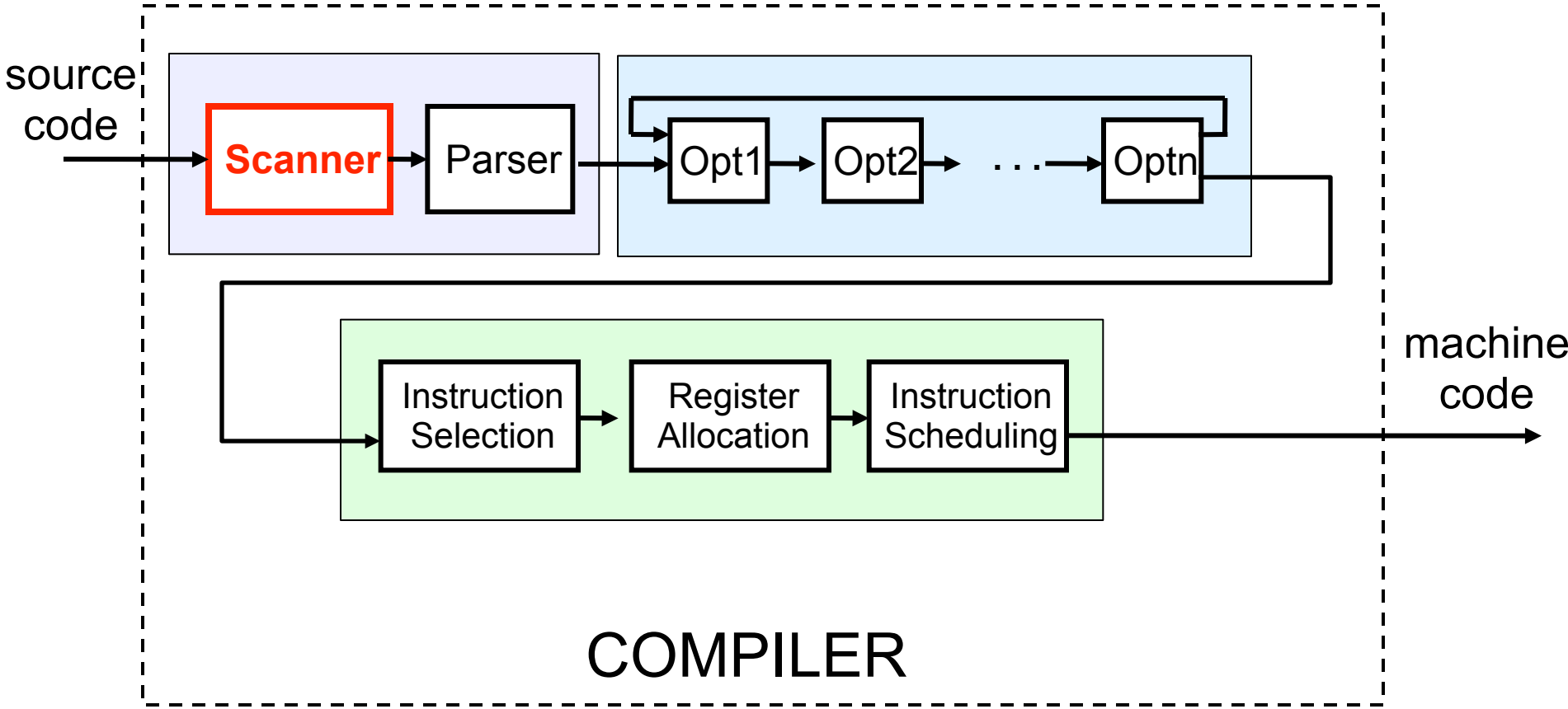
Lexical Analysis ("scanning", "lexing")

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

The Compiler Big Picture



Lexical Analysis

- Lexical Analysis is also called ‘scanning’ or ‘lexing’
- It does two things:
  - Transforms the input source string into a sequence of substrings
  - Classifies them according to their “role”
- The input is the source code
- The output is a list of tokens
- Example input:

```
if (x == y)
        z = 12;
else
        z = 7;
```

- This is really a single string:

i	f		(	x	=	=	y	)	\n	\t	z		=		1	2	;	\n	e	l	s	e	\n	\t	z		=		7	;	\n

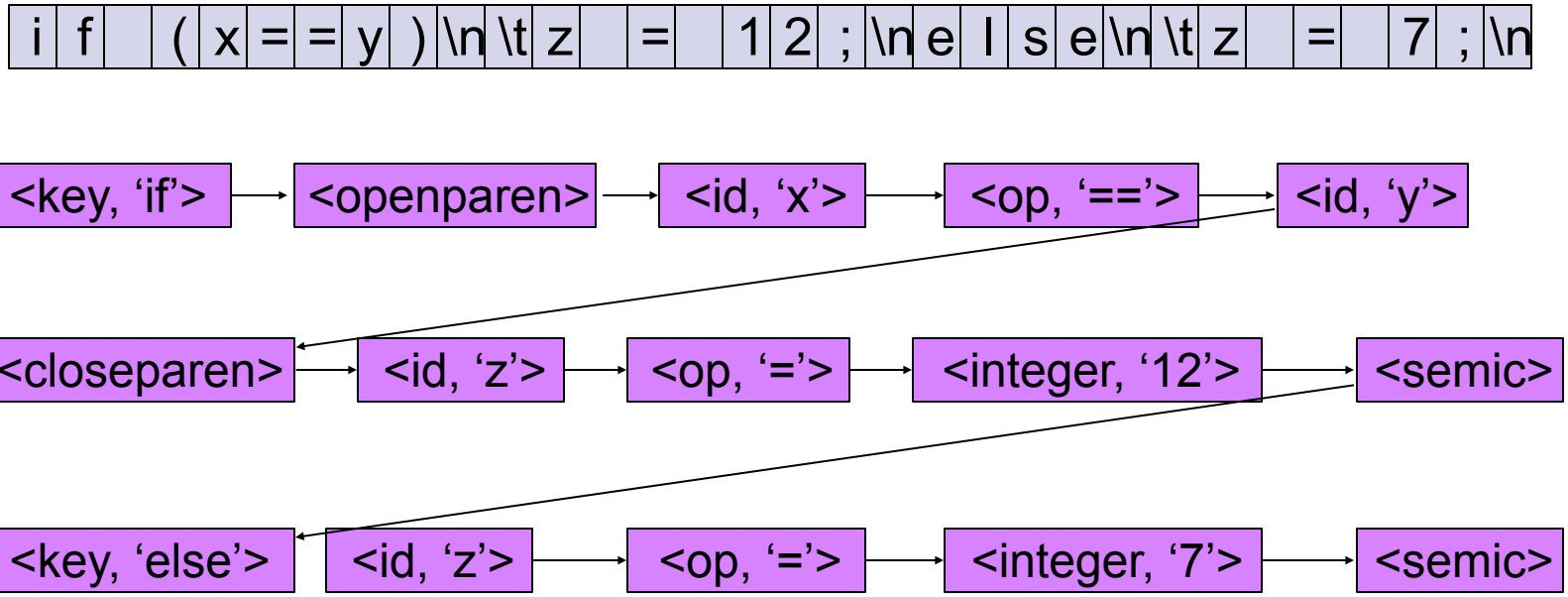
Tokens

- A token is a lexical category
- Example tokens:
  - Identifier
  - Integer
  - Floating-point number
  - Keyword
  - etc.
- In English we would talk about
  - Noun
  - Verb
  - Adjective
  - etc.

Lexeme

- A lexeme is the string that represents an instance of a token
  - In English, “big” and “small” would be lexemes off the adjective token
- The set of all possible lexemes that can represent a token instance is described by a pattern
  - Good luck with that in English :)
- For instance, we can decide that the pattern for an identifier is:
  - A string of letters, numbers, or underscores, that starts with a capital letter

Lexing output



- Note that the lexer removes non-essential characters
  - Spaces, tabs, linefeeds
  - And comments!
  - Typically a good idea for the lexer to allow for arbitrary numbers of white spaces, tabs, and linefeeds

A Lexer by Hand?

- Example: Say we want to write the code to recognizes the keyword ‘if’

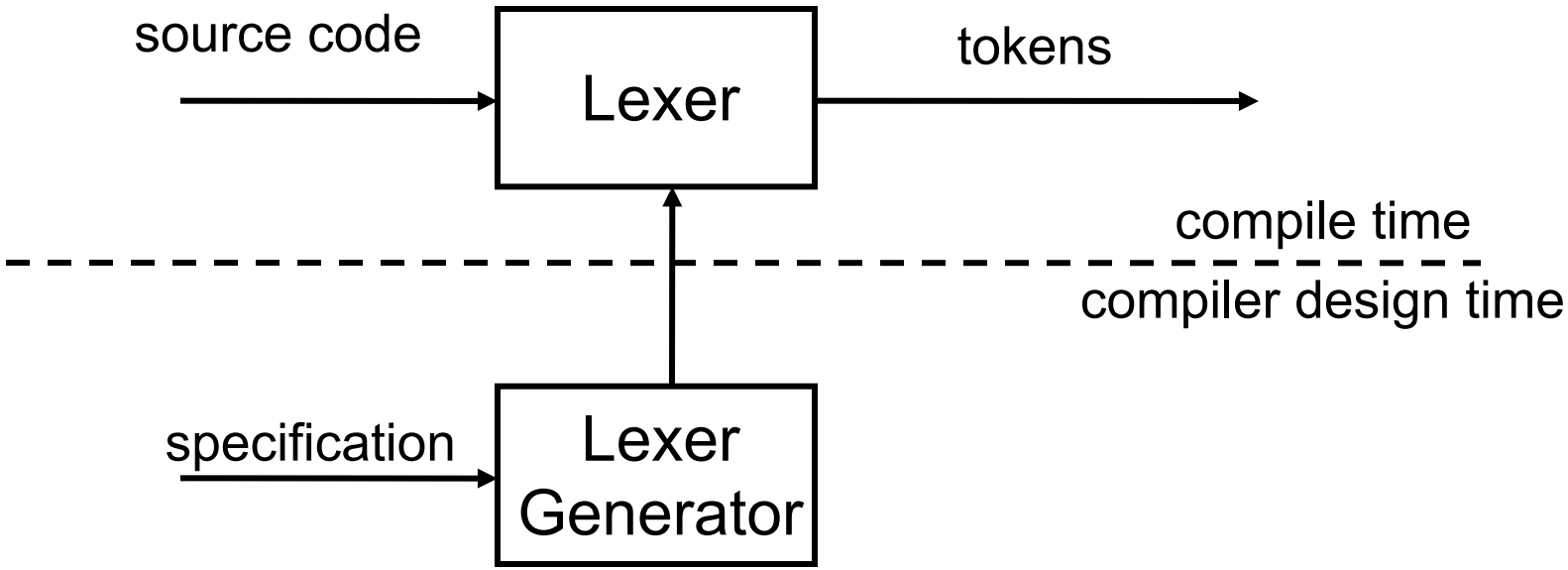
```
c = readchar();
if (c == ‘i’) {
        c = readchar();
        if (c == ‘f’) {
                c = readchar();
                if (c not alphanumeric) {
                        pushback(c);
                        emit(TOKEN_IF)
                } else {
                        // build a TOKEN_ID
                }
        } else {
                // something else
        }
} else {
        // something else
}
```

A Lexer by Hand? You're joking!

- There are many difficulties when writing a lexer by hand as in the previous slide
  - Many types of tokens
    - fixed string
    - special character sequences (operators)
    - numbers defined by specific/complex rules
  - Many possibilities of token overlap
  - Hence, many nested if-then-else in the lexer's code
- Coding all this by hand in this manner is very, very painful
  - And it's almost impossible to get it right

Automatic Lexer Generation?

- To avoid the endless nesting of if-then-else one needs a formalization of the lexing process
- If we have a good formalization, we could even generate the lexer's code automatically!



Lexer Specification

- Question: How do we tell the lexer how to recognize the tokens of a specific language?
- We need a way to describe each token precisely
- Example: “an integer is a sequence of 1 or more digits, where a digit is 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9”
- Example: “an assignment operator is an equal sign”
- Example: “a equality operator is a sequence of two equal signs”
- Example: “a variable name is a sequence of any alphanumeric character, but it can’t start with a digit, and it can’t have = or ; in it, but it’s ok if it has _ or,….”
- Example: “a floating point number is…. well, either a + or - or nothing, and then an integral part or not, and a decimal point, or not… wait…also there can be an exponent, with is denoted by ‘e’ or ‘E”, followed by an integer unless… okokok, let me start again….”
- Doing this in plain English really gets difficult!

Lexer Specification

- We need to formalize the process in the previous slide
- How do we formalize things? We define a language!
- A language is a subset of the power set of an alphabet
  - Our alphabet (typically denoted by $\sum$): the ASCII characters allowed in source code
  - Our language: a subset of all the possible strings
- Problem: our language is infinite (or very very large)
- We need to define the **patterns** of valid strings in our language
- It turns out that for all (reasonable) programming languages, the tokens can be described by regular expressions (regex)
  - See ICS 222, ICS 241, ICS 313

Regular Expressions (Regex)

- Regular expressions are notations
  - A regular expression is a string (in a meta-language) that describes a pattern (in the token language)
  - You have seen them in ICS 241!

Expression	Meaning
ε	empty pattern
a	One occurence of pattern ‘a’
ab	Strings with pattern ‘a’ followed by pattern ‘b’
a|b	Strings with pattern ‘a’ or pattern ‘b’
a^*	Zero or more occurrences of pattern ‘a’
a^+	One or more occurrences of pattern ‘a’
a?	$(a \mid \varepsilon)$
.	Any single character (not very standard)

REs for Keywords

- It is easy to define a RE that describes all keywords

Keyword = ‘if’ | ‘else’ | ‘for’ | ‘while’ | ‘int’ | ..

- These can be split in groups if needed

Keyword = ‘if’ | ‘else’ | ‘for’ | …
Type = ‘int’ | ‘double’ | ‘long’ | …

- The choice depends on what the next component (i.e., the parser) would like to see

RE for Numbers

- Straightforward representation for integers
  - digit = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’
  - integer = digit^{+}
- RE systems allow ranges, sometimes with ‘[‘ and ‘]’
  - integer = [0-9]+
- Floating point numbers are much more complicated
  - 2.00, .12e-12, 312.00001E+12, 4, 3.141e-12
- Here is one attempt
  - (’+’|’-’|ε)(digit^{+} ‘.’? | digit* (’.’ digit^{+})) ((’E’|’e’)(’+’|’-’|ε) digit+)))?
- Note the difference between meta-character and language-characters
  - ‘+’ versus +, ‘-’ versus -, ‘(’ versus (, etc.
- Often books/documentations use different fonts for each level of language
  - In these slides I use single quotes for the “programming” language

RE for Identifiers

- Here is a typical description
  - letter = ‘a’-‘z’ | ‘A’-‘Z’
  - ident = letter (letter | digit | ‘_’)*
    - Starts with a letter
    - Has any number of letter or digit or ‘_’ afterwards
- In C: ident = (letter | ‘_’) (letter | digit | ‘_’)*
- Let’s try to compile “int 3x;” with gcc..

RE for Phone Numbers

- Simple RE
  - digit = [‘0’-‘9’]
  - area = digit digit digit
  - exchange = digit digit digit
  - local = digit digit digit digit
  - phonenumber = ‘(’ area ‘)’ ‘ ‘? exchange (’-’|’ ‘) local

- The above describes the 10^{3+3+4} strings of the L(phonenumber) language

REs in Practice

- The Linux grep utility allows the use of REs
  - Example with phone numbers
    - grep ‘([0-9]\{3\}) \{0,1\}[0-9]\{3\}[-|][0-9]\{4\}’ file
  - The syntax is different from that we’ve seen, but equivalent
  - Sadly, there is no single standard for RE syntax
- Perl/Python/Java/… all implement regular expressions, all with similar-ish syntaxes
- (Good) text editors implement regular expressions
  - .e.g., for string replacements
  - Let’s see it for vim and sed
- At the end of the day, we often use tons of regular expressions
- And most programs you use everyday use REs internally
- And of course, so do compilers

In-class Exercise

- Which of the regular expressions below corresponds to: “All non-empty strings over alphabet {a,b,c}”?
  - **#1:** a* b* c*
  - **#2:** a+ b+ c+
  - **#3:** (a | b | c)+
  - **#4:** a+ | b+ | c+
  - **#5:** a | b | c

In-class Exercise (solution)

- Which of the regular expressions below corresponds to: “All non-empty strings over alphabet {a,b,c}”?
  - **#1:** a* b* c*
  - **#2:** a+ b+ c+
  - **#3:** (a | b | c)+
  - **#4:** a+ | b+ | c+
  - **#5:** a | b | c

In-class Exercise

- Write a regular expression for “All strings over alphabet {a,b,c} that contain substring ‘abc’”?

In-class Exercise (solution)

- Write a regular expression for “All strings over alphabet {a,b,c} that contain substring ‘abc’”?

(a | b | c)* a b c (a | b | c)*

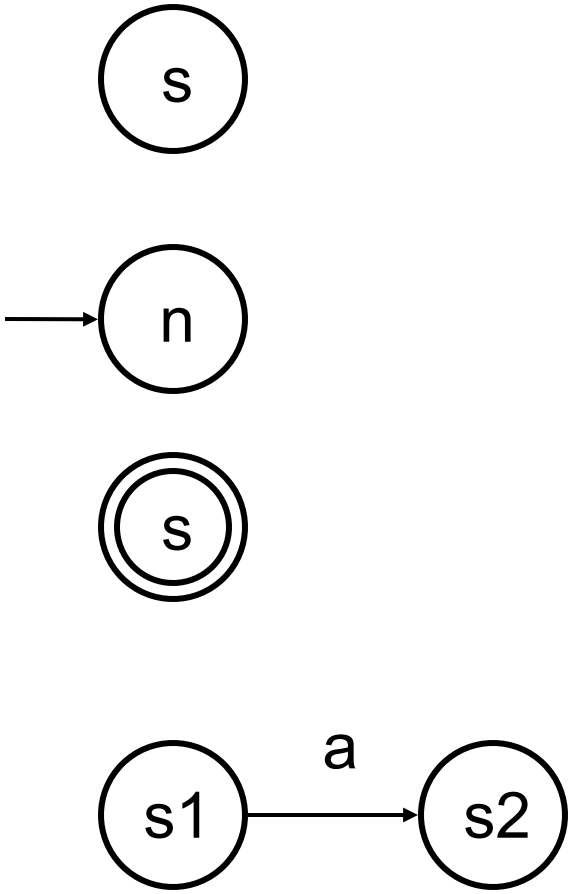
Now What?

- Now we have a nice way to formalize each token (which is a set of possible strings)
- Each token is described by a RE
  - And hopefully we have made sure that our REs are correct
  - Easier than writing the lexer from scratch!
  - But still requires that one be careful
- Question: How do we use these REs to parse the input source code and generate the token stream?
- A little bit of theory:
  - REs characterize Regular Languages
  - Regular Languages are recognized by Finite Automata (which you have also seen in ICS241)

Finite Automata

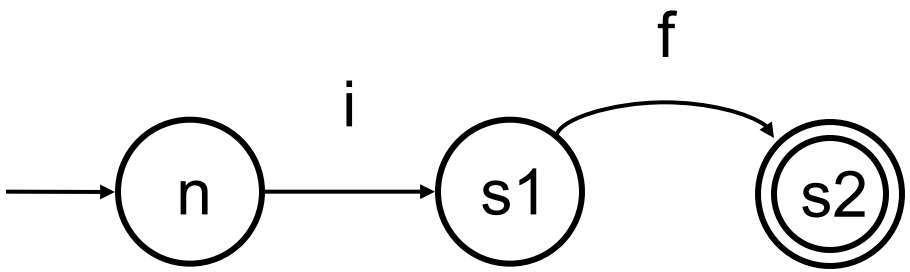
- A finite automaton is defined by
  - An input alphabet: $\sum$
  - A set of states: S
  - A start state: n
  - A set of accepting states: F (a subset of S)
  - A set of transitions between states: subset of SxS
- Transition Example
  - s1: a $\rightarrow$ s2
  - If the automaton is in state s1, reading a character ‘a’ in the input takes the automaton in state s2
  - Whenever reaching the ‘end of the input,’ if the state the automaton is in in a accept state, then we accept the input
  - Otherwise we reject the input

Finite Automata as Graphs



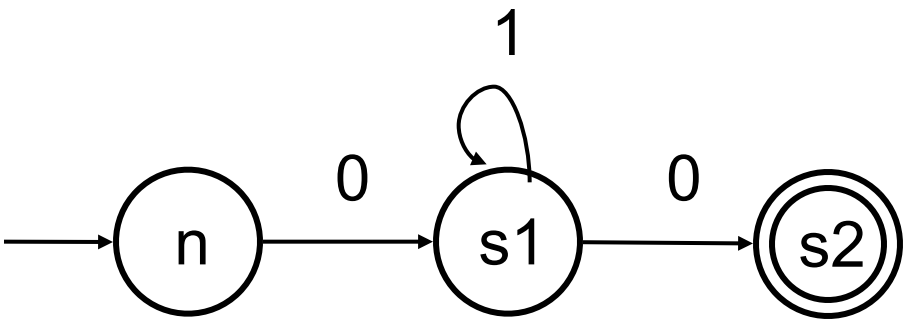
- A state
- A start state
- An accepting state
- A transition

Automaton Examples



- This automaton accepts input ‘if’

Automaton Examples



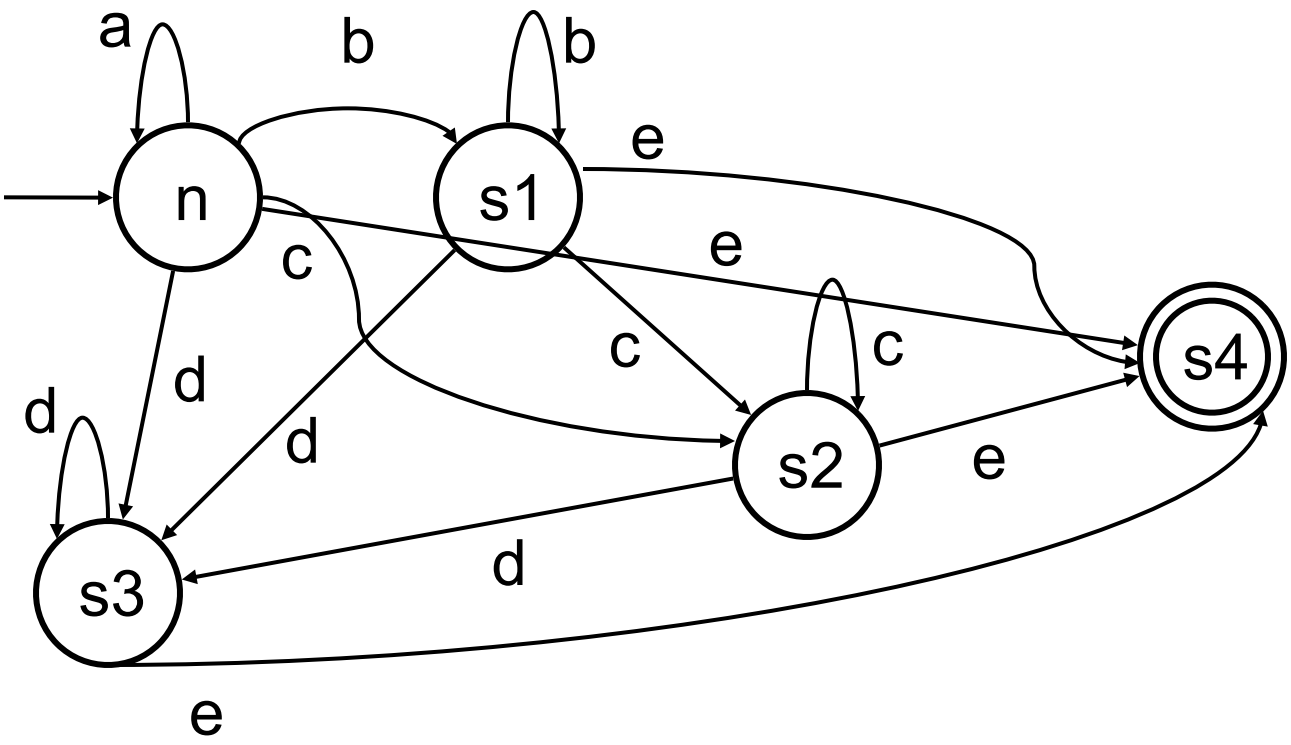
- This automaton accepts strings that start with a 0, then have any number of 1's, and end with a 0
- Note the natural correspondence between automata and REs: 01^*0
- Question: can we represent all REs with finite automata?
- Answer: yes
- Therefore, if we write a piece of code that implements arbitrary automata, we have a piece of code that implements arbitrary REs, and we have a lexer!
  - Not this simple, but really close

Non-deterministic Automata

- The automata in the previous slide are called Deterministic Finite Automata (DFA)
  - At each state, there is at most one edge for a given symbol
  - At each state, transition can happen only if an input symbol is read
    - Or the string is rejected
- It turns out that it's easier to translate REs to Non-deterministic Finite Automata (NFA)
  - There can be 'ε-transitions'!
    - Taken arbitrarily without consuming an input character
  - There can be multiple possible transitions for a given input symbol at a state
    - The automaton can take them all simultaneously (see later)

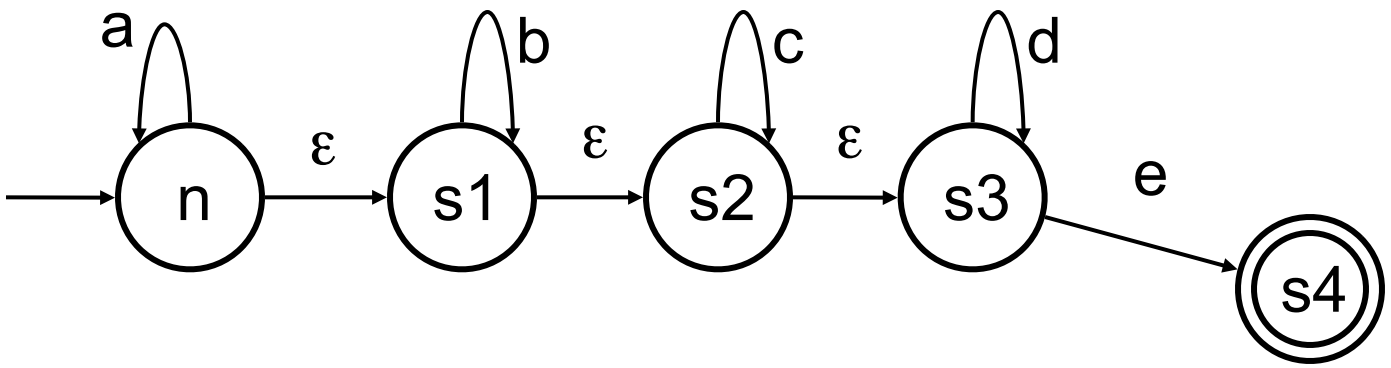
Example REs and DFA

- Say we want to represent RE ‘a*b*c*d*e’ with a DFA



Example REs and NFA

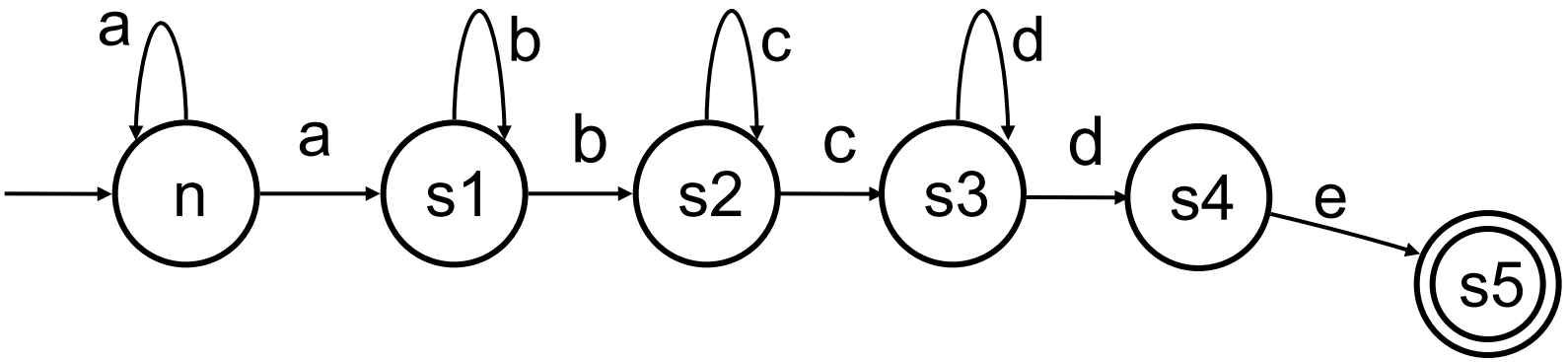
- ‘a*b*c*d*e’: much simpler with a NFA



- With ε-transitions, the automaton can ‘choose’ to skip ahead, non-deterministically
- This means that the automaton keeps track of all possible paths the automaton could have taken based on its input

Example REs and NFA

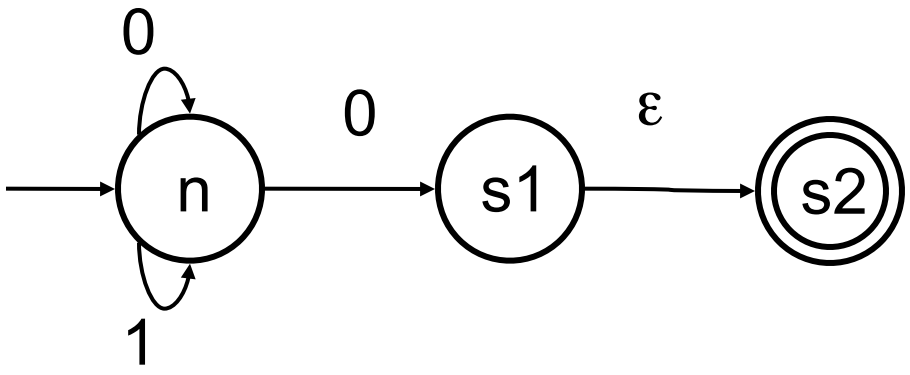
- ‘$a^+b^+c^+d^+e$’: easy modification



- But now we have multiple choices for a given character at each state!
  - e.g., two ‘a’ arrows leaving n

NFA Acceptance

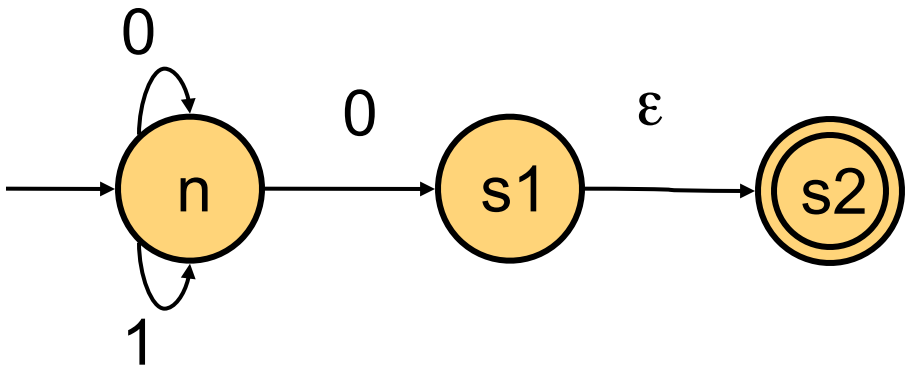
- When using an NFA, one must constantly keep track of all possible states
- If at the end of the input (at least) one of these states is an accepting state, then accept, otherwise reject



input string: 010

NFA Acceptance

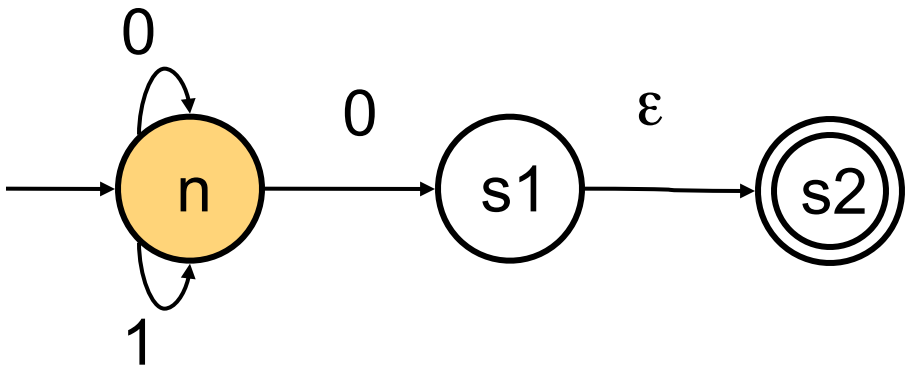
- When using an NFA, one must constantly keep track of all possible states
- If at the end of the input (at least) one of these states is an accepting state, then accept, otherwise reject



input string: 010 (after reading a 0 I could be in any state!)

NFA Acceptance

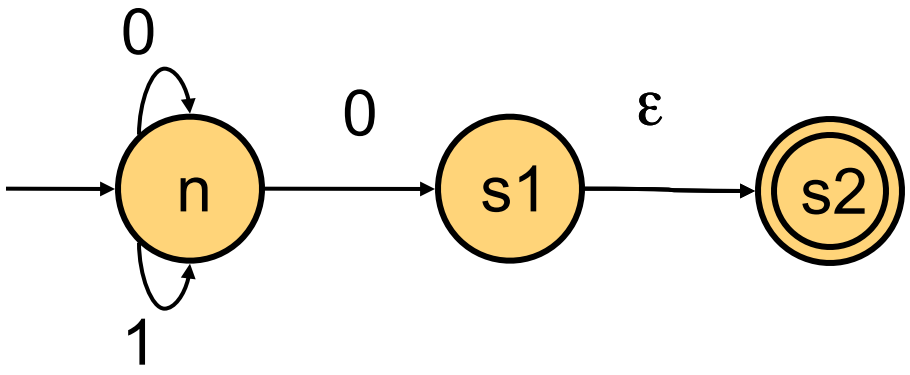
- When using an NFA, one must constantly keep track of all possible states
- If at the end of the input (at least) one of these states is an accepting state, then accept, otherwise reject



input string: 010 (after reading a 1, I can only be in state n)

NFA Acceptance

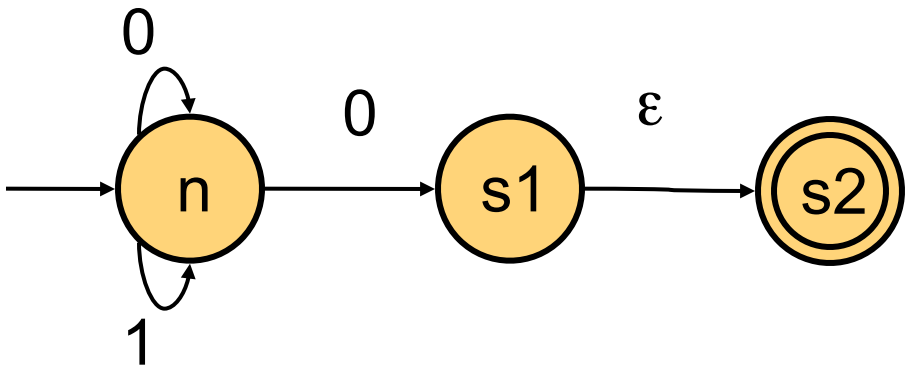
- When using an NFA, one must constantly keep track of all possible states
- If at the end of the input (at least) one of these states is an accepting state, then accept, otherwise reject



input string: 010 (after reading a 0, I can by in any state)

NFA Acceptance

- When using an NFA, one must constantly keep track of all possible states
- If at the end of the input (at least) one of these states is an accepting state, then accept, otherwise reject

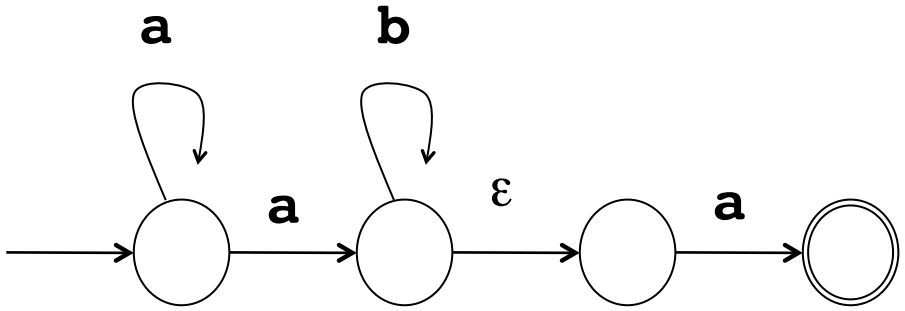


input string: 010

ACCEPT because we **could** be in s2 after the input has been processed entirely

In-class Exercise

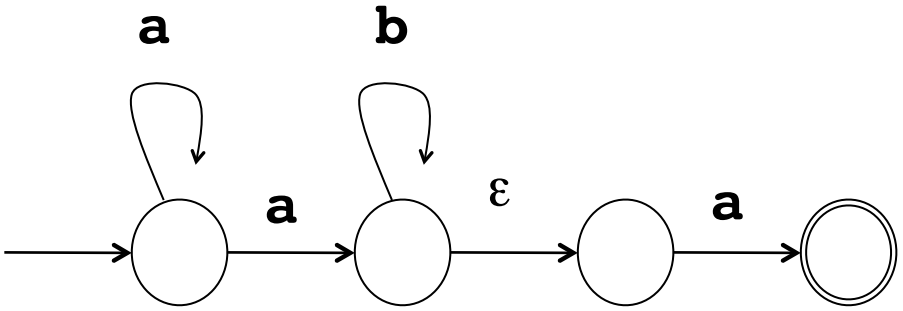
- Which RE corresponds to this automaton



- #1: a a* b a
- #2: a+ b+ a
- #3: a a* b* a
- #4: a* b* a
- #5: a+ b*

In-class Exercise (solution)

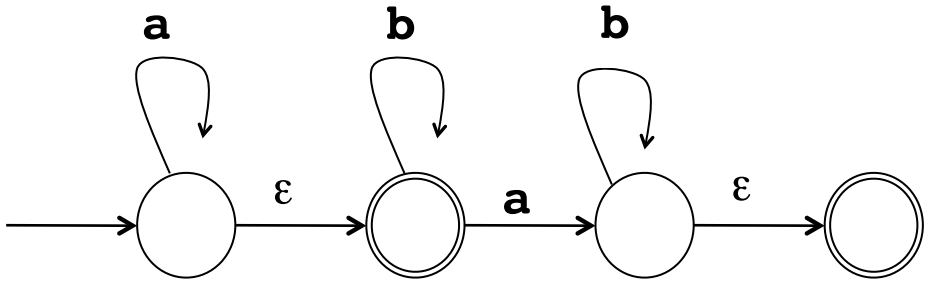
- Which RE corresponds to this automaton



- #1: a a* b a
- #2: a+ b+ a
- #3: a a* b* a (equivalent to a+ b* a)
- #4: a* b* a
- #5: a+ b*

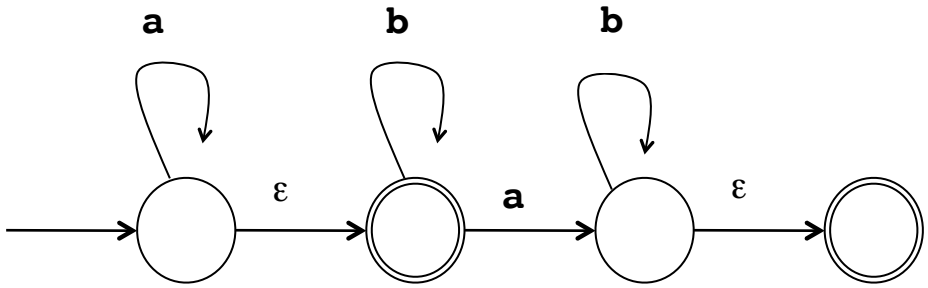
In-class Exercise

- Write a RE for this NFA



In-class Exercise (Solution)

- Write a RE for this NFA



a* b* | a* b* a b*

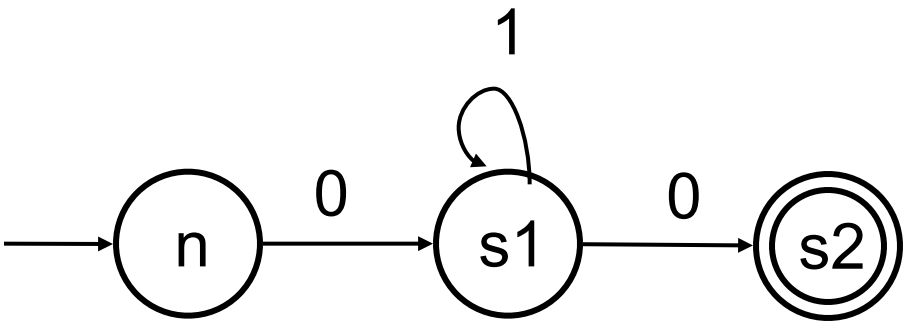
REs and NFA

- So now we're left with two possibilities
- Possibility #1: design DFAs
  - Easy to follow transitions once implemented
  - But really cumbersome
- Possibility #2: design NFAs
  - Really trivial to implement REs as NFAs
  - But what happens on input characters?
    - Non-deterministic transitions
    - We need to keep track of all possible states where the automaton could be at a given point in the input!
- It turns out that:
  - NFAs are not more powerful than DFAs
  - There are systematic algorithms to convert NFAs into DFAs and to limit their sizes (see a graduate-level course)
  - There are simple techniques to implement DFAs in software quickly

Putting it All Together

- Steps to designing/building a lexer
  - Come up with a RE for each token category
  - Come up with an NFA for each RE (easy for a human to do)
  - Convert the NFA (automatically) to a DFA (we have great algorithms to do it)
  - Write a piece of code that implements a DFA (easy for a human to do with a simple transition table)
  - Implement your lexer as a ‘bunch of DFAs’
- Let’s see an example of DFA implementation…

Example DFA Implementation



state	char	next state	decision / continue
n	0	s1	REJECT / YES
n	1	-	REJECT / NO
s1	0	s2	ACCEPT / YES
s1	1	s1	REJECT / YES
s2	0	-	REJECT / NO
s2	1	-	REJECT / NO

```
state = STATE_N
while (c == get_next_character()) {
    transition(state, c, &next_state,
                    &decision, &continue);
    if (!continue)
        break;
    state = next_state
}
return decision;
```

The ‘bunch of automata’

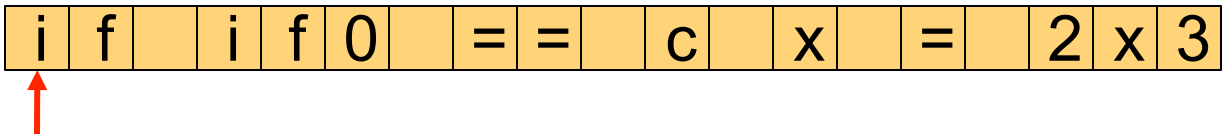
- How the lexer works
  - The lexer is simply a ‘bunch of automata’
  - It runs them all at the same time until they have all rejected the input
  - It then rewinds to the one that accepted last
    - this is the one that accepted the **longest** string
    - ‘rewinding’ uses lookahead/pushback
  - This one corresponds to the right token
  - This approach is called “maximum munch”
- Let’s look at this on an example…

Example

- Say we have the following tokens (described by a RE, and thus an NFA, and thus a DFA):
  - TOKEN_IF: ‘if’
  - TOKEN_IDENT: letter (letter | digit | ‘_’)*
  - TOKEN_NUMBER: (digit)+
  - TOKEN_COMPARE: ‘==’
  - TOKEN_ASSIGN: ‘=’
- This is a very small set of tokens for a tiny language
- The language assumes that tokens are all separated by spaces
- Let’s see what happens on the following input:

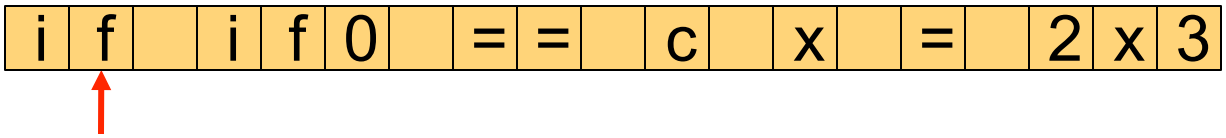


Example



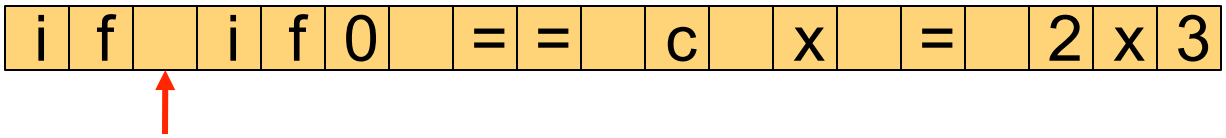
DFA	DECISION
TOKEN_IF	ok so far
TOKEN_IDENT	ok so far
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

Example



DFA	DECISION
TOKEN_IF	ok so far
TOKEN_IDENT	ok so far
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

Example

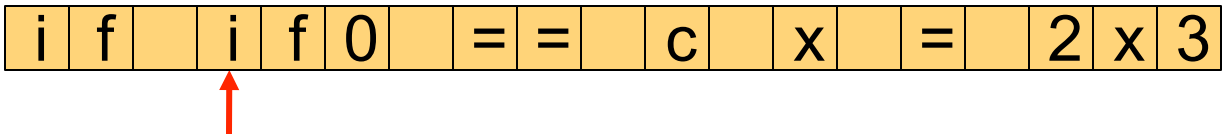


DFA	DECISION
TOKEN_IF	accept
TOKEN_IDENT	accept
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

Both TOKEN_IF and TOKEN_IDENT were the last ones to accept

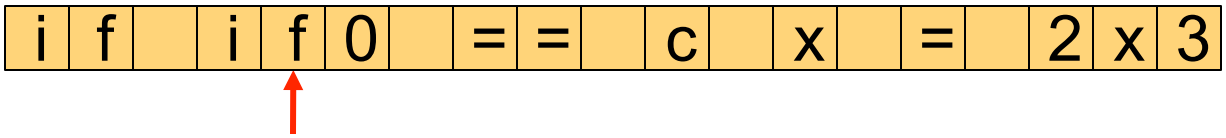
Emit TOKEN_IF because we build our lexer with the notion of **reserved** keywords

Example



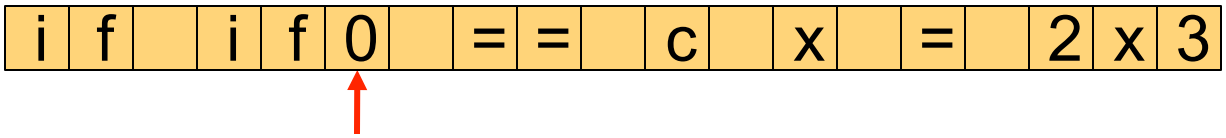
DFA	DECISION
TOKEN_IF	ok so far
TOKEN_IDENT	ok so far
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

Example



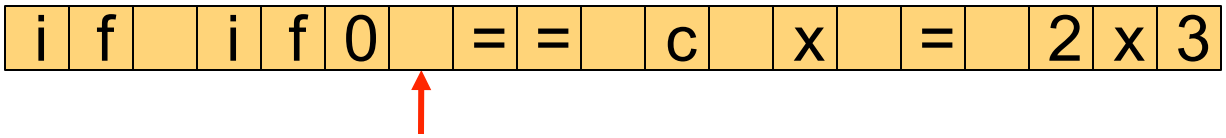
DFA	DECISION
TOKEN_IF	ok so far
TOKEN_IDENT	ok so far
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

Example



DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	ok so far
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

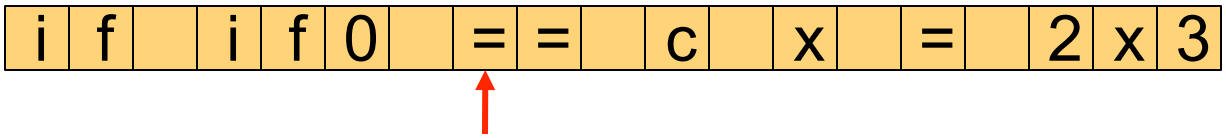
Example



DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	accept
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

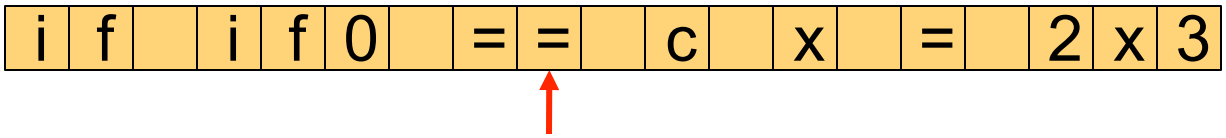
Emit TOKEN_IDENT (with string ‘if0’) because it accepted the latest

Example



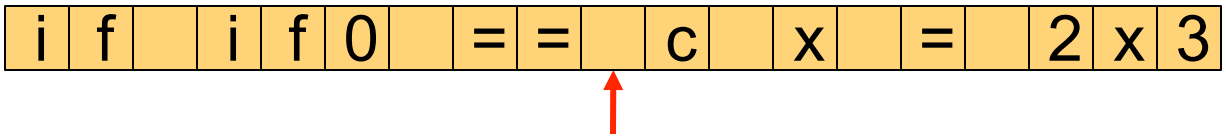
DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	reject
TOKEN_NUMBER	reject
TOKEN_COMPARE	ok so far
TOKEN_ASSIGN	ok so far

Example



DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	reject
TOKEN_NUMBER	reject
TOKEN_COMPARE	ok so far
TOKEN_ASSIGN	reject

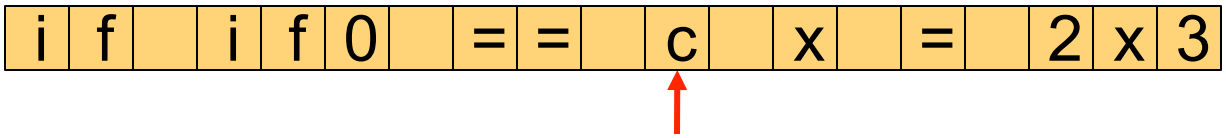
Example



DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	reject
TOKEN_NUMBER	reject
TOKEN_COMPARE	accept
TOKEN_ASSIGN	reject

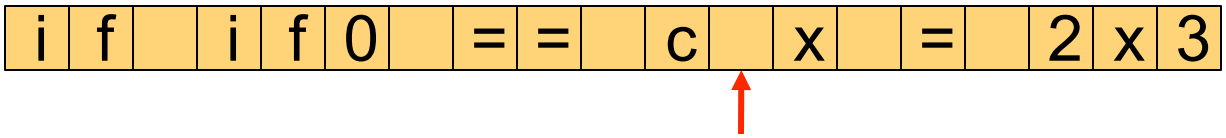
Emit TOKEN_COMPARE because it accepted the latest

Example



DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	ok so far
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

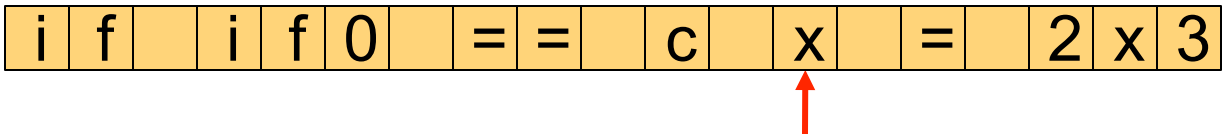
Example



DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	accept
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

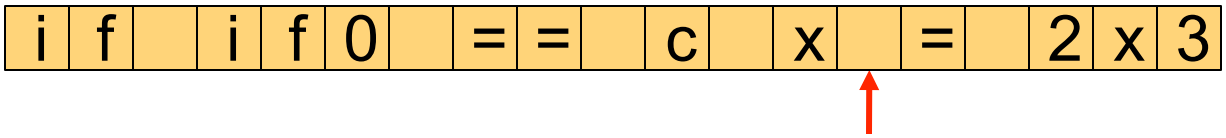
Emit TOKEN_IDENT (with string ‘c’) because it accepted the latest

Example



DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	ok so far
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

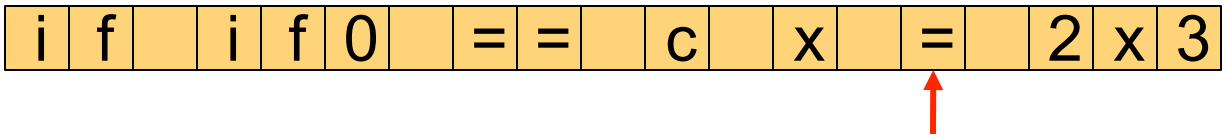
Example



DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	accept
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

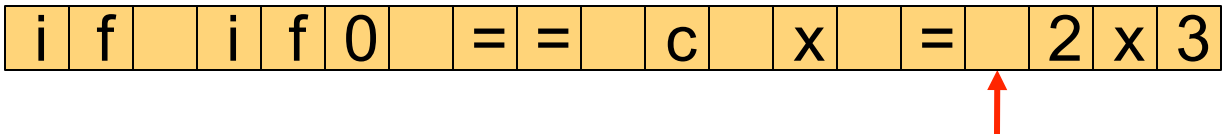
Emit TOKEN_IDENT (with string ‘x’) because it accepted the latest

Example



DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	reject
TOKEN_NUMBER	reject
TOKEN_COMPARE	ok so far
TOKEN_ASSIGN	ok so far

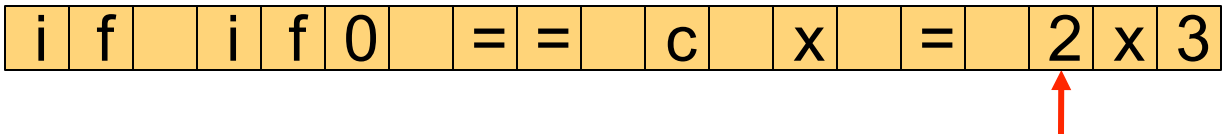
Example



DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	reject
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	accept

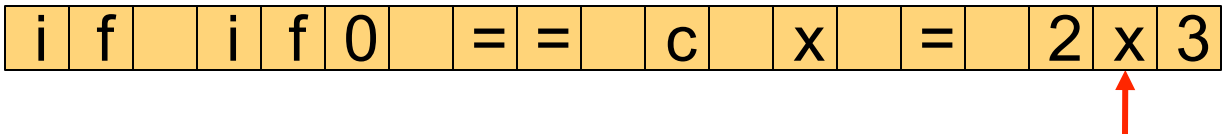
Emit TOKEN_ASSIGN because it was the only one accepted

Example



DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	reject
TOKEN_NUMBER	ok so far
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

Example



DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	reject
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

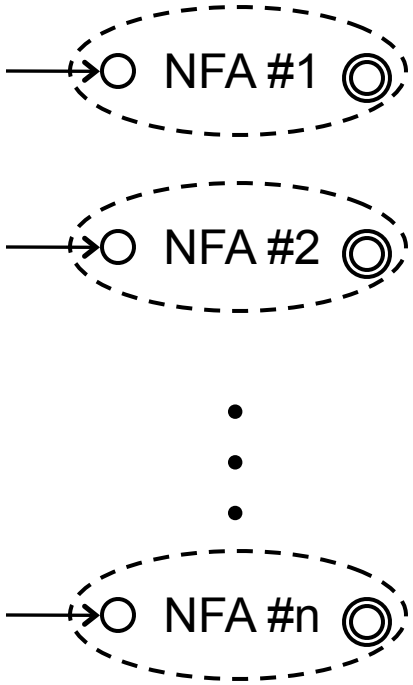
Abort and print a Syntax Error Message!!

Example

- If there had been no syntax error, the lexer would have emitted:
  - <TOKEN_IF>
  - <TOKEN_ID, ‘if0’>
  - <TOKEN_COMPARE>
  - <TOKEN_ID, ‘c’>
  - <TOKEN_ID, ‘x’>
  - <TOKEN_ASSIGN>
  - <TOKEN_NUMBER,’23’>

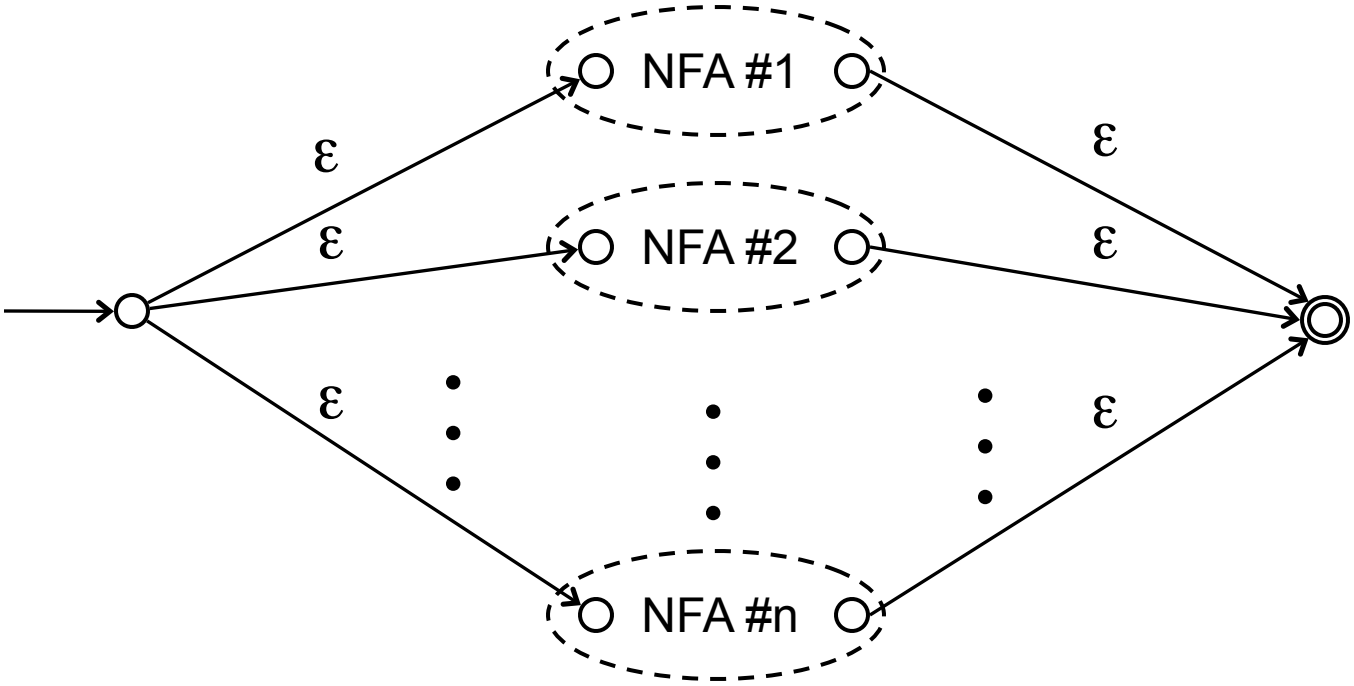
Implementing ‘bunch of DFAs’

- We have one NFA per token
- We can easily combine them in one single NFA



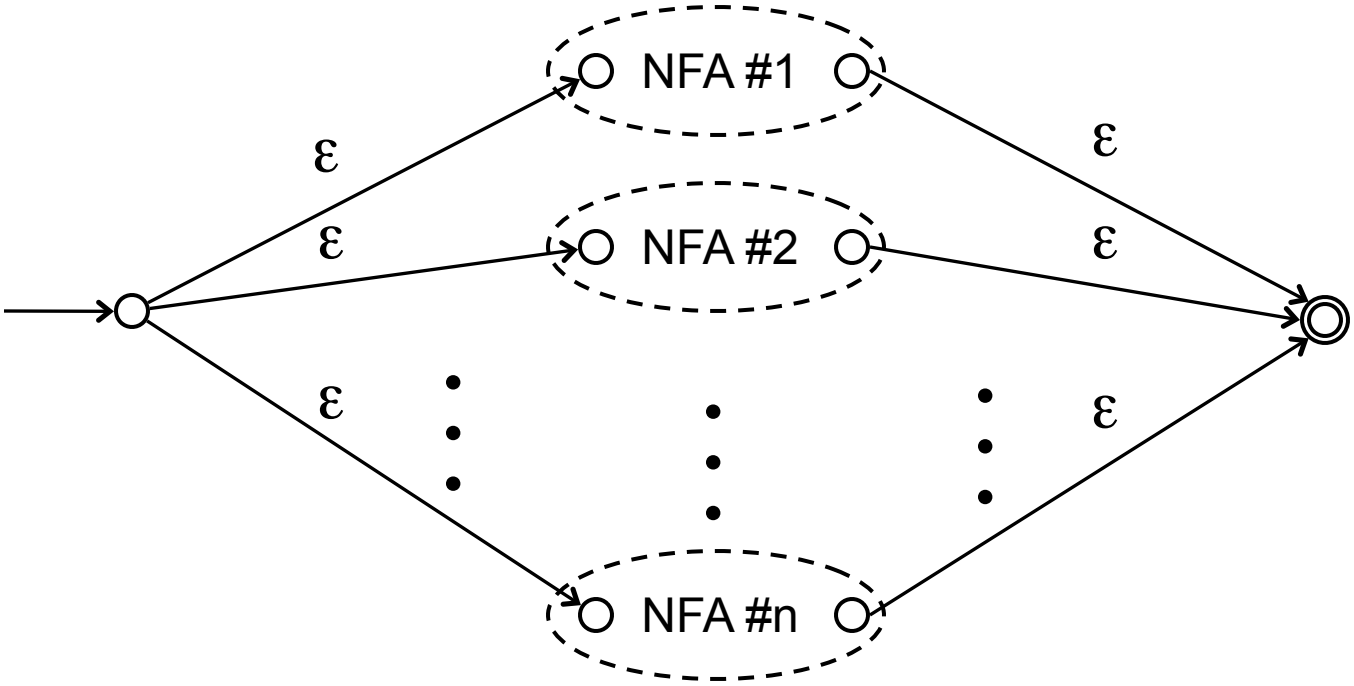
Implementing ‘bunch of DFAs’

- We have one NFA per token
- We can easily combine them in one single NFA



Implementing ‘bunch of DFAs’

- We have one NFA per token
- We can easily combine them in one single NFA



- We can then convert it to a (massive!) DFA

Implementing a Lexer

- Implementing a Lexer is now straightforward
  - Come up with a RE for each token category
  - Come up with an NFA for each RE
  - Merge them into a bunch of NFAs
  - Convert this (enormous) NFA (automatically) to a (ginormous) DFA
  - Write a piece of code that implements a DFA
  - And voila!
- The above has been understood for decades
- And it's so "easy" that we now have automatic lexer generators!
- Well-known examples are lex, flex

Important Takeaways

- The patterns of the tokens in a programming language are described by humans first via regular expressions, and then via non-deterministic finite automata
- A lexer implements token recognition using deterministic finite automata
- You must be:
  - Familiar with regular expression syntax
  - Able to write correct regular expressions
  - Be able to go back and forth between a regular expression and a non-deterministic finite automata

Conclusion

- Lexing has been well-understood for decades and lexer generators are available
  - The only motivation to write a lexer by hand today: speed
- There are popular tools to automatically generate the code of a lexer
- Next set of lecture notes we'll see something more powerful than automata/regex: context-free grammars!
- But first, let's do some of the posted Practice Problems…