From Regular Expressions to Lexical Analyzer

RE → NFA → DFA → minimal DFA → Java code implementing minimal DFA

Michael Wollowski

Rose-Hulman Institute of Technology

Lexical Analyzer

```
//simple example
while (sum < total) {
  sum = sum + x;
}
```

Input

- A stream of bytes
- //simple\bexample\nwhile\b(sum\b<\btotal)\b{\n\tsum\b=\bsum\b+\bx;\n}\n

Output

- A stream of *words*
- [while] [lparen] [id,sum] [lt] [id,total] [rparen] [lbrace] [id,sum] [equals] [id,sum] [plus] [id,x] [semi] [rbrace]
- Notice that some of the words are categorized

Theoretical Basis

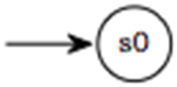
Lexical grammars are typically regular

Regular languages

- Can be recognized by finite automata – FA
- Can be represented as regular expressions

Finite Automata

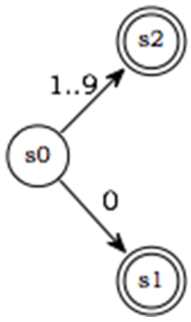
Start state:



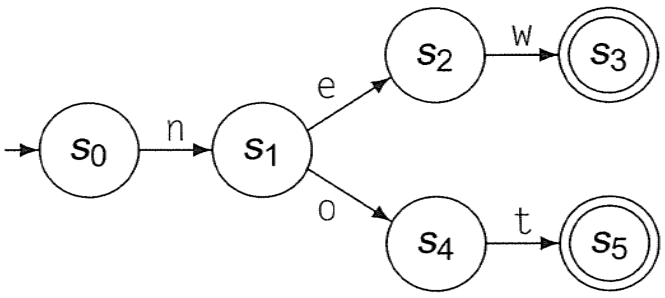
Transition, accepting state:

Finite Automata

Branching:



Example

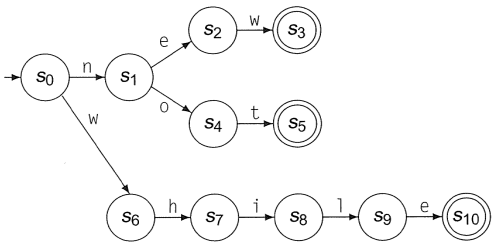


Formal Definition of a FA

A *Finite Automata* (FA) is a 5-tuple (S, Σ, d, s_o, S_A), where:

- S is a finite set of states, along with an error state s_e.
- Σ is a finite alphabet. Typically Σ is the union of the edge labels in the transition diagram.
- $d(s, c)$ is the transition function. It maps each state $s \in S$ and each character $c \in \Sigma$ into some next state. In state s_i, with input character c, the FA takes the transition $s_i \xrightarrow{c} d(s_i, c)$
- $s_o \in S$ is the designated start state.
- S_A is the set of accepting states, $S_A \subseteq S$. Each state in S_A appears as a double circle in the transition diagram.

Example FA



$$S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_e\}$$

$$\Sigma = \{\mathtt{e}, \mathtt{h}, \mathtt{i}, \mathtt{l}, \mathtt{n}, \mathtt{o}, \mathtt{t}, \mathtt{w}\}$$

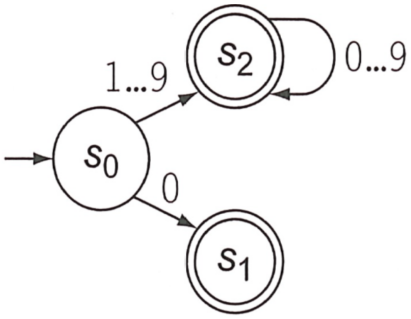
$$\delta = \left\{ \begin{array}{llllll} s_0 \xrightarrow{\mathtt{n}} s_1, & s_0 \xrightarrow{\mathtt{w}} s_6, & s_1 \xrightarrow{\mathtt{e}} s_2, & s_1 \xrightarrow{\mathtt{o}} s_4, & s_2 \xrightarrow{\mathtt{w}} s_3, \\ s_4 \xrightarrow{\mathtt{t}} s_5, & s_6 \xrightarrow{\mathtt{h}} s_7, & s_7 \xrightarrow{\mathtt{i}} s_8, & s_8 \xrightarrow{\mathtt{l}} s_9, & s_9 \xrightarrow{\mathtt{e}} s_{10} \end{array} \right\}$$

$$s_0 = s_0$$

$$S_A = \{s_3, s_5, s_{10}\}$$

A more complex FA

What does the following FA recognize?



Turning the Tables

Draw an FA to recognize

- C++ comments (i.e., // …)
- C comments (i.e., /* … */)

Q1,2 IN GROUPS

Regular Expressions

The set of words accepted by a Finite Automaton, F, forms a language, denoted *L(F)*.

The transition diagram of the FA specifies, in precise detail, that language.

For any FA, we can also describe its language using a notation called regular expression (RE).

Regular expressions are equivalent to FAs.

Regular Expressions

Given an alphabet, ∑, consisting of
- a non-empty, finite set of symbols and
- augmented by ε to represent the empty string

For a given RE, we denote the language that it specifies as *L(r)*.

An RE is built up from three basic operations:
- *Alternation*: The alternation or union of two sets of strings, R and S, denoted R | S, is: { x | x ∈ R or x ∈ S}
- *Concatenation*: The concatenation of two sets of strings, R and S, denoted RS, is: { xy | x ∈ R and y ∈ S }
- *Closure*: The Kleene closure of a set R, denoted R^* is

  $\cup_{i=0..\infty} R^i = \{\varepsilon\} \cup R \cup RR \cup RRR \cup \ldots$

Sometimes, we write R^2 for RR, R^3 for RRR, …

Convenient Shorthand

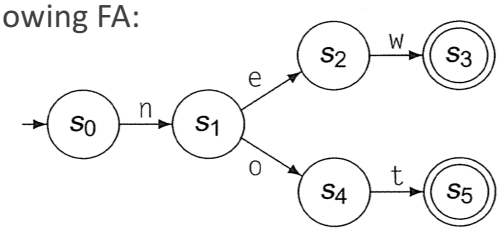
[a-z] denotes set of lowercase letters

[a-zA-Z] denotes upper and lowercase letters

r^+ is shorthand for rr*

Example

Consider the following FA:

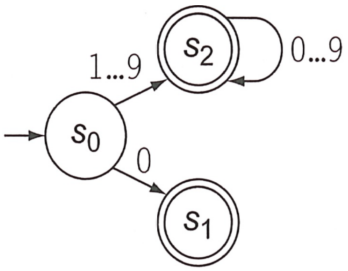


An equivalent RE is:

n (ew | ot)

Example

Consider the following FA:



An equivalent RE is:

(0 | [1-9][0-9]*)

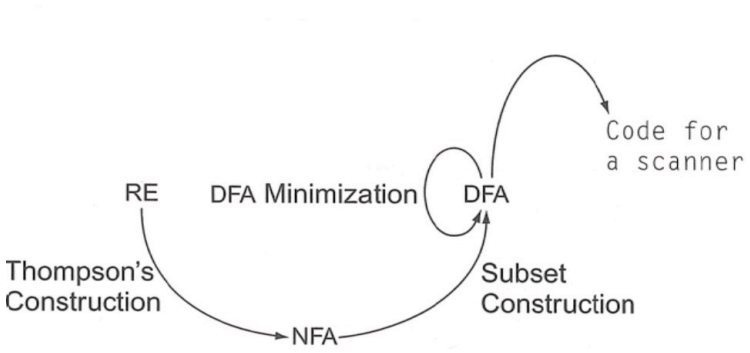
Backus-Naur Form (BNF)

We use BNF to specify MiniJava:

```
Program      ::= (Token | Whitespace)*
Token        ::= ID | Integer | ReservedWord | Operator |
                 Delimiter
ID           ::= Letter (Letter | Digit)*
Letter       ::= a | ... | z | A | ... | Z
Digit        ::= 0 | ... | 9
NonZeroDigit ::= 1 | ... | 9
Integer      ::= NonZeroDigit Digit* | 0
ReservedWord ::= class | public | static | extends | void | int |
                 boolean | if | else | while | return | null |
                 true | false | this | new | String | main |
                 System.out.println
Operator     ::= + | - | * | / | < | <= | >= | > | == | != | &&
                 | || | !
Delimiter    ::= ; | . | , | = | ( | ) | { | } | [ | ]
Whitespace   ::= <space> | <tab> | <newline> | Comment
Comment      ::= ("//" to end of line, "/*...*/" non-nested block
                 comments)
```

From BNF to Scanners

BNF → RE → NFA → DFA → minimal DFA → Java code implementing minimal DFA



From BNF to RE

First step: Create a regular expression based on the specification of MiniJava.

Notice that the BNF is specified through regular expressions.

You have to massage it a bit so that the terminal states correspond to the categories shown in the Lexer Milestone.

In-class Assignment

Give an RE for MiniJava identifiers.

They are defined as follows:

```
ID          ::= Letter (Letter | Digit)*
Letter      ::= a | ... | z | A | ... | Z
Digit       ::= 0 | ... | 9
```

From RE to NFA

We have three operations to produce an NFA, corresponding to the there operations on RE: concatenation, alternation and closure.

We will use what is called *Thompson's construction*.

The construction begins by building trivial NFAs for each character in the input RE.

Next, it applies the transformations for alternation, concatenation and closure to the collection of trivial NFAs in the order dictated by precedence and parentheses.

Parentheses have highest precedence

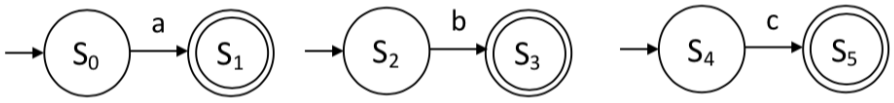
Closure has higher precedence than concatenation

Constructing an NFA for an RE

Let's construct an NFA for the RE:

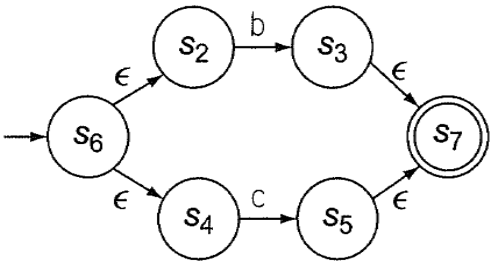
a (b|c)*

We'll begin with constructing NFAs for "a", "b" and "c":



From RE to NFA: Alternation

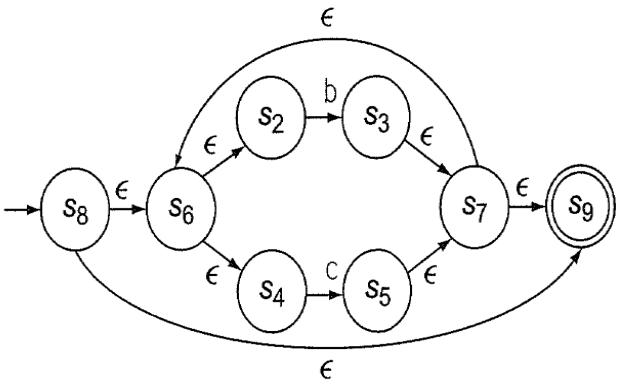
Next, we construct the NFA for b | c



(b) NFA for "*b* | *c*"

a (b|c)*

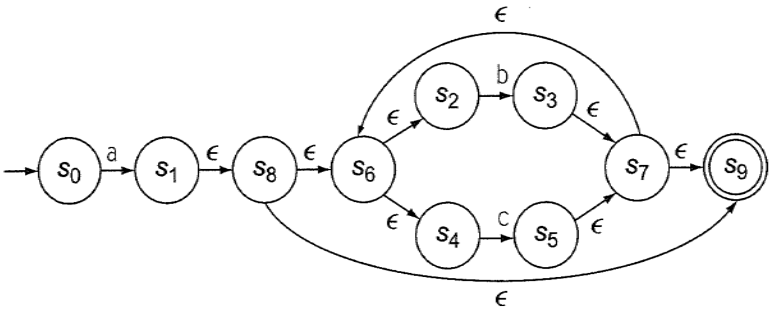
From RE to NFA: Closure



(c) NFA for "$(b \mid c)^*$"

a (b|c)*

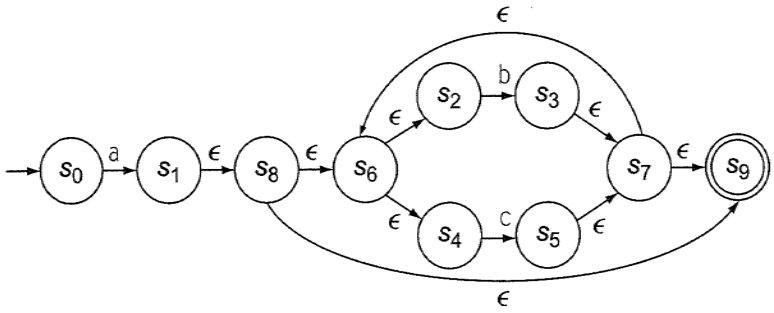
From RE to NFA: Concatenation



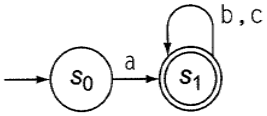
(d) NFA for "$a(b \mid c)^*$"

a (b|c)*

From RE to NFA: Concatenation



(d) NFA for "$a(b \mid c)^*$"



From NFA to DFA

An NFA is inefficient.

We would need to perform backtracking when encountering an empty transitions.

Instead, we will convert the NFA to a DFA, thereby removing any empty transitions.

We will use what is called the *Subset Construction.*

Subset Construction

Take as input an NFA: ($N, \Sigma, \delta_N, n_o, N_A$)

Produce a DFA: ($D, \Sigma, \delta_D, d_o, D_A$)

The NFA and DFA use the same alphabet Σ.

The DFAs start state d_o and accepting states D_A will emerge from the construction.

The complex part of the construction is the derivation of the set of DFA states D from the NFA states N and the derivation of the DFA transition function δ_D.

Subset Construction: Basic Idea

Given an NFA state, find the ε-reachable states

Group these into a single DFA state

Find all the reachable NFA states for each symbol in ∑

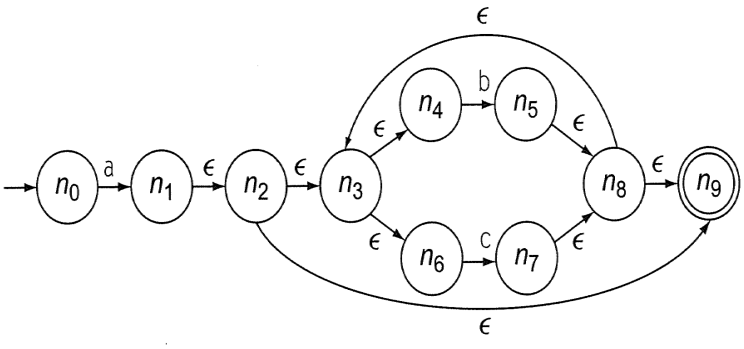
Rinse, lather, repeat

Subset Construction: Algorithm

```
q0 ← ϵ-closure({n0});
Q ← q0;
WorkList ← {q0};

while (WorkList ≠ Ø ) do
    remove q from WorkList;
    for each character c ∈ Σ do
        t ← ϵ-closure(Delta(q,c));
        T[q,c] ← t;
        if t ∉ Q then
            add t to Q and to WorkList;
    end;
end;
```

Example



(a) NFA for "$a(b \mid c)^*$" (With States Renumbered)

Example

Set Name	DFA States	NFA States	ϵ-closure(Delta(q,*))		
			a	b	c
q_0	d_0	n_0	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	– none –	– none –
q_1	d_1	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	– none –	$\{n_5, n_8, n_9, n_3, n_4, n_6\}$	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$
q_2	d_2	$\{n_5, n_8, n_9, n_3, n_4, n_6\}$	– none –	q_2	q_3
q_3	d_3	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$	– none –	q_2	q_3

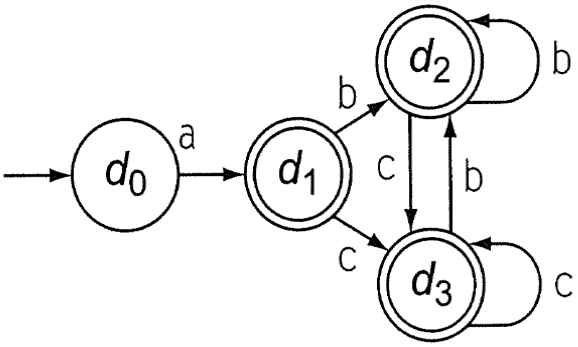
(b) Iterations of the Subset Construction

Example

The algorithm takes the following steps:

1. We initialize q_0 to ε-*closure*($\{n_0\}$), which is just n_0.
2. The first iteration computes ε-*closure*(*Delta*(q_0, *a*)), which contains six NFA states, as well as ε-*closure*(*Delta*(q_0, *b*)) and ε-*closure*(*Delta*(q_0, *c*)), which are empty.
3. The second iteration of the while loop examines q_1. It produces two configurations and names them q_2 and q_3.
4. The third iteration of the while loop examines q_2. It constructs two configurations, which are identical to q_2 and q_3.
5. The fourth iteration of the while loop examines q_3. Like the third iteration, it reconstructs q_2 and q_3.

Example



(a) Resulting DFA

DFA Minimization

A DFA minimization algorithm determines which states are distinct.

States that are not distinct can be merged, to create a simpler DFA.

Several algorithms and variants are known

DFA Minimization

Let M = (Q, Σ, δ, q_0, F) be a DFA.

We define distinctness inductively.

Two states p and q are distinct if

1. $p \in F$ and $q \notin F$ or vice versa, or
2. for some $a \in \Sigma$, $\delta(p, a)$ and $\delta(q, a)$ are distinct

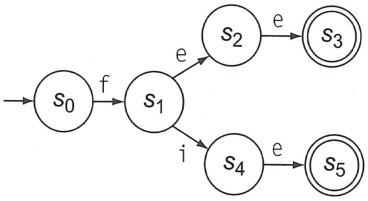
DFA Minimization

The algorithm for determining distinct states is as follows:

1. Create a table with an entry for each pair of states. The entries are initially blank.
2. For every pair of states (p, q):
   - If p is final and q is not, or vice versa, mark the appropriate entry in the table.
3. Loop until no change for an iteration:
   - For each pair of states (p, q) and each character a in the alphabet:
     - If the entry for p and q is blank and the entry for $\delta(p, a)$ and $\delta(q, a)$) is **not** blank, then mark the table entry.
     - If a state does not have a transition for a given symbol, it will remain in that state, when the symbol is presented.
4. Combine all states that are not distinct

DFA Minimization Example

Consider:



(a) DFA for "*fee* | *fie*"

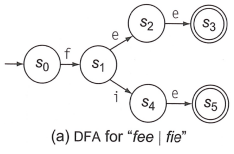
Create:

S_1					
S_2					
S_3					
S_4					
S_5					
	S_0	S_1	S_2	S_3	S_4

DFA Minimization Example

Initialize:

S_1					
S_2					
S_3	X	X	X		
S_4				X	
S_5	X	X	X		X
	S_0	S_1	S_2	S_3	S_4



(a) DFA for "*fee* | *fie*"

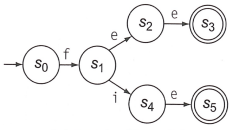
Loop: Pick an entry that is empty and a symbol from the alphabet for which we transition into a marked entry.

- s_1 and s_2 on e: $s_1 \xrightarrow{e} s_2$, $s_2 \xrightarrow{e} s_3$ and s_2 and s_3 are marked.
- Indicated in the table with "E", the character that justifies the entry.
- Similarly for the pair s_1 and s_4 on e

S_1					
S_2		E			
S_3	X	X	X		
S_4		E		X	
S_5	X	X	X		X
	S_0	S_1	S_2	S_3	S_4

DFA Minimization Example

This will give us more entries for our table.
Consider s_0 and s_4 on f: $s_0 \xrightarrow{f} s_1$, $s_4 \xrightarrow{f} s_4$
s_1 and s_4 are marked, so we place an F into s_0 s_4



(a) DFA for "*fee* | *fie*"

Similarly for s_0 s_2

For s_0 s_1 you need to transition on i

The resulting table looks like this:

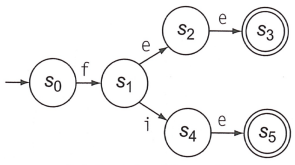
Try to create an entry for s_2 s_4 or s_3 s_5

Can't be done.

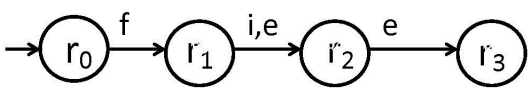
S_1	I				
S_2	F	E			
S_3	X	X	X		
S_4	F	E		X	
S_5	X	X	X		X
	S_0	S_1	S_2	S_3	S_4

DFA Minimization Example

Based on the table, combine states s_2 s_4 and s_3 s_5

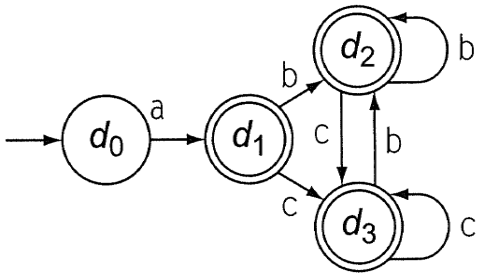


(a) DFA for "*fee* | *fie*"



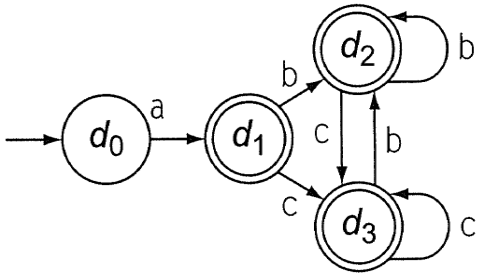
DFA Minimization Class Assignment

Minimize the following DFA

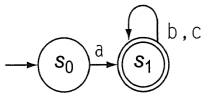


DFA Minimization Class Assignment

Minimize the following DFA



Solution:



Implementing Lexers

Create an RE for each syntactic category.

Convert to DFA

Write code to read a character and simulate the DFA.

When you recognize a word, i.e. end up in an accepting state, return the word and its category.

Do this until EOF.

If your lexer is in a state and there is no transition on the symbol you are reading:

- Roll back to last accepting state encountered and report success
- If there was no accepting state, report failure.

Implementing Lexers

Three approaches:

- Table-driven, uses a transition table
- Direct-coded, similar to above
- Hand-coded

A surprisingly large number of commercial compiler groups use hand-coded lexers.