Scanning

- When you write a computer program, your view of the world is that you are writing C or java or Pascal
  - But in reality all you are doing is typing ASCII characters that have some contextual meaning to you
  - Your “program” is stored on disk as a simple ASCII file
- From the compiler’s perspective, it is told to go look at your file, and turn it into assembly code
  - All that it has to work with is a series of ASCII characters
  - Some special characters help – spaces, tabs, newlines
    - We call these whitespace characters
- First thing compiler has to do is convert a series of ASCII characters into some context
  - Find the “words” in the file
  - This is the job of the **Scanner** or **Lexical Analyzer**

Scanning

- The scanner breaks the input ASCII file up into pieces, called **tokens**
  - Each token represents some aspect of a computer program
    - Identifiers - variables, class names, method names
    - Key words, like if, else, class
    - Constants - numbers, either integer or floating point
    - Symbols - + ; < ( ) [ ], etc.
  - We assign a TokenType to each keyword and symbol
  - Just one TokenType for all Identifiers and one (or two) for all Constants
- When it finds a token, the scanner will return an object of type Token
  - What instance variables would the Token class need?

Scanning

- The Token class will need to store
  - TokenType
  - Constant value for constants (either int or float)
  - String for Identifiers
- In C++, the constant value and the string value could be put in a union, requiring less storage space
- In Java, how could we combine them?

- We just declare an Object reference, which can hold a String, Integer, Float, or Double
- So what do constructors need to look like?

Scanning

- So … what's a TokenType?
- Anything else our Token class needs to do?

```
public class Token {

    private TokenType tokenType;
    private Object tokenData;

    public Token (type) {
        this (type, null);
    }

    public Token (TokenType type, Object data) {
        tokenType = type;
        tokenData = data;
    }

    // some access methods
}
```

Scanning

- We need to define the enum for TokenType
  - Probably within the Scanner package somewhere, most likely in the Token class

```
public class Token {
    public enum TokenType {
        IDENT_TOKEN,
        ASSIGN_TOKEN,
        IF_TOKEN,
        // rest of tokens ....
    }

    private TokenType tokenType;
    private Object tokenData;
        // rest of class ....

}
```

Scanning

- Well, if we just knew how to find tokens, we at least have a data structure for them!
- How do we find tokens in the ASCII file?
  - Whitespace characters ought to help
    - Spaces, tabs, newlines certainly break up tokens
    - Comments also break up tokens
  - If all tokens were broken up by whitespace, scanning would be pretty easy
    - distance = velocity * acceleration ;
  - However, the following is also legal
    - distance=velocity*acceleration;
  - Scanning is made even more difficult with ambiguity
    - Is <= a LESSTHANEQUAL_TOKEN, or two separate tokens: LESSTHAN_TOKEN and ASSIGN_TOKEN
    - It seems context is important in scanning

Scanning

- Hopefully you have an idea now of what task our Scanner has to do
- Let's talk about the context in which a Scanner is used
  - Two basic ways we could use a Scanner
    1. It reads in the entire file at one time, and creates a linked list of tokens, or possibly writes the tokens in some format to a file
    2. It reads one token at a time, as the rest of the compiler needs the information
  - Turns out the 2nd way is typically the most useful
  - The Parser needs to go through the entire file, working with Tokens
    - On demand, it asks the Scanner for the next Token
  - Note that your main() method probably would want to be able to test a complete scan

Scanning

- Since we understand the context, we should be able to define the ADT
  - Because Parsers may need to do Look-Ahead to next Token, need a “view” method that doesn’t cause Scanner to look further into file

```
public interface Scanner {

    public Token getNextToken ();
    public Token viewNextToken ();

}
```

Scanning

- Let’s sketch what a particular Scanner might look like

```
public class CMinusScanner implements Scanner {

    private BufferedReader inFile;
    private Token nextToken;

    public CMinusScanner (String filename) {
        inFile = new BufferedReader(new FileReader(filename));
        nextToken = scanToken();
    }
    public Token getNextToken () {
        Token returnToken = nextToken;
        if (nextToken.getType() != Token.TokenType.EOF_TOKEN)
            nextToken = scanToken();
        return returnToken;
    }
    public Token viewNextToken () {
        return nextToken;
    }
}
```

Scanning

- Hopefully we've got the big picture now
  - We just need to develop the scanToken() method

- Unfortunately, to write this method, we need to learn a bit of theory
  - We've seen that scanning can be a bit tricky, since it's not just a matter of finding whitespace
- Before we can build our scanToken method, we need to learn about:
  - Regular Expressions
  - Finite Automata

Scanning

- Why do we need all this theory about Regular Expressions and Finite Automata?
  - We need a way of expressing what legal sequence of ASCII characters form a Token such as an Identifier or Constant
    - Guy1 is a legal Identifier; 1Guy is not
    - 1.34e-23 is legal constant; 1.34g-23 is not
  - Regular Expressions is a convenient form for describing the requirements for a series of characters to form a legal Token
  - A Finite Automata (like state diagrams) is a machine that knows how to recognize or accept Regular Expressions
  - Approach
    1. Express language requirements using Regular Expressions
    2. Implement Finite Automata which accept the language
       - Automated tools (Lex, Flex) exist to do this

Regular Expressions

- A Regular Expression is a format for expressing patterns of characters
- For a given Regular Expression *r*, there is a set of strings (possibly an infinite set) that match it
  - Example – “the set of strings which begin with an a and end with an a”
  - We call this set of strings the Language of *r*, or *L(r)*
  - Only certain characters are legal in a particular Regular Expression
    - We call this the Alphabet of the Regular Expression, and symbolize it with a Σ (Sigma)
    - Generally, this will be the ASCII character set or some subset

Regular Expressions

- Example - “the set of strings which begin with an a and end with an a”
  - Assuming our Alphabet is the set {a, b}, we can express this as: a(a|b)*a|a

- We can define regular expressions as a series of rules
  1. ε is a regular expression denoting {ε}, the set containing the empty string
  2. If a is a symbol in Σ, then a is a regular expression that denotes {a}
  3. Choice (the “or” operation) – denotes by the | symbol
  4. Concatenation – denoted by juxtaposition of characters
  5. Closure (repetition) – denoted by * symbol
  6. Parentheses– to override precedence

Regular Expressions

- Example: Find a Regular Expression for “the set of strings with a single b surrounded by equal numbers of a’s”
  - {b, aba, aabaa, …}

- This set cannot be described by a Regular Expression
  - One way of looking at it is “regular expressions can’t count”
  - a*ba* comes close, but no guarantee that first set of a’s is same length as 2nd set
- Example: Find a Regular Expression for “the set of strings which begin with “ab” and end with “ba”, with no occurrences of “ba” in between
  - Does this sound familiar?
  - Turns out it is do-able, but complicated
  - Comments of this nature are usually searched for (and eliminated) by Scanner as a special case

Regular Expressions

- We have looked at basic rules for Regular Expressions
  - Let's look at some common extensions to simplify expressing more complicated items
- Named items
  - It gets old doing 0|1|2|3|…|9
  - We can do digit=0|1|2|3|…|9
  - An integer might then be: digit digit*
- Not operator
  - Frequently want to say "the set of strings such that the 1st character is not a …."
  - We can say ~a or ¬ a
- One or more occurrences
  - Frequently want to say "one of more … followed by …."
  - $(a|b)^+$ means "one or more of either a or b"

Regular Expressions

- Other shortcuts
  - Any character
    - Maybe want to express "… followed by any character, followed by …."
    - Use . symbol – "any expression containing at least one b" could be expressed .*b.*
  - Range of characters
    - Rather than digit = 0|1|2|3|…|9
    - Can say digit = [0-9]
  - Zero or one occurrence (optional occurrence)
    - Instead of a | ba | ca
    - Can say (b|c)? a
- Different automated Scanner tools (like Lex) define their own standards for shortcuts

Regular Expressions

- That's a little bit about Regular Expressions
  – Let's look at them from a compiler perspective

- We already saw how to do Identifiers
  – ident = (letter|undersc) (letter|undersc|digit)*
- But how do we distinguish between Identifiers and Keywords?
  – Several approaches
    1. Don't worry about it, and let the Parser figure it out
    2. After deciding you've found an Identifier, string compare the Identifier against known keywords
    3. Define Regular Expression matching the keywords (shouldn't be too difficult!) and then build in some sort of precedence such that if a string is matched by both an Identifier and a keyword, the keyword is chosen

Regular Expressions

- How are Constants handled
  - Integers could be digit+
  - Floats (simplified) might be
    - nat = [0-9]+
    - signedNat = (“+”|”-”)? nat
    - float = signedNat (“.” nat)?((e|E) signedNat)?
      - Note that since the “.” is a legal metacharacter, we put it in quotes to indicate the character
        - Probably more standard to use \.
  - String constants might be
    - quote (~quote)* quote

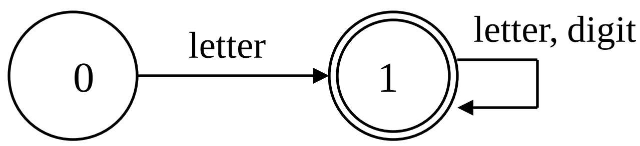
Finite Automata

- If you are going to use an automated tool such as Lex, then that's as far as you need to go
  - Just express your compiler language as a series of Regular Expressions (typically using a syntax unique to the tool), and then the tool will automatically generate a Scanner for you
  - We will look more at this later
- Next we will look at Finite Automata
  - This gives you the tools to build a Scanner by hand
  - This is also the way that the automated tools build a Scanner

- Remember, we said that a Finite Automata was a machine for evaluating whether a particular string matches a Regular Expression
  - We will learn to build Finite Automata from Regular Expressions

Finite Automata

- Start with example of ident=letter (letter|digit)*
  - Can implement this with a state diagram



- Diagram includes
  - Start state – state 0
  - Accepting state(s) – state 1
  - Set of legal transitions
- How would the string “hello” be treated by this state diagram?
  - If in Accepting State at end of string, then this is a valid string matching the Regular Expression

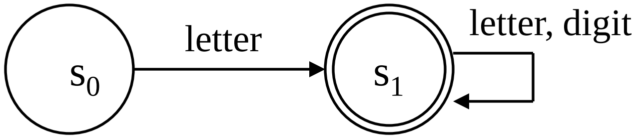
Finite Automata

- We will be looking at 2 types of Finite Automata in this course
  1. Deterministic Finite Automata (DFA) – given the current state and an input character, the next state can be uniquely identified
  2. Non-deterministic Finite Automata (NFA) – next state may be ambiguous based on input character

- Let's give a more precise definition for DFAs
  - A mathematic model consisting of 5 elements
    1. An alphabet Σ
    2. A set of states S
    3. A start state s_0
    4. A set of accepting states F
    5. A transition function T: S X Σ

Finite Automata

- For the DFA below
  - Σ = {letter, digit, other}
  - S = {s_0, s_1}
  - s_0
  - F = {s_1}
  - T: S X Σ
- Note: no transitions on “other” are shown
  - By convention, transitions to an “error” state are not shown, to simplify the diagram



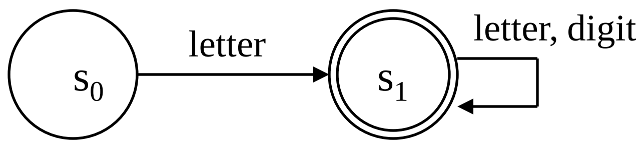
Finite Automata

- Given a DFA, it should be fairly straightforward to implement in the compiler
  - Can be straightforwardly done with switch statement
    - As complexity goes up, may use nested switch statements

```
switch (state) {
    case (States.Start):
        if (isLetter(inputChar)
            state = States.InVar;
        else scanError (inputChar);
        break;
    case (States.InVar):
        if ( (isLetter(inputChar) || isDigit (inputChar) )
            state = States.InVar;
        else scanError (inputChar);
        break;
}
```

Finite Automata

- An even more powerful way of implementing a DFA is to use a table-driven approach
  - The transition function is represented in a 2D table



input / State	letter	digit	other	Accept?
0	1			No
1	1	1		Yes

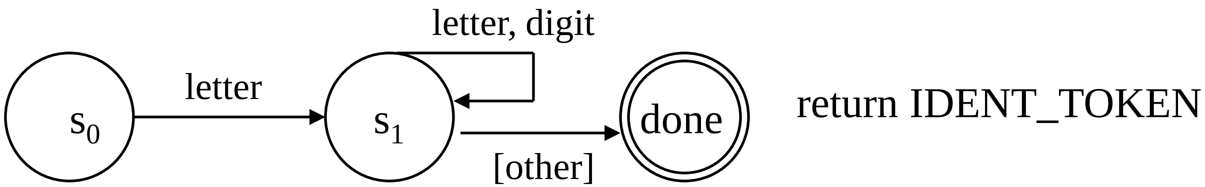
Finite Automata

- So, we can go from DFAs to code fairly easily
- The problem is that it is not easy to automate the transition from Regular Expressions to DFAs
  - We can do it by hand fairly well, but it is not easy to automate

- Problems with simple DFAs
  - Not all transitions from the accepting state are errors
    - For statement: studentGrade=“F”, if I am in an accepting state for Identifiers after seeing “studentGrade”, the “=“ is not an error, but the beginning of the next Token
      - Need to be able to indicate I have found the Identifier token, and am working on the next
    - This decision required the use of a lookahead character

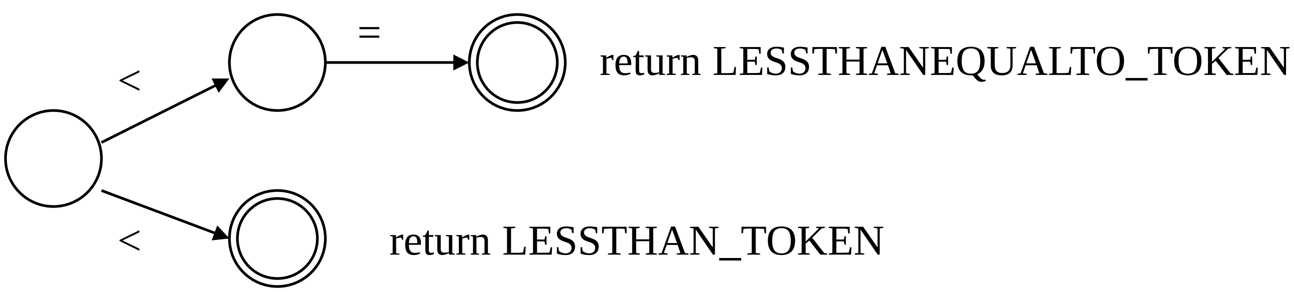
Finite Automata

- Problems with simple DFAs (cont)
  - Lookahead characters
    - Need to be able to see next char, without consuming it
    - Another alternative is backtracking
      - We just consume the next character, but if we see that it is not part of this string, we somehow "push" it back into the character stream
    - Note use of [other] to indicate it is not consumed



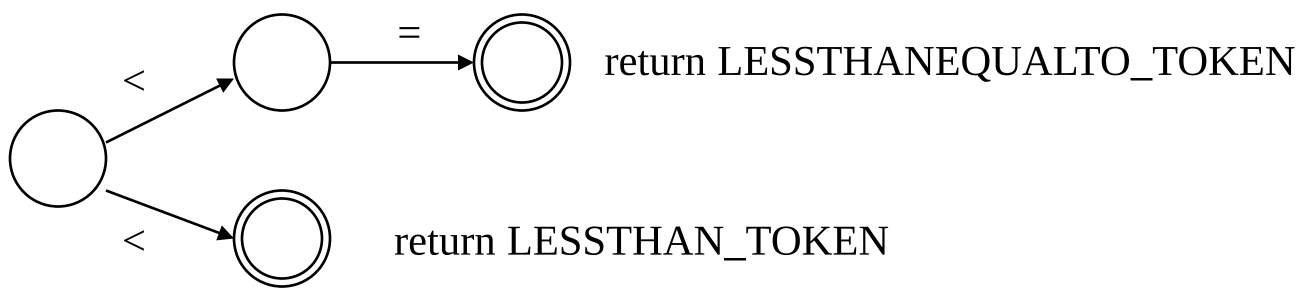
Finite Automata

- Problems with DFAs (cont)
  - Another problem is that DFAs require that you deterministically move from one state to another
  - Consider trying to recognize the LESSTHAN_TOKEN
    - How do you differentiate it from LESSTHANEQUALTO_TOKEN?
  - From a start state, you could choose one of two paths, which indicated which Token you were working on
    - But this is illegal in DFAs



Finite Automata

- Problems with DFAs (cont)
  - Ad hoc solution is to redo into a DFA by sharing common state other than start state
    - But hard to automate from Regular Expressions

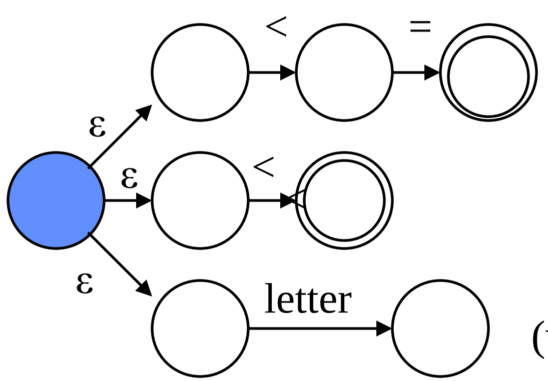


Finite Automata

- Some of these difficulties with DFAs can be overcome by the use of Non-deterministic Finite Automatas (NFAs)
  - Like a DFA, an NFA is defined by 5 elements
    1. An alphabet Σ
    2. A set of states S
    3. A start state s_0
    4. A set of accepting states F
    5. A transition function T: S X Σ
  - Difference is in the transition function
    - It allows ε-transitions
      - Can go from one state to another without consuming input
    - For a given state, can transition to several states on same input character
      - Previous state diagram for <, <= legal NFA

Finite Automata

- The example involving < and <= is a special case of the overall Start State problem which DFAs have
  - Once we have finished recognizing a Token, we go to a state where we are ready to look for next token
  - But we don't know which of the several regular expressions we will start on next
    - Could have a single start state for all Regular Expressions
    - NFAs allow ε-transitions from a super start-state



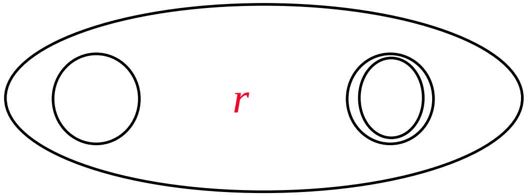
(working on Identifier)

Finite Automata

- That's really an ugly state diagram
  - Why would I want to do one of those?
    - Can you spell "M-I-D-T-E-R-M E-X-A-M" ??
    - Well, there's actually a better (?) reason
      - An automated tool can create an NFA from a Regular Expression
- Big Picture
  - Programmer develops Regular Expressions capturing the grammar of the language
  - Regular Expressions converted into NFAs (automated)
  - NFAs converted into DFAs (automated)
  - DFAs converted into code (automated)

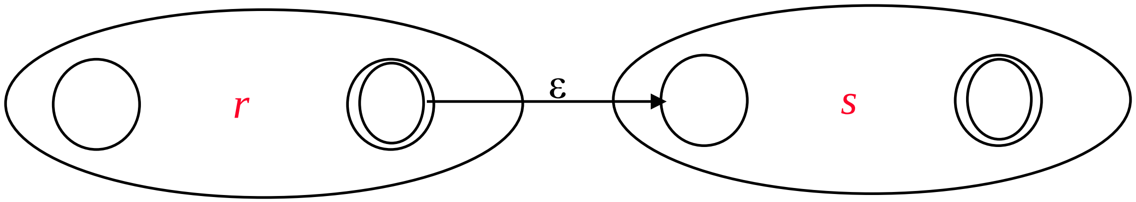
Finite Automata

- First we will look at creating NFAs from Regular Expressions
  - Then we will check out converting NFAs to DFAs
- Known as Thompson's construction
- Based on the recursive nature of Regular Expressions
  - Either a basic character, or the result of choice, concatenation, or closure on basic characters
- For a Regular Expression *r*, we abstract it as an ellipse with start and end states

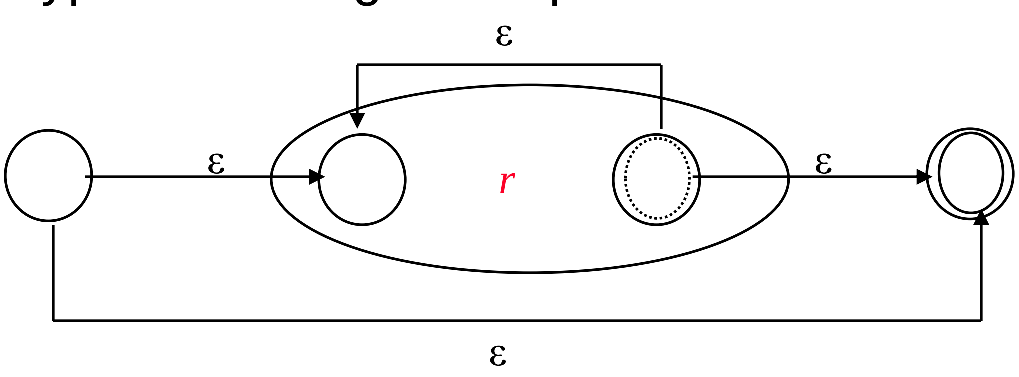


Finite Automata

- We can do concatenation of *r* and *s* by simply connecting 2 ellipses with an ε-transition
  – Middle 2 states sometimes combined

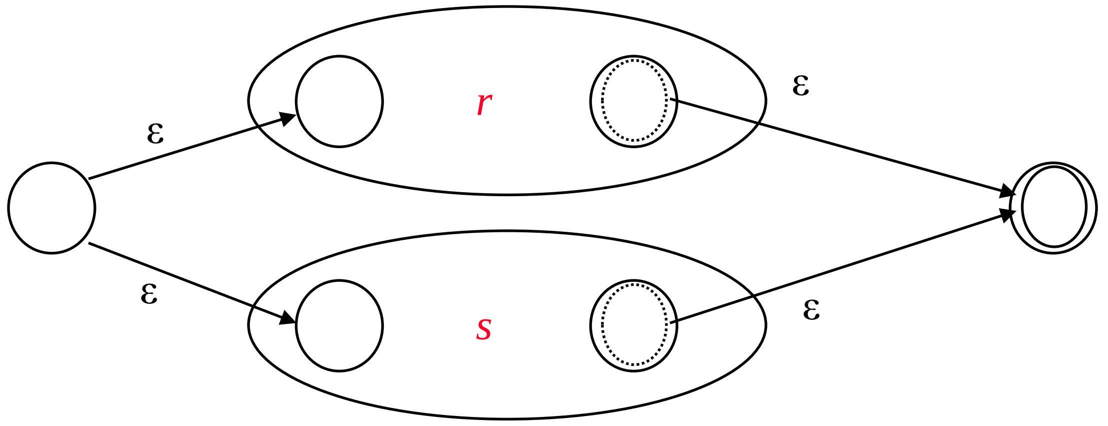


- Closure can be done by adding ε-transitions to both repeat and to bypass the Regular Expression

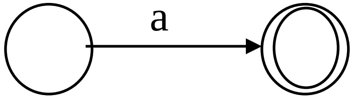


Finite Automata

- Choice can be done easily also



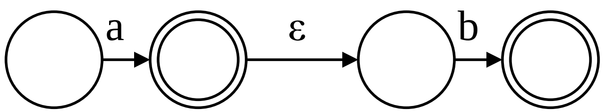
- The last thing we need is the basic NFA for a single character



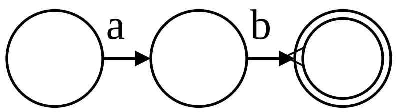
Finite Automata

- Algorithm
  - Start at innermost part of Regular Expression, and recursively build the NFA inside-out

- Example – build an NFA for the Regular Expression ab



- Or simply,



Finite Automata

- Good thing the automated tool is creating all these ugly diagrams!

- So I made a Regular Expression, and a tool converted it into an NFA
  - Now we need the tool to convert it to a DFA
    - Technique is known as subset construction
  - Must convert it to an equivalent DFA, which accepts exactly the same set of strings
  - Our technique must accomplish 2 things
    - Eliminate all ε-transitions
    - Eliminate multiple transitions from a state on same character
  - Foundational principle: each state in DFA will equate to some set of states from the NFA

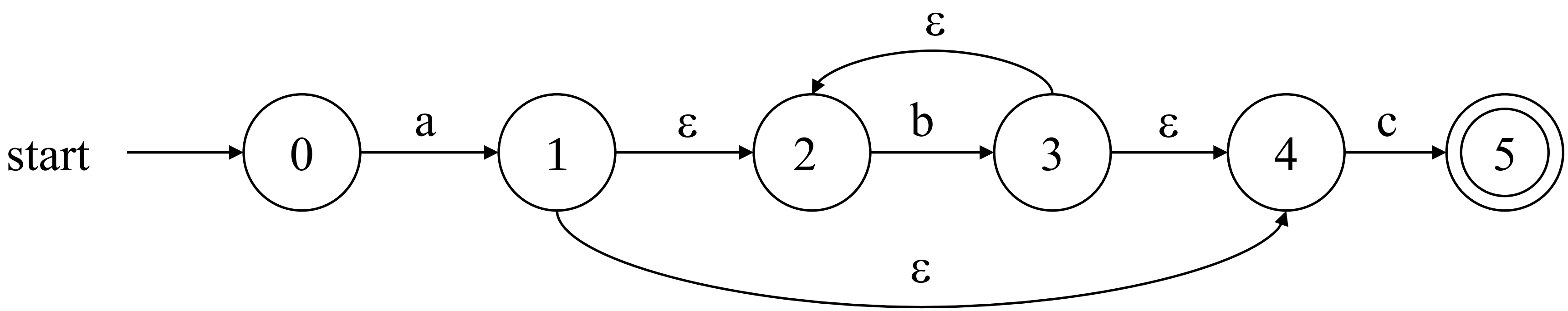
Finite Automata

- First, some definitions -
  - For a state s, we define ε-closure (s) as the set of states reachable from s via zero or more ε-transitions
    - Note ε-closure (s) must contain s
  - For a set of states T, ε-closure (T) is the set of states reachable from any state in T via zero or more ε-transitions
  - For a set of states T and an input a, move (T, a) is the set of states reachable from any state in T via a transition on input a

Finite Automata

- Subset construction
  - Create s_0' (start state of DFA) by taking ε-closure (s_0)
    - It starts off "unvisited"
  - While "unvisited" states of DFA exist
    - Visit one of the unvisited states (T)
      - Mark T "visited"
      - For each input symbol a
        - Compute U= ε-closure (move(T,a))
        - If U not already a state, add as new unmarked state – it should be accepting state if it contains an original accepting state
        - Add transition from T to U on a

Finite Automata



DFA State	NFA Sets of states	a	b	c
A	Close(0) = {0}	1 (B)		
B	Close(1) = {1,2,4}		3 (C)	5 (D)
C	Close(3)={2,3,4}		C	D
D	Close(5)={5}			

Example from Karen Tomko

Finite Automata

- Note that the example of converting letter (letter|digit)* into a DFA did not result in a minimal solution
  - Subset construction, in the worst case, can result in an explosion of states
    - If the NFA has n states, how many states could the DFA possibly have?
      - For each state in DFA, it may/may not contain each state of NFA
  - In practice of most programming languages, subset construction works well
  - However, we would still like to reduce the number of redundant states in the DFA

Finite Automata

- Minimizing the DFA
  - Fortunately, automata theory has proven that for any given NFA, a unique minimum-state DFA exists and can be found using a minimization algorithm
  - General algorithm
    - Initially, partition all states into two sets: non-accepting and accepting – optimistically trying to build 2-state DFA
    - For each set (and any other which must be added)
      - Consider each possible transition from states in the set
      - If all transitions on a go to a state in the accepting set, or all go to a state in the non-accepting set, the set of states defined so far is OK
      - If transitions go to different states, then one of the sets must be split

Scanning

- Ok, back to Big Picture again
  - You create Regular Expressions
  - Tool like Lex
    - Converts Regular Expressions into NFAs
    - Converts NFAs to DFAs
    - Minimizes DFA
    - Converts it to code (maybe using a table-driven approach)
- For your project
  - You create Regular Expressions
  - You convert to DFAs by hand
  - You convert into code by hand
- If you were building a full compiler, you should probably use a tool like Lex

Lex

- Let's look at bit more at Lex
  - You create an input file containing your Regular Expressions
    - Call it something like CMinus.l
  - You run Lex, and it creates a file typically called lex.yy.c
    - This file contains a function yylex() which provides a table-driven scanner
    - In a C++ program, it can be compiled directly and linked with the rest of the code you are developing
    - In the case of java, the output of the Lex-equivalent program is a class which can be instantiated
  - Lex creates the lex.yy.c file by automating the techniques we've looked at

Lex

- Lex input file format
  - 3 sections, split by %%
    - Definitions
      - Includes any preprocessor directives your yy.lex.c should include (like Token definitions)
      - Names for regular expressions
        - digit [0-9]
    - Rules
      - Matches a sequence of Regular Expressions with the code you want executed when found
        - Typically to return a particular Token
    - Any C code you want inserted in this file, or any support routines needed for Rules section

Lex

- Recall that we had several problems with Regular Expressions
  - Ambiguity
    - If input has substrings which match multiple Regular Expressions, Lex will match longest substring
      - whileLoop will be an Identifier, not a keyword
    - If more than one Regular Expression matches longest substring, 1st listed in Lex input file takes priority
      - “else” is listed before Identifier expression, so “else” will return ELSE_TOKEN rather than IDENT_TOKEN
  - Comments
    - Can be handled by ad hoc code in Lex file
    - Match “/*”, then throw away anything until find “*/
- Author’s lex.l on page 537-538
- Author’s comment handling on page 87-88

Scanner Project

- A few words on the Scanner project
  - You create a DFA similar to Fig 2.10 on page 77
  - Note there is a single Done state, and that all successful expressions go to Done state
  - The getToken() routine should
    - Start in Start state
    - Iterate till Done
    - When Done, create Token containing TokenType and TokenData
      - Maybe check for keywords
    - If Error, return TokenType = ERROR_TOKEN
  - Author gives great example in Appendix