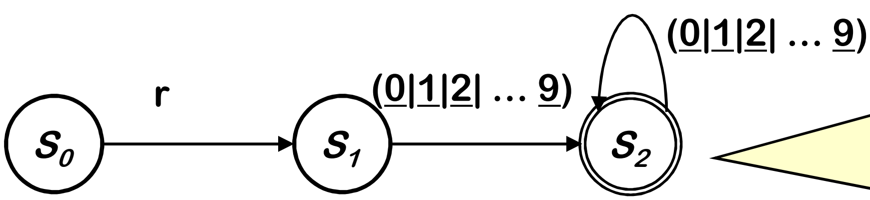
- Alphabet {0,1}
- What language does this recognize?



“Realistic” example

- Recognizing machine register names
  - Typically “r” followed by register number (how many?)

$$\mathit{Register} \rightarrow \underline{r}\ (\underline{0}|\underline{1}|\underline{2}|\ \ldots\ |\ \underline{9})\ \ (\underline{0}|\underline{1}|\underline{2}|\ \ldots\ |\ \underline{9})^{*}$$



Recognizer for *Register*

$\underline{r17}$ takes it through s_0, s_1, s_2 and accepts

$\underline{r}$ takes it through s_0, s_1 and fails

$\underline{a}$ takes it straight to s_e

REs and DFAs

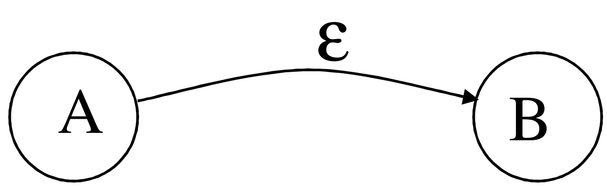
- **Key idea**:
  - Every regular expression has an equivalent DFA that accepts only strings in the language

- **Problem**:
  - How do we construct the DFA for an arbitrary regular expression?
  - Not always easy

Example

- What is the FA for a(a|ε)b?

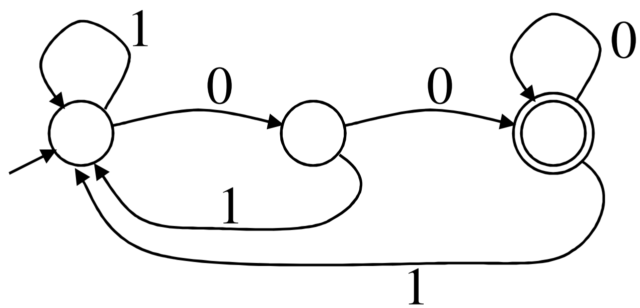
- Need ε moves



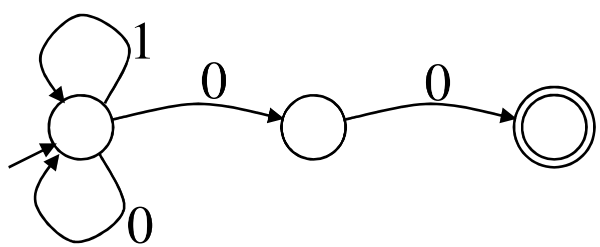
- Transition A to B without consuming input!

Another example

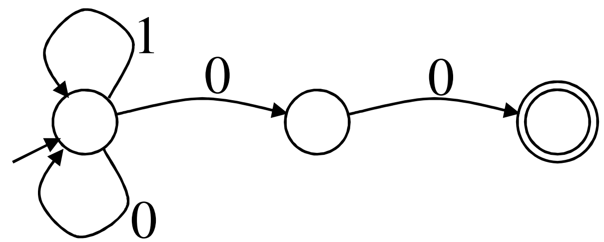
- Remember this DFA?



- We can simplify it as follows:



A different kind of automaton



- Accepts the same language

  Actually, it's easier to understand!

- What's different about it?
  - Two different transitions on '0'
  - This is a ***non-deterministic finite automaton***

DFAs and NFAs

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves

- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves

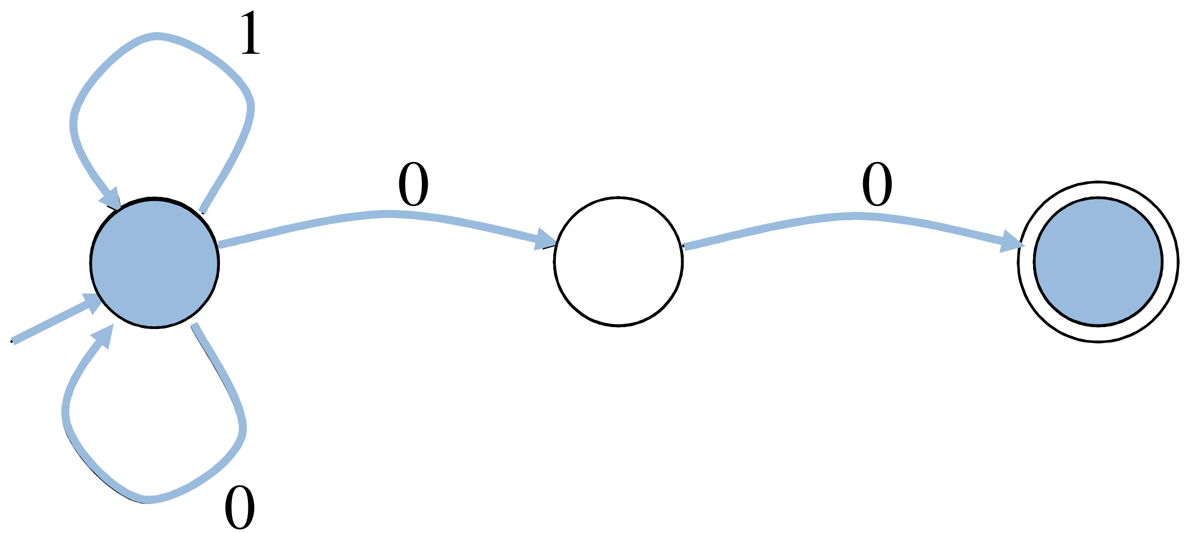
Execution of Finite Automata

- DFA can take only one path through the state graph
  - Completely determined by input

- NFAs can choose
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take

Acceptance of NFAs

- An NFA can get into multiple states



- *Input*: 1 0 0
- *Rule*: NFA accepts if it <u>can</u> get in a final state

Non-deterministic finite automata

- An NFA accepts a string *x* iff ∃ a path through the transition graph from s_0 to a final state such that the edge labels spell *x*

  (Transitions on ε consume no input)

- To "run" the NFA, start in s_0 and ***guess*** the right transition at each step
  - Always guess correctly
  - If some sequence of correct guesses accepts x then accept

Why do we care about NFAs?

- Simpler, smaller than DFAs
- More importantly:
  - Need them to support all RE capabilities
  - Systematic conversion from REs to NFAs
  - Need ε transitions to connect RE parts
- **Problem**: how to implement NFAs?
  - How do we guess the right transition?

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 *(obvious)*

- 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

Automatic 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!

Scanner construction

[0] Define tokens as regular expressions

[1] Construct NFA for all REs

- Connect REs with ε transitions
- ***Thompson's construction***

[2] Convert NFA into a DFA

- DFA is a simulation of NFA
- Possibly much larger than NFA
- ***Subset construction***

[3] Minimize the DFA

- ***Hopcroft's algorithm***

[4] Generate implementation

[1] Thompson's construction

- ***Goal:***

  Systematically convert regular expressions for our language into a finite state automaton

- ***Key idea***:
  - FA "pattern" for each RE operator
  - Start with atomic REs, build up a big NFA

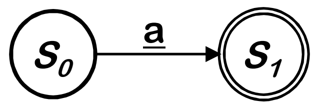
- Idea due to Ken Thompson in 1968

Thompson's construction

By induction on RE structure

- ***Base case:***

  Construct FA that recognizes atomic regular expressions:



- ***Induction:***

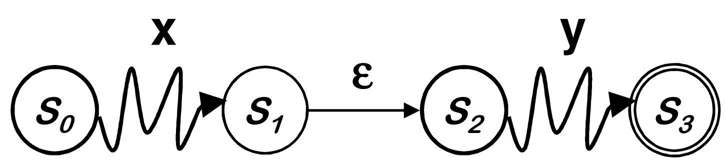
  Given FAs for two regular expressions, **x** and **y**, build a new FA that recognizes:

  - **xy**
  - **x|y**
  - **x***

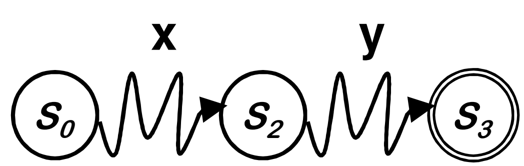
Thompson's construction

- Given:

- Build **xy**

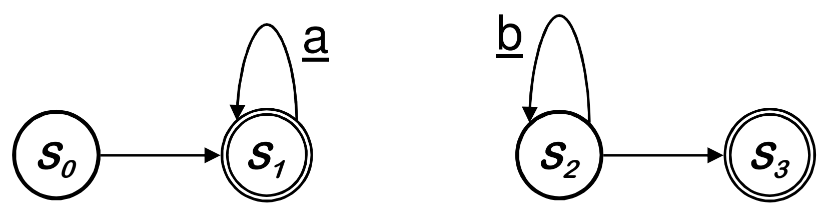


- Why can't we do this?

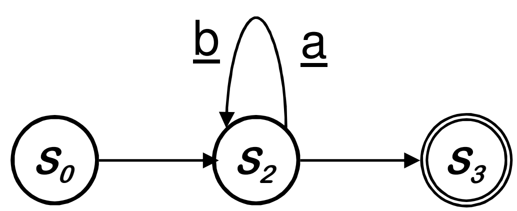


Need for ε transitions

- What if **x** and **y** look like this:

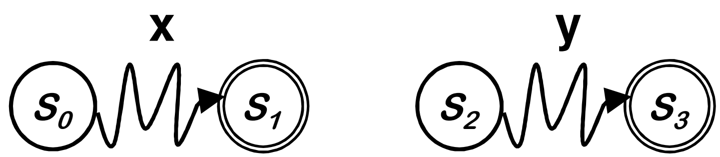


- Then **xy** ends up like this:

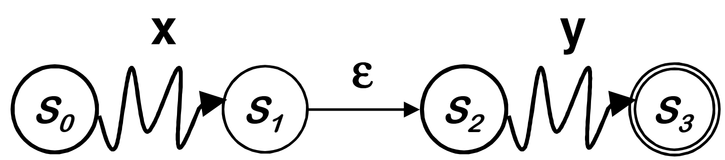


Thompson's construction

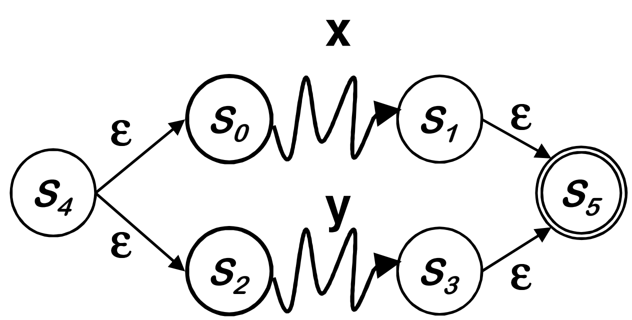
- Given:



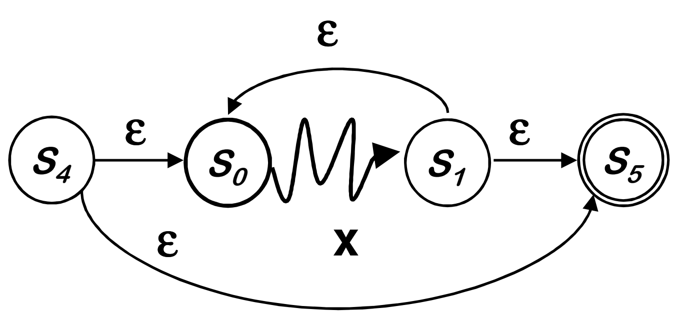
- **xy**



- **x|y**



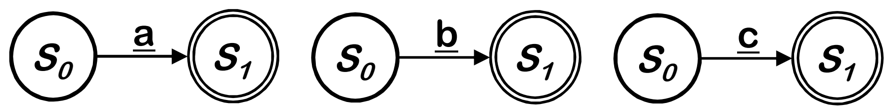
- **x***



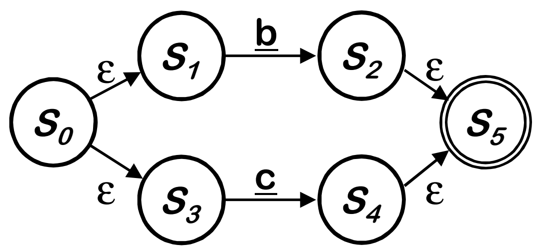
Example

Regular expression: a (b | c)*

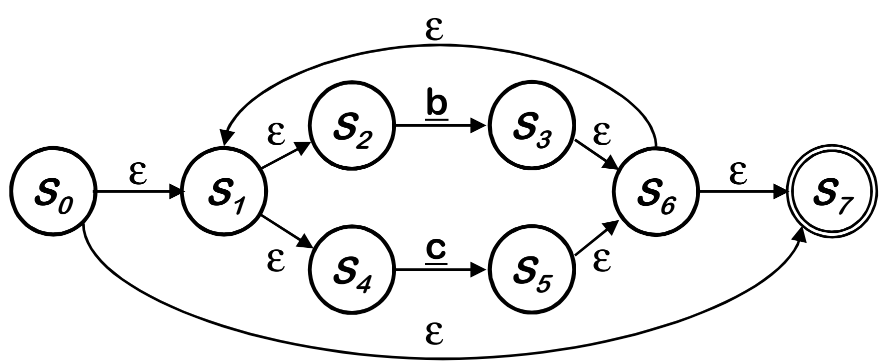
- a, b, & c



- b | c

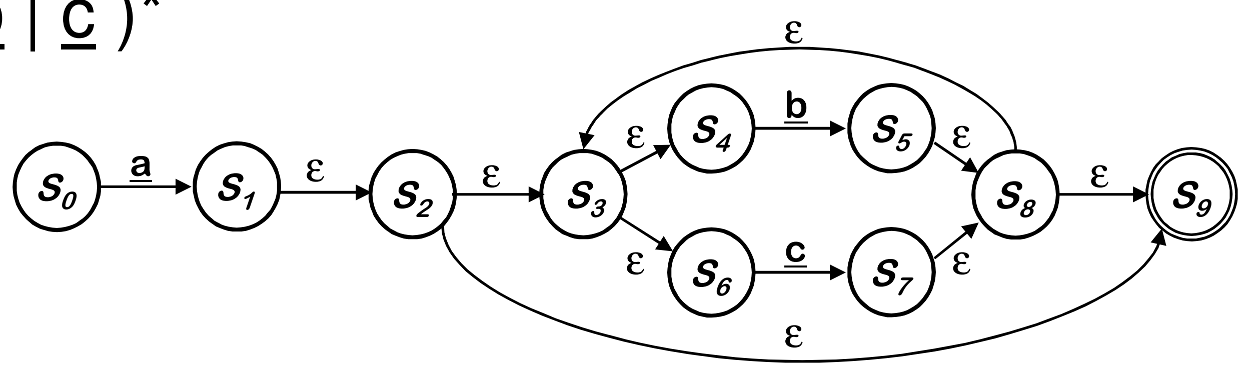


- (b | c)*

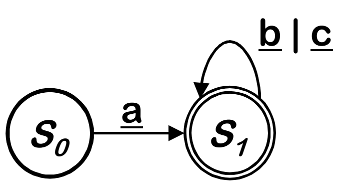


Example

- a (b | c)*



- Note: a human could design something simpler…
  - Like what?



Problem

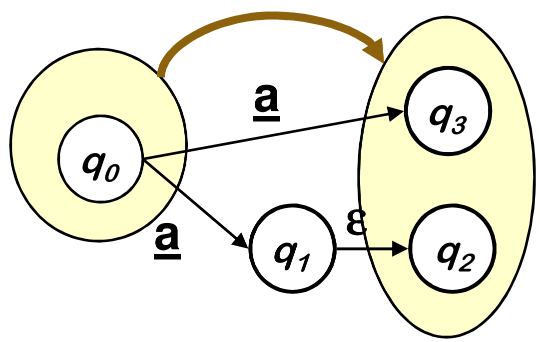
- How to implement NFA scanner code?
  - Will the table-driven scheme work?
  - Non-determinism is a problem
  - Explore all possible paths?

- Observation:

  We can build a DFA that simulates the NFA
  - Accepts the same language
  - Explores all paths simultaneously

[2] NFA to DFA

- Subset construction algorithm
  - **Intuition**: each DFA state represents the *possible* states reachable after one input in the NFA



State in DFA = set of states from NFA

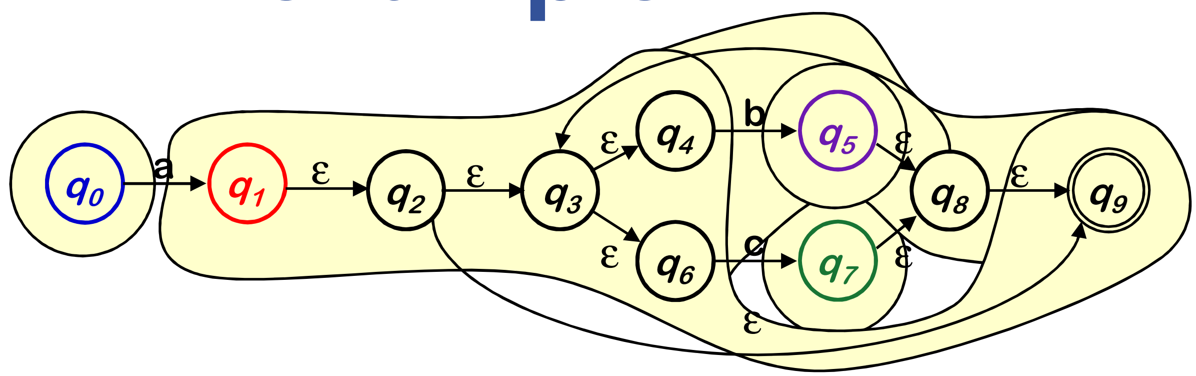
$s_1 = \{ q_0 \}$

$s_2 = \{ q_2, q_3 \}$

  - Two key functions
    - **next(s_i, a)** – the set of states reachable from s_i on a
    - **ε-closure(s_i)** – the set of states reachable from s_i on ε
  - DFA transition function
    - Edge labeled a from state **s_i** to state **ε-closure(next(s_i, a))**

NFA to DFA example

<u>a</u> (<u>b</u> | <u>c</u>)* :

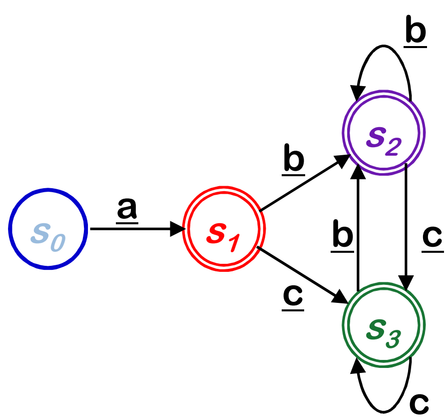


Subsets S *(DFA states)*	NFA states	ε-closure(next(s,α)) <u>a</u>	<u>b</u>	<u>c</u>
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	*none*	*none*
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	*none*	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$	*none*	*(also s_2)*	*(also s_3)*
s_3	$q_7, q_8, q_9, q_3, q_4, q_6$	*none*	*(also s_2)*	*(also s_3)*

Accepting states

NFA to DFA example

- Convert each subset in S into a state:



δ	a	b	c
s_0	s_1	-	-
s_1	-	s_2	s_3
s_2	-	s_2	s_3
s_3	-	s_2	s_3

- All transitions are deterministic
- Smaller than NFA, but still bigger than necessary

Subset construction

- Algorithm

  Build a set of subsets of NFA states

```
s0 ← ε-closure(initial)
S ← { s0 }
worklist ← { s0 }
while ( worklist is not empty )
   remove s from worklist
   for each α ∈ Σ
      t ← ε-closure( next(s,α))
      if ( t ∉ S ) then
         add t to S
         add t to worklist
         add transition (s, α, t)
```

Start with the ε-closure over the initial state

Initialize the worklist to this one subset

While there are more subsets on the worklist, remove the next subset

Apply each input symbol to the set of NFA states to produce a new set.

If we haven't seen that subset before, add it to S and the worklist, and record the set-to-set transition

Does it work?

- Does the algorithm halt?
  - S contains no duplicate subsets
  - $2^{|NFA|}$ is finite
  - Main loop adds to S, but does not remove

    *It is a **monotone** function*
- S contains all the reachable NFA states

  Tries all input symbols, builds all NFA configurations
- **Note**: important class of compiler algorithms
  - ***Fixpoint*** computation
  - Monotonic update function
  - Convergence is guaranteed

[3] DFA minimization

- Hopcroft's algorithm
  - Discover sets of *equivalent* states in DFA
  - Represent each set with a single state

- When would two states in the DFA be equivalent?

- Two states are equivalent ***iff***:
  - For all input symbols, transitions lead to equivalent states

⇨ This is the key to the algorithm

DFA minimization

- A ***partition*** P of the states S
  - Each $s \in S$ is in exactly one set $p_i \in P$
  - Idea:

    If two states s and t transition to different partitions, then they must be in different partitions

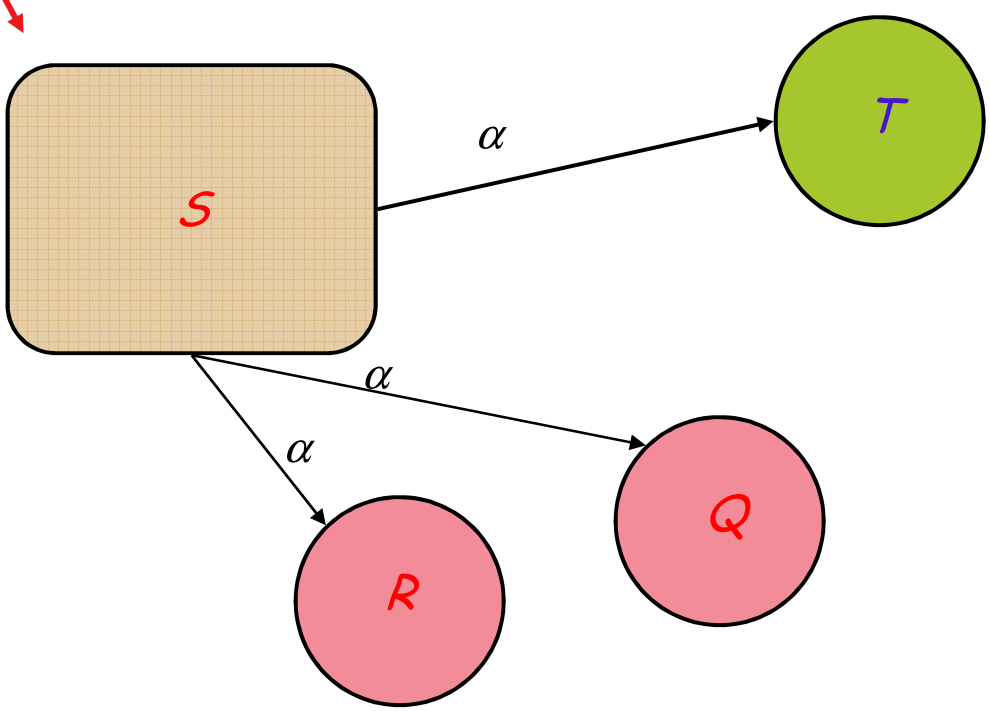
- Algorithm:

  Iteratively partition the DFA's states

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

Splitting S around α

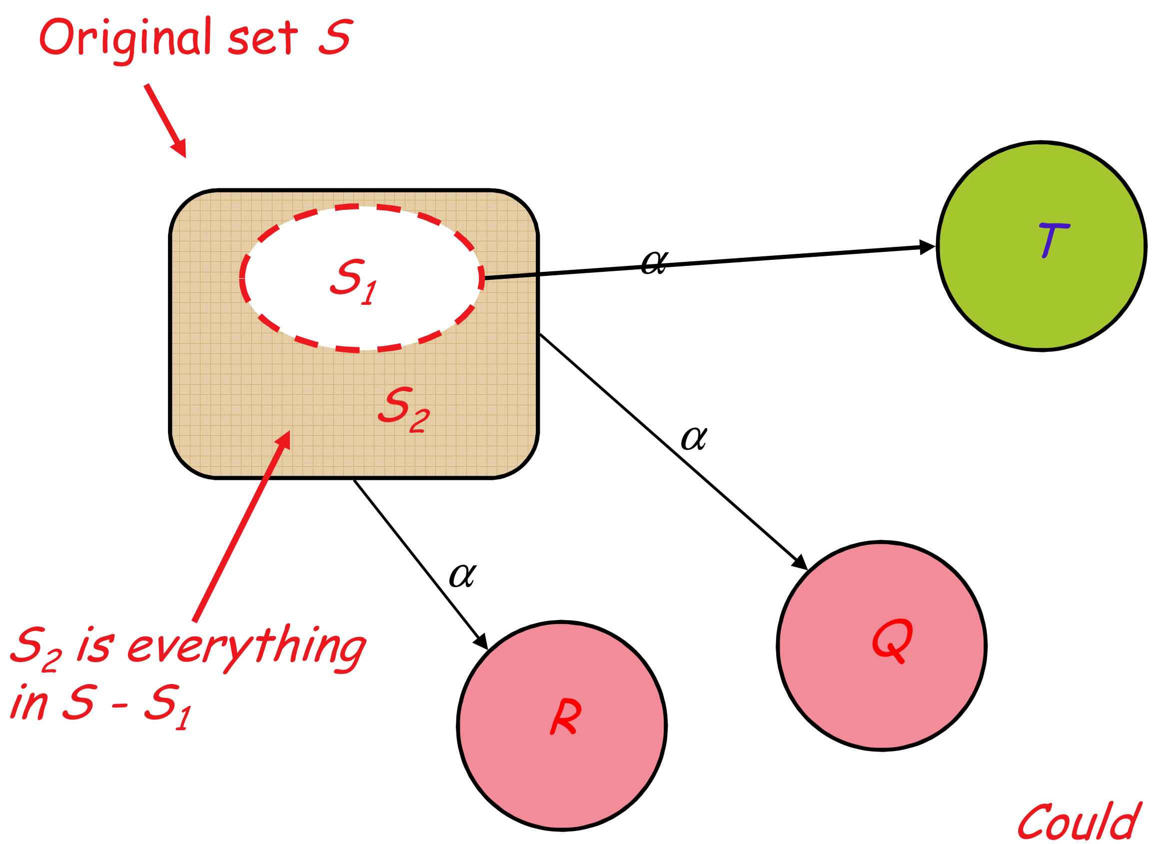
Original set *S*



S has transitions on α to *R*, *Q*, & *T*

The algorithm partitions S around α

Splitting S around α



Could we split S_2 further?

Yes, but it does not help asymptotically

DFA minimization

- Details:
  - Given DFA $(S, \Sigma, \delta, s_0, F)$
  - Initial partition: $P_0 = \{F, S\text{-}F\}$

    Intuition: final states are always different
- Splitting a set around symbol a
  - Assume s_a & $s_b \in p_i$, and $\delta(s_a, \underline{a}) = s_x$, & $\delta(s_b, \underline{a}) = s_y$
  - Split p_i if:
    - If s_x & s_y are not in the same set
    - If s_a has a transition on a, but s_b does not

    Intuition: one state in DFA cannot have two transitions on a

DFA minimization algorithm

$P \leftarrow \{ F, \{Q-F\}\}$
while (P is still changing)
    $T \leftarrow \{ \}$
    for each set $S \in P$
        for each $\alpha \in \Sigma$
            partition S *by* α
                into S_1, *and* S_2
            $T \leftarrow T \cup S_1 \cup S_2$
    if $T \neq P$ *then*
        $P \leftarrow T$

Start with two sets: final states, everything else

Build a new partitioning

For each set and each input symbol, try to partition the set

Collect the resulting sets in a new partition, see if it's different

This is a fixed-point algorithm!

Does it work?

- Algorithm halts
  - Partition $P \in 2^S$
  - Start off with 2 subsets of S $\{F\}$ *and* $\{S\text{-}F\}$
  - *While* loop takes $P_i \rightarrow P_{i+1}$ by splitting 1 or more sets
  - P_{i+1} is at least one step closer to partition with $|S|$ sets
  - Maximum of $|S|$ splits
- Note that
  - Partitions are <u>never</u> combined
  - Initial partition ensures that final states are intact

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 been 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

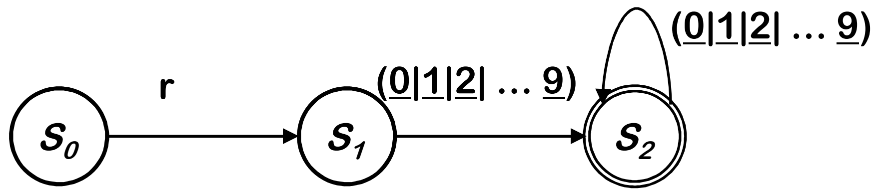
This version looks at each $S \in P$ many fewer times

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

Implementation

- Finite automaton
  - States, characters
  - State transition δ uniquely determines next state
- Next character function
  - Reads next character into buffer
  - (May compute ***character class*** by fast table lookup)
- Transitions from state to state
  - Implement δ as a table
  - Access table using current state and character

Example



Turning the recognizer into code

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

Char ← *next character*
State ← s_0

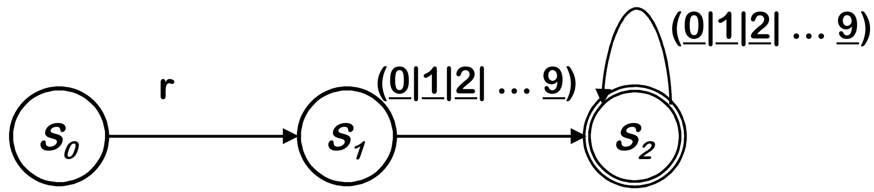
while (Char ≠ EOF)
    State ← δ(State,Char)
    Char ← *next character*

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

Skeleton recognizer

Example

Adding actions



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

Char $\leftarrow$ *next character*
State $\leftarrow$ s_0

while (Char $\neq$ EOF)
    State $\leftarrow$ δ(State,Char)
    perform specified action
    Char $\leftarrow$ *next character*

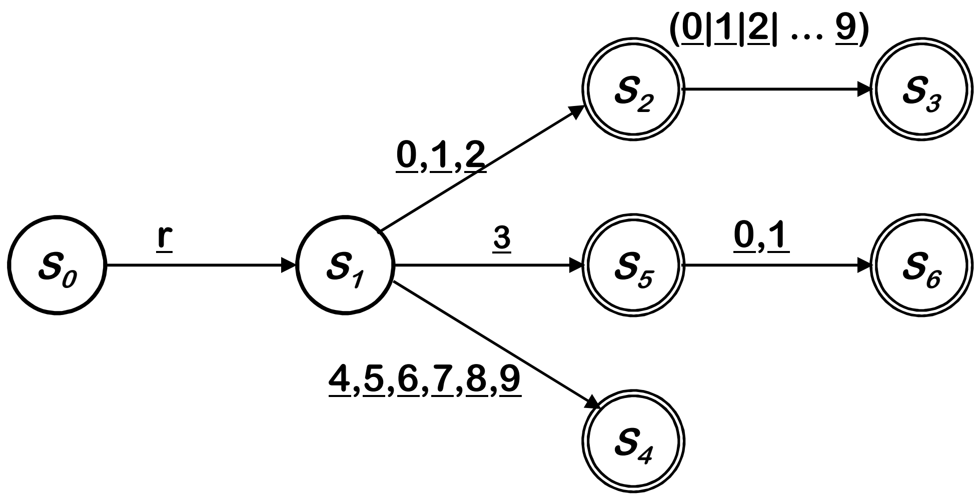
if (State is a final state)
    then report success
    else report failure

Skeleton recognizer

Tighter register specification

- 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

Tighter register specification

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

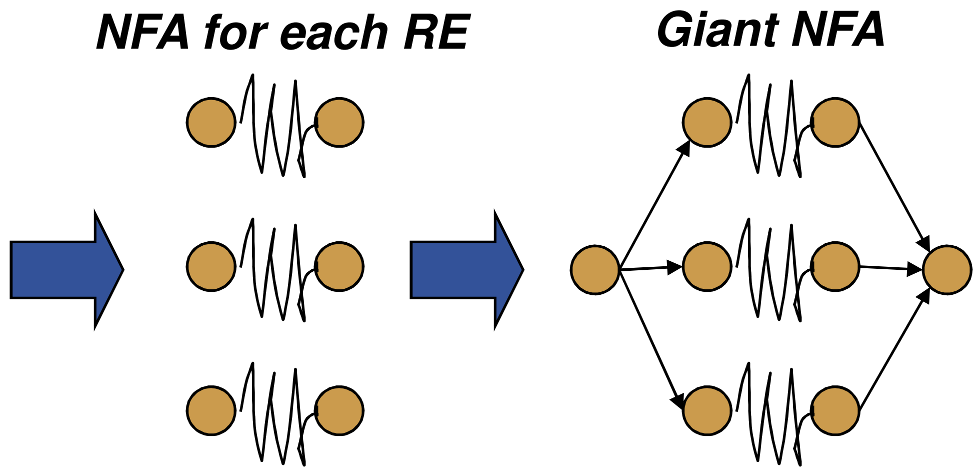
Runs in the same skeleton recognizer

Table encoding RE for the tighter register specification

Building a scanner

Specification

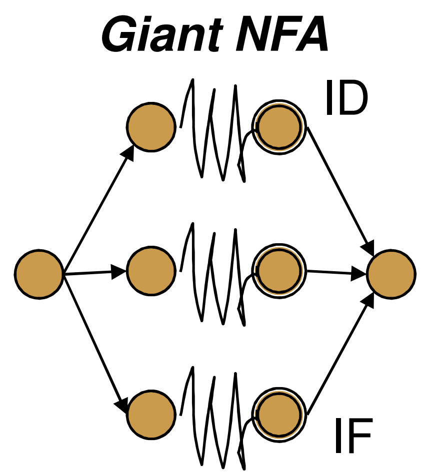
```
"if"
"while"
[a-zA-Z][a-zA-Z0-9]*
[0-9][0-9]*
(
)
…
```



- Language: `if` | `while` | `[a-zA-Z][a-zA-Z0-9]*` | `[0-9][0-9]*`…
- **Problem**:
  - Giant NFA either accepts or rejects a one token
  - We need to ***partition*** a string, and indicate the kind

Partitioning

- ***Input***: stream of characters



$\mathbf{x_0, x_1, x_2, x_3, \ldots, x_n}$

- Annotate the NFA
  - Remember the accepting state of each RE
  - Annotate with the kind of token
- Does giant NFA accept some substring $\mathbf{x_0 \ldots x_i}$?
  - Return substring and kind of token
  - Restart the NFA at $\mathbf{x_{i+1}}$

Partitioning problems

- Matching is ambiguous
  - **Example**: "`foo+3`"
  - We want <foo>,<+>,<3>
  - But: <f>,<oo>,<+>,<3> also works with our NFA
    - Can end the identifier anywhere
    - Note: "foo+" does not satisfy NFA

- Solution: ***"maximal munch"***
  - Choose the longest substring that is accepted
  - Must look at the next character to decide -- ***lookahead***
  - Keep munching until no transition on lookahead

More problems

- Some strings satisfy multiple REs
  - **Example**: "`new foo`"
  - <new> could be an identifier or a keyword

- **Solution**: rank the REs
  - First, use maximal munch
  - Second, if substring satisfies two REs, choose the one with higher rank
  - Order is important in the specification
  - Put keywords first!

C scanner

Declarations

Short-hand

REs and actions

```
%{
#include "parser.tab.h"
%}

identifier      ([a-zA-Z_][0-9a-zA-Z_]*)
octal_escape    ([0-7][^'"\n]*)
any_white       ([ \011\013\014\015])
%%
{any_white}+    { }
for             { lval.tok = get_pos(); return ctokFOR;}
if              { lval.tok = get_pos(); return ctokIF;}
{identifier}    { lval.tok = get_pos();
                  lval.idN = new idNode(cbtext, cblval.tok);
                  if ( is_typename(cbtext))  return TYPEDEFname;
                  else                       return IDENTIFIER; }
{decimal_constant}  { lval.exprN = atoi(cbtext);
                      return INTEGERconstant; }
%%
...any special code...
```

Implementation

- Table driven
  - Read and classify character
  - Select action
  - Find the next state, assign to state variable
  - Repeat

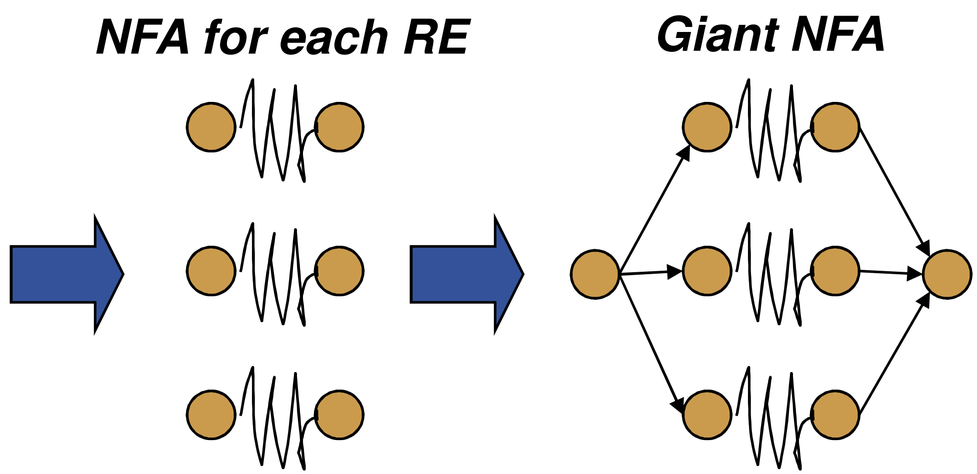
- Alternative: direct coding
  - Each state is a chunk of code
  - Transitions test and branch directly
  - Very ugly code – but who cares?
  - Very efficient

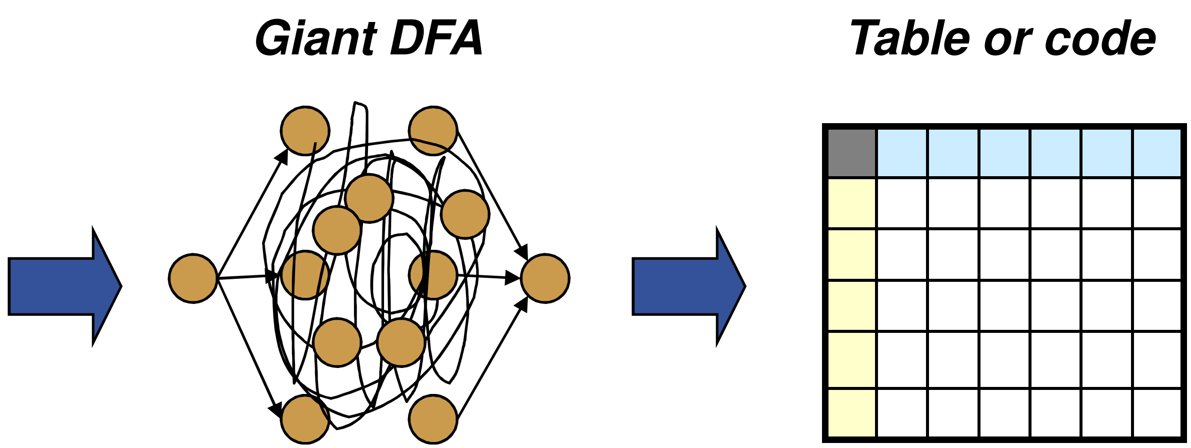
This is how lex/flex work: states are encoded as cases in a giant switch statement

Building a lexer

Specification

```
"if"
"while"
[a-zA-Z][a-zA-Z0-9]*
[0-9][0-9]*
(
)
```





Building scanners

- The point
  - Theory lets us automate construction
  - Language designer writes down regular expressions
  - Generator does: RE → NFA → DFA → code
  - Reliably produces fast, robust scanners

- Works for most modern languages

  Think twice about language features that defeat the DFA-based scanners