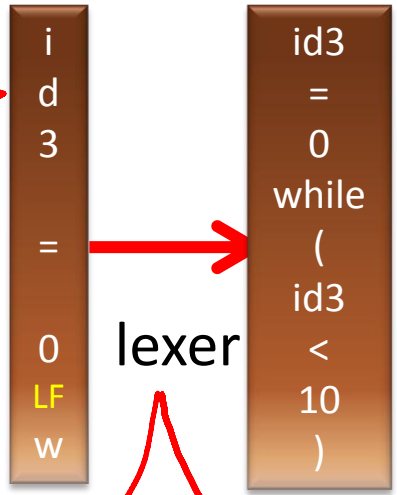
Lexer input and Output

Stream of Char-s
(lazy List[Char])

```
class CharStream(fileName : String){
 val file = new BufferedReader(
     new FileReader(fileName))
  var current : Char = ' '
  var eof : Boolean = false

  def next = {
    if (eof)
     throw EndOfInput("reading" + file)
    val c = file.read()
    eof = (c == -1)
    current = c.asInstanceOf[Char]
  }

  next // init first char
}
```




```
class Lexer(ch : CharStream) {
  var current : Token
  def next : Unit = {
    lexer code goes here
  }
}
```

Stream of **Token**-s

```
sealed abstract class Token
case class ID(content : String) // “id3”
            extends Token
case class IntConst(value : Int) // 10
            extends Token
case class AssignEQ() ‘=‘
            extends Token
case class CompareEQ // ‘==‘
            extends Token
case class MUL() extends Token // ‘*’
case  class PLUS() extends Token // +
case clas LEQ extends Token // ‘<=‘
case class OPAREN extends Token //(
case class CPAREN extends Token //)
...
case class IF extends Token // ‘if’
case class WHILE extends Token
case class EOF extends Token
            // End Of File
```

We have seen how to...

- prove things about languages by induction
- use regular expressions to describe tokens
- compute first symbols of regular expressions
- build lexical analyzers (lexers) to recognize
  - identifiers
  - integer literals

Decision Tree to Map Symbols to Tokens

```
ch.current match {
  case '(' => {current = OPAREN; ch.next; return}
  case ')' => {current = CPAREN; ch.next; return}
  case '+' => {current = PLUS; ch.next; return}
  case '/' => {current = DIV; ch.next; return}
  case '*' => {current = MUL; ch.next; return}
  case '=' => { // more tricky because there can be =, ==
    ch.next
    if (ch.current == '=') {ch.next; current = CompareEQ; return}
    else {current = AssignEQ; return}
  }
  case '<' => { // more tricky because there can be <, <=
    ch.next
    if (ch.current == '=') {ch.next; current = LEQ; return}
    else {current = LESS; return}
  }
}
```

What happens if we omit it?
consider input '<= '

Skipping Comments

```
if (ch.current='/') {
  ch.next
  if (ch.current='/') {
     while (!isEOL && !isEOF) {
       ch.next
     }
  } else {      // what do we set as the current token now?
  }
}
```

Nested comments? /* foo /* bar */ baz */

Longest Match (Maximal Munch) Rule

- There are multiple ways to break input chars into tokens
- Consider language with identifiers - ID, <=, <, =
- Consider these input characters:

```
interpreters <= compilers
```

- These are some ways to analyze it into tokens:

  ID(interpreters) LEQ ID(compilers)

  ID(inter) ID(preters) LESS AssignEQ ID(com) ID(pilers)

  ID(i) ID(nte) ID(rpre) ID(ter) LESS AssignEQ ID(co) ID(mpi) ID(lers)

- This is resolved by **longest match rule**:

If multiple tokens could follow, take the **longest token** possible

Consequences of Longest Match Rule

- Consider language with three operators:

  <, <=, =>

- For sequence '<=>' , lexer will report an error
  - Why?

- In practice, this is not a problem
  - we can always insert extra spaces

Longest Match Exercise

- Recall the maximal munch rule: lexical analyzer should eagerly accept the longest token that it can recognize from the current point.
- Consider the following specification of tokens, the numbers in parentheses gives the name of the token given by the regular expression.

  (1) a(ab)* (2) b*(ac)* (3) cba (4) c+
- Use the maximal munch rule to tokenize the following strings according to the specification
  - c a c c a b a c a c c b a b c
  - c c c a a b a b a c c b a b c c b a b a c
- If we do not use the maximal munch rule, is another tokenization possible?
- Give an example of a regular expression and an input string, where the regular expression is able to split the input strings into tokens, but it is unable to do so if we use the maximal munch rule.

Token Priority

- What if our token classes intersect?
- Longest match rule does not help
- Example: a keyword is also an identifier
- Solution - **priority**: order all tokens,
  if overlap, take one with higher priority

- Example: if it looks both like keyword and like identifier, then it is a keyword (we say so)

Automating Construction of Lexers

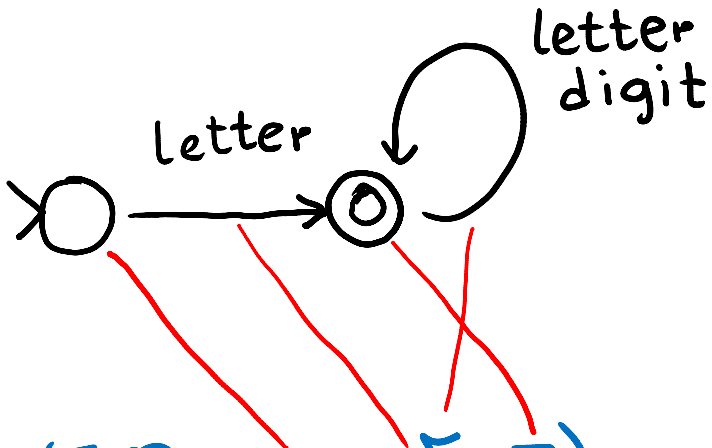
Example in javacc

```
TOKEN: {
   <IDENTIFIER: <LETTER> (<LETTER> | <DIGIT> | "_")* >
  | <INTLITERAL: <DIGIT> (<DIGIT>)* >
  | <LETTER: ["a"-"z"] | ["A"-"Z"]>
  | <DIGIT: ["0"-"9"]>
}
SKIP: {
  " "   |  "\n"   |   "\t"
}
```

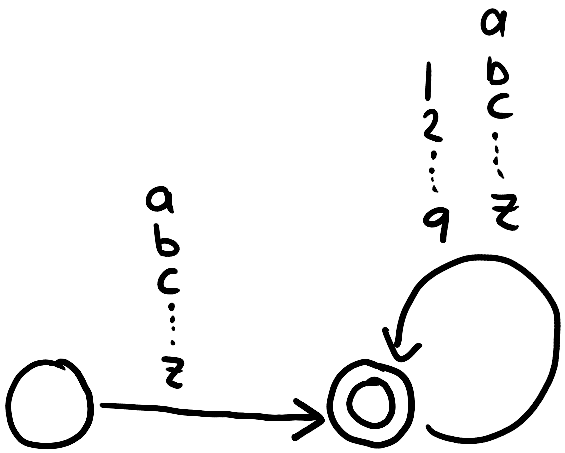
--> get automatically generated code for lexer!

But how does javacc do it?

Finite Automaton (Finite State Machine)



i.e.



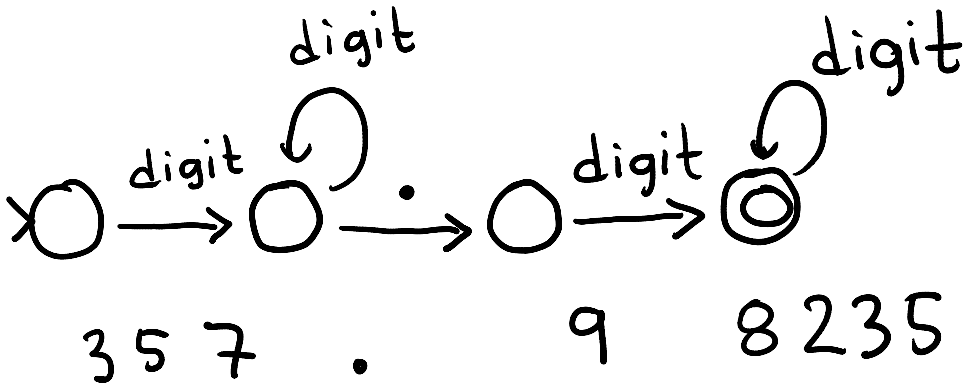
$A = (\Sigma, Q, q_0, \delta, F)$

$\delta \subseteq Q \times \Sigma \times Q$

$(q_1, a, q_2) \in \delta$

- Σ - alphabet
- Q - states (nodes in the graph)
- q_0 - initial state (with '>' sign in drawing)
- δ - transitions (labeled edges in the graph)
- F - final states (double circles)

Numbers with Decimal Point



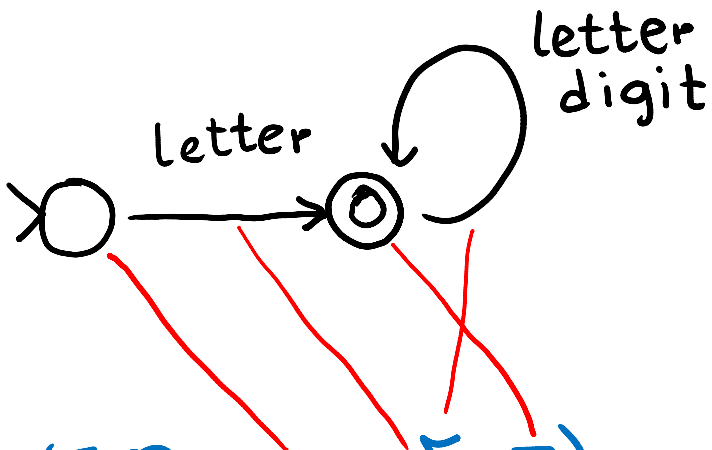
digit digit* . digit digit*

What if the decimal part is optional?

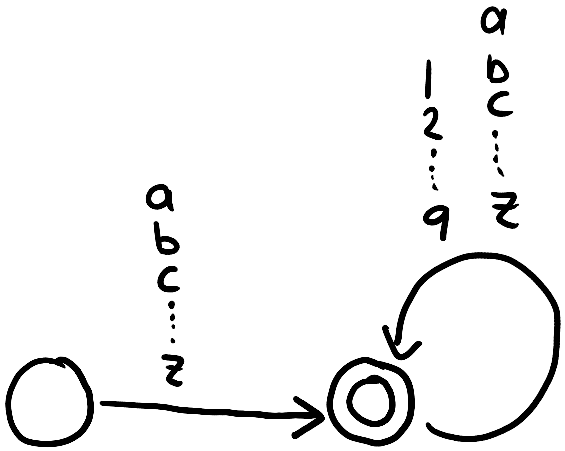
Exercise

- Design a DFA which accepts all the numbers written in binary and divisible by 6. For example your automaton should accept the words 0, 110 (6 decimal) and 10010 (18 decimal).

Kinds of Finite State Automata



i.e.

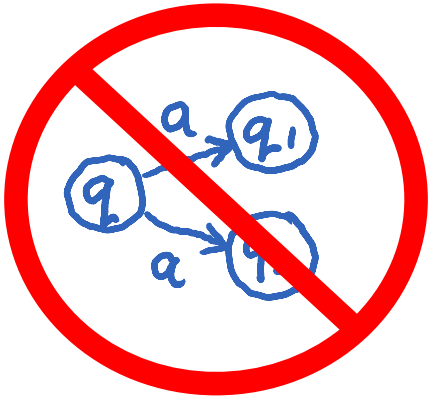


$A = (\Sigma, Q, q_0, \delta, F)$

$\delta \subseteq Q \times \Sigma \times Q$

$(q_1, a, q_2) \in \delta$

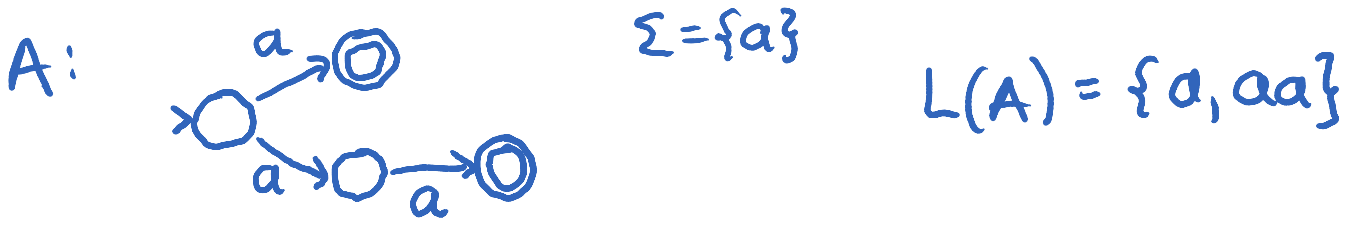
- Deterministic: δ is a function



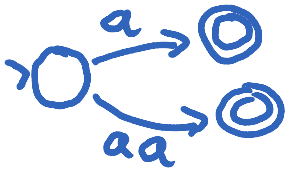
$$\frac{(q, a, q_1) \in \delta \quad (q, a, q_2) \in \delta}{q_1 = q_2}$$

- Otherwise: non-deterministic

Interpretation of Non-Determinism



- For a given word (string), a path in automaton lead to accepting, another to a rejecting state
- Does the automaton accept in such case?
  - yes, if there **exists** an accepting path in the automaton graph whose symbols give that word
- Epsilon transitions: traversing them does not consume anything (empty word)
- More generally, transitions labeled by a word: traversing such transition consumes that entire word at a time



Regular Expressions and Automata

Theorem:

If L is a set of words, then it is a value of a regular expression if and only if it is the set of words accepted by some finite automaton.

Algorithms:

- regular expression → automaton (important!)
- automaton → regular expression (cool)

Recursive Constructions

- Union $r_1 \mid r_2$
- Concatenation $r_1 r_2$
- Star r^*

Eliminating Epsilon Transitions

Exercise: (aa)* | (aaa)*

Construct automaton and eliminate epsilons

Determinization: Subset Construction

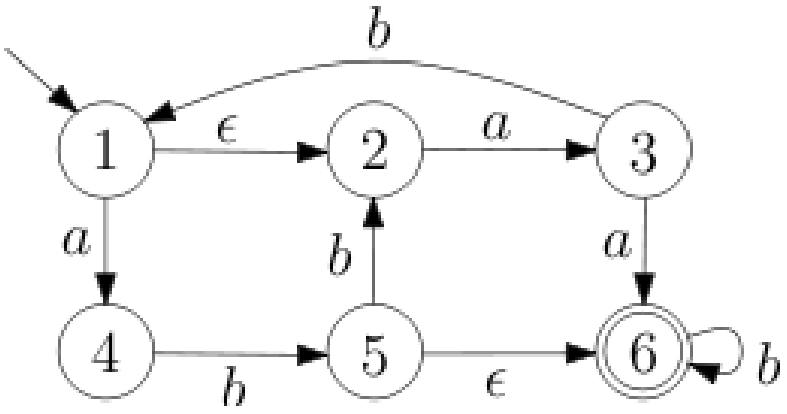
- keep track of a set of all possible states in which automaton could be
- view this finite set as one state of new automaton

- Apply to (aa)* | (aaa)*

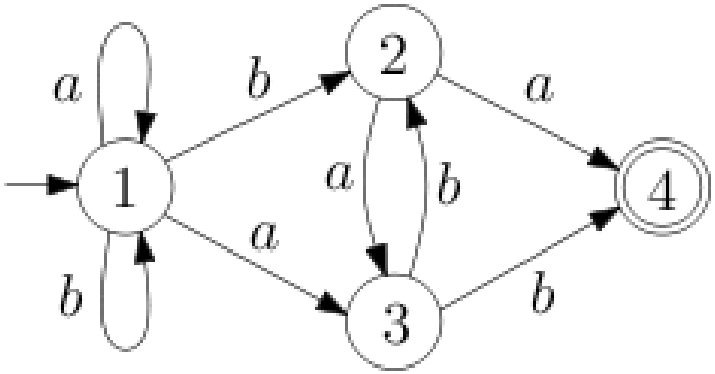
Exercise

Convert the following NFAs to deterministic finite automata.

a)

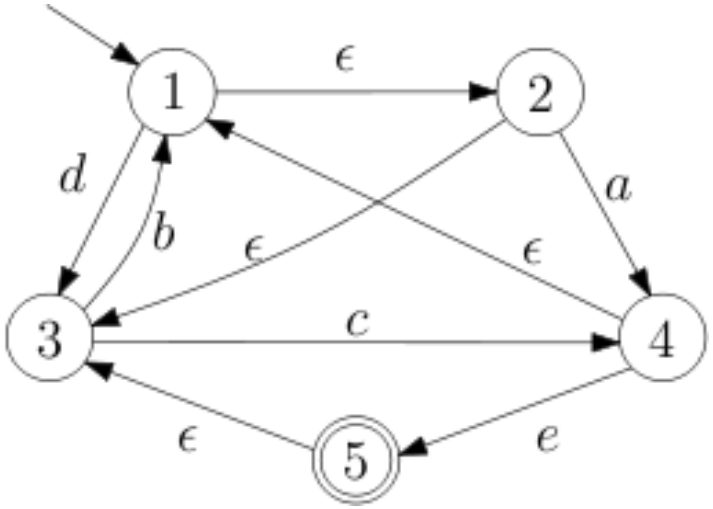


b)



Exercise: first, nullable

- For each of the following languages find the *first* set. Determine if the language is *nullable*.
  - (a|b)*(b|d)((c|a|d)* | a*)
  - language given by automaton:



Minimization: Merge States

- We should only limit the freedom of merge if we have evidence that they behave differently (acceptance, or leading to different states)
- When we run out of evidence, merge the rest
  - merge the states in the previous automaton for (aa)* | (aaa)*

Automated Construction of Lexers

- let r_1, r_2, ..., r_n be regular expressions for token classes
- consider combined regular expression: $(r_1 \mid r_2 \mid ... \mid r_n)^*$
- recursively map a regular expression to a non-deterministic automaton A_1
- eliminate epsilon transitions from A_1 by adding more edges → A_2
- determinize A_2 using the subset construction → A_3
- minimize A_3 to reduce its size → A_4
- the result only checks that input can be split into tokens, does not say how!
- so, must maintain different final states for different tokens
- and, must remember **last** accepting state to know when to return token and to implement the longest match rule
- we can implement an automaton using a big array or a map for transitions

Making $(r_1|r_2|\ldots|r_n)$ work as Lexer

- Modify state machine
- for each accepting state specify the token recognized (but do not stop)
  - use token class priority if multiple options
- for longest match rule: remember last token and input position for last accepted state
- when no accepting state can be reached (effectively: when we are in a trap state)
  - revert position to last accepted state
  - return last accepted token

Exercise: Build Lexical Analyzer Part

For these two tokens, using longest match,

where first has the priority:

binaryToken ::= (**z**|1)*

ternaryToken ::= (0|1|2)*

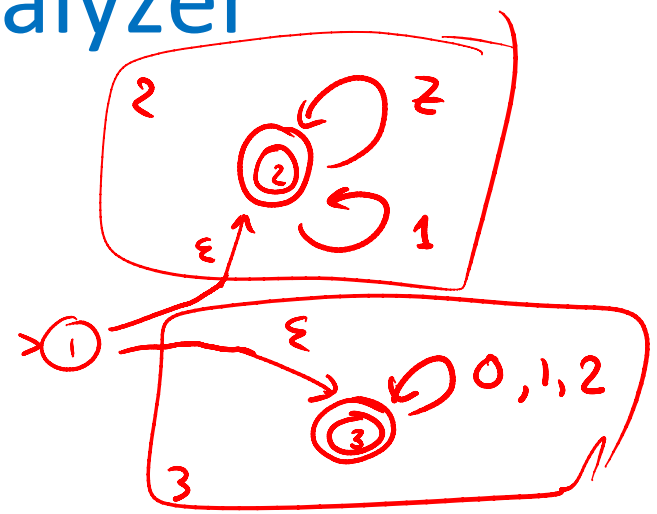
1111z1021z1 →

Lexical Analyzer

1) binaryToken ::= (**z**|1)*
2) ternaryToken ::= (0|1|2)*

1111z1021z1 →

binary tern binary

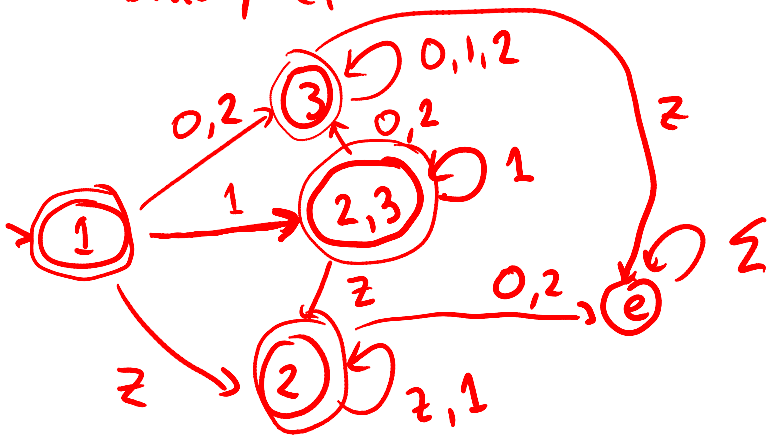


(binaryToken | ternaryToken)*

Σ = {0,1,2,z}

1 1 1 1 1 5

binary (priority)



Exercise: Realistic Integer Literals

- Integer literals are in three forms in Scala: decimal, hexadecimal and octal. The compiler discriminates different classes from their beginning.
  - Decimal integers are started with a non-zero digit.
  - Hexadecimal numbers begin with 0x or 0X and may contain the digits from 0 through 9 as well as upper or lowercase digits A to F afterwards.
  - If the integer number starts with zero, it is in octal representation so it can contain only digits 0 through 7.
  - l or L at the end of the literal shows the number is Long.
- Draw a single DFA that accepts all the allowable integer literals.
- Write the corresponding regular expression.

Exercise

- Let L be the language of strings A = {<, =} defined by regexp (<|=| <====*), that is, L contains <,=, and words $<=^n$ for n>2.
- Construct a DFA that accepts L
- Describe how the lexical analyzer will tokenize the following inputs.

  1) <=====

  2) ==<==<==<==<==

  3) <======<

More Questions

- Find automaton or regular expression for:
  - Sequence of open and closed parentheses of even length?
  - as many digits before as after decimal point?
  - Sequence of balanced parentheses
    ((()) ()) - balanced
    ( )) (( ) - not balanced
  - Comment as a sequence of space,LF,TAB, and comments from // until LF
  - Nested comments like /* ... /* */ ... */

Automaton that Claims to Recognize $\{ a^n b^n \mid n >= 0 \}$

We can make it deterministic

Let the result have K states

Feed it a, aa, aaa, ….

consider the states it ends up in

Limitations of Regular Languages

- Every automaton can be made deterministic
- Automaton has finite memory, cannot count
- Deterministic automaton from a given state behaves always the same
- If a string is too long, deterministic automaton will repeat its behavior
  - say A accepted $a^n b^n$ for all n, and has K states

Context-Free Grammars

- Σ - terminals
- Symbols with recursive defs - nonterminals
- Rules are of form

  N ::= v

  v is sequence of terminals and non-terminals
- Derivation starts from a starting symbol
- Replaces non-terminals with right hand side
  - terminals and
  - non-terminals

Balanced Parentheses Grammar

- Sequence of balanced parentheses

  ((()) ()) - balanced

  ( )) (( ) - not balanced

Remember While Syntax

program ::= statmt*

statmt ::= println(stringConst , ident)

| ident = expr

| if (expr) statmt (else statmt)?

| while (expr) statmt

| { statmt* }

expr ::= intLiteral | ident

| expr (&& | < | == | + | - | * | / | %) expr

| ! expr | - expr

Eliminating Additional Notation

- Grouping alternatives

  s ::= P | Q instead of s ::= P
  s ::= Q

- Parenthesis notation

  expr (&& | < | == | + | - | * | / | %) expr

- Kleene star within grammars

  { statmt* }

- Optional parts

  if (expr) statmt (else statmt)$^{?}$