Language

Definition

A *language* over alphabet A is a set $L \subseteq A^*$. Example:

- a finite language like $L = \{1, 10, 1001\}$ or the empty language $\emptyset$
- infinite but very difficult to describe (there are random languages: there exist more languages as subsets of A^* than there are finite descriptions)
- infinite but having some nice structure, where words follow a certain "pattern" that we can describe precisely and check efficiently $\longleftarrow$ these are our focus

$L_2 = \{01, 0101, 010101, \ldots\}$ = those non-empty words that are of the form $01 \ldots 01$ where the block 01 is repeated some finite positive number of times. Using notation $(01)^n$ for a word consisting of block 01 repeated n times, we can write $L_2 = \{(01)^n \mid n \geq 1\}$.

Languages are sets, so we can take their union ($\cup$), intersection ($\cap$), and apply other set operations on languages.

Languages $\emptyset$ and $\{\varepsilon\}$ are very different: $\emptyset$ is a set that contains no words, whereas $\{\varepsilon\}$ contains precisely one word, the word of length zero.

Concatenating Languages

In addition to operations such as intersection and union that apply to sets in general, languages support additional operations, which we can define because their elements are words. The first one translates concatenation of words to sets of words, as follows.

Definition (Language concatenation)

Given $L_1 \subseteq A^*$ and $L_2 \subseteq A^*$, define $L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$

The definition above states that $w \in L_1 L_2$ if and only if there is a way to split w into two words w_1 and w_2, so that $w = w_1 w_2$ and such that $w_1 \in L_1$ and $w_2 \in L_2$.

Definition (Language exponentiation)

Given $L \subseteq A^*$, define

$$L^0 = \{\varepsilon\}$$
$$L^{n+1} = L \cdot L^n$$

Theorem

Given $L \subseteq A^*$, $L^n = \{w_1 \dots w_n \mid w_1, \dots, w_n \in L\}$

Expanding the Definition

If L is an arbitrary language, compute each of the following:

- $L\emptyset$
- $\emptyset L$
- $L\{\varepsilon\}$
- $\{\varepsilon\}L$
- $\emptyset\{\varepsilon\}$
- LL
- $\{\varepsilon\}^n$

Expanding the Definition

If L is an arbitrary language, compute each of the following:

- $L\emptyset$
- $\emptyset L$
- $L\{\varepsilon\}$
- $\{\varepsilon\}L$
- $\emptyset\{\varepsilon\}$
- LL
- $\{\varepsilon\}^n$

Note the difference between:

- the empty language $\emptyset$, which contains no words
- the language $\{\varepsilon\}$, which contains exactly one word, ε

Concatenation of Languages

Let A be alphabet. Consider the set of all languages $L \subseteq A^*$

Is this a monoid?

Concatenation of Languages

Let A be alphabet. Consider the set of all languages $L \subseteq A^*$

Is this a monoid?

Does the cancelation law hold?

Representing Languages in Programs

In general not possible: formal languages need not be recursively enumerable sets.
A reasonably powerful representation: computable characteristic function.
As for any subset of some fixed set, a language $L \subseteq A^*$ is given by its characteristic function $f_A : A^* \to \{0,1\}$ defined by $f_A(w) = 1$ for $w \in L$ and $f_A(w) = 0$ for $w \notin L$.
Here we use the `contains` field as the characteristic function and build the language $L_2 = \{(01)^n \,|\geq 0\}$.

```
case class Lang[A](contains: List[A] -> Boolean)
def f(w: List[Int]): Boolean = w match {
  case Cons(0, Cons(1, Nil())) ⇒ true
  case Cons(0, Cons(1, wRest)) ⇒ f(wRest)
  case _ ⇒ false
}
val L2 = Language(f)
val test = L2.contains(0 :: 1 :: 0 :: 1 :: Nil())
```

Representing Language Concatenation

W can use code to express concatenation of computable languages.

```
def concat(L1: Lang[A], L2: Lang[A]): Lang[A]= {
  def checkFrom(i: BigInt, len: BigInt) = {
    require(0 <= i && i <= len)
    (L1.contains(w.slice(0, i)) && L2.contains(w.slice(i, len)) ||
    (i < len && checkFrom(i + 1, len))
  }
  def f(w: List[A]) = checkFrom(0, w.length)
  Lang(f) // return the language whose characteristic function is f
}
```

Repetition of a Language: Kleene Star

Definition (Kleene star)

Given $L \subseteq A^*$, define

$$L^* = \bigcup_{n \geq 0} L^n$$

Theorem

For $L \subseteq A^*$, for every $w \in A^*$ we have $w \in L^*$ if and only if

$$\exists n \geq 0. \exists w_1, \ldots, w_n \in L.\ w = w_1 \ldots w_n$$

$\{a\}^* = \{\varepsilon, a, aa, aaa, \ldots\}$
$\{a, bb\}^* = \{\varepsilon, a, bb, abb, bba, aa, bbbb, aabb, \ldots\}$ (describe this language)
Can L^* be finite for some L?

Star and the Empty Word

Because concatenating with an empty word has no effect, we have the following:

$$L^* = \{\varepsilon\} \cup \big(L \setminus \{\varepsilon\}\big)^*$$

Equivalently: $w \in L^*$ if and only if either $w = \varepsilon$ or, for some n where $1 \leq n \leq |w|$,

$$w = w_1 \ldots w_n$$

where $w_i \in L$ and $|w_i| \geq 1$ for all i where $1 \leq i \leq n$.

Star and the Empty Word

Because concatenating with an empty word has no effect, we have the following:

$$L^* = \{\varepsilon\} \cup \big(L \setminus \{\varepsilon\}\big)^*$$

Equivalently: $w \in L^*$ if and only if either $w = \varepsilon$ or, for some n where $1 \leq n \leq |w|$,

$$w = w_1 \dots w_n$$

where $w_i \in L$ and $|w_i| \geq 1$ for all i where $1 \leq i \leq n$.

If L is computable (has a computable characterstic function), is L^* also computable?

Star and the Empty Word

Because concatenating with an empty word has no effect, we have the following:

$$L^* = \{\varepsilon\} \cup \left(L \setminus \{\varepsilon\}\right)^*$$

Equivalently: $w \in L^*$ if and only if either $w = \varepsilon$ or, for some n where $1 \leq n \leq |w|$,

$$w = w_1 \ldots w_n$$

where $w_i \in L$ and $|w_i| \geq 1$ for all i where $1 \leq i \leq n$.

If L is computable (has a computable characterstic function), is L^* also computable?

- try all possible ways of splitting w (but there are better ways)

Starring: $\{a, ab\}$

Let $A = \{a, b\}$ and $L = \{a, ab\}$.
Come up with a property "..." that describes the language L^*:

$$L^* = \{w \in A^* \mid \ldots\}$$

Prove that the property and L^* denote the same language.

Further Examples

Let A = {a,b}
Let L = {a,ab}

L L = { aa, aab, aba, abab }

compute LLL

L* = {ε, a, ab, aa, aab, aba, abab, aaa, ... }

Is bb inside L* ?

Further Examples

Let $A = \{a,b\}$

Let $L = \{a,ab\}$

$L\,L = \{ aa, aab, aba, abab \}$

compute LLL

$L^* = \{\varepsilon, a, ab, aa, aab, aba, abab, aaa, \ldots \}$

Is bb inside L^* ?

Question: Is it the case that

$L^*=\{ w \mid$ immediately left of each **b** is an **a** $\}$

If yes, prove it. If no, give a counterexample.

Precise Statement and Proof

Reminder: $L^* = \{ w_1 \dots w_n \mid n \geq 0, w_1 \dots w_n \in L \}$

Claim: $\{a,ab\}^* = S$ where

$S = \{w \in \{a,b\}^* \mid \forall 0 \leq i < |w|.\ \text{if } w_{(i)} = b \text{ then: } i > 0 \text{ and } w_{(i-1)} = a\}$

Proof. We show **1)** $\{a,ab\}^* \subseteq S$ and **2)** $S \subseteq \{a,ab\}^*$.

1) $\{a,ab\}^* \subseteq S$: We show: for all *n*, $\{a,ab\}^n \subseteq S$, by induction on *n*

- Base case, n=0. $\{a,ab\}^0 = \{\varepsilon\}$, so $i<|w|$ is always false and '->' is true.

- Suppose $\{a,ab\}^n \subseteq S$. Showing $\{a,ab\}^{n+1} \subseteq S$. Let $w \in \{a,ab\}^{n+1}$.
Then $w = vw'$ where $w' \in \{a,ab\}^n$, $v \in \{a,ab\}$. Let $i < |w|$ and $w_{(i)} = b$.
$v_{(0)} = a$, so $w_{(0)} = a$ and thus $w_{(0)} \mathrel{!}= b$. Therefore $i > 0$. Two cases:
1.1) v=a. Then $w_{(i)} = w'_{(i-1)}$. By I.H. $i-1>0$ and $w'_{(i-2)} = a$. Thus $w_{(i-1)} = a$.
1.2) v=ab. If i=1, then $w_{(i-1)} = w_{(0)} = a$, as needed. Else, i>1 so
$w'_{(i-2)} = b$ and by I.H. $w'_{(i-3)} = a$. Thus $w_{(i-1)} = (vw')_{(i-1)} = w'_{(i-3)} = a$.

Proof Continued

recall: $S = \{w \in \{a,b\}^* \mid \forall 0 \leq i < |w|.\ \text{if } w_{(i)} = b \text{ then: } i > 0 \text{ and } w_{(i-1)} = a\}$

For the second direction, we first prove:

(*) If $w \in S$ and $w = w'v$ then $w' \in S$.

Proof of (*): Let $i < |w'|$, $w'_{(i)} = b$. Then $w_{(i)} = b$ so $w_{(i-1)} = a$ and thus $w'_{(i-1)} = a$.

2) $S \subseteq \{a,ab\}^*$. We prove, by induction on n, that for all n,

for all w, if $w \in S$ and $n = |w|$ then $w \in \{a,ab\}^*$.

- Base case: n=0. Then w is empty string and thus in $\{a,ab\}^*$.
- Let n>0. Suppose property holds for all k < n. Let $w \in S$, $|w| = n$.

There are two cases, depending on the last letter of w.

2.1) $w = w'a$. Then $w' \in S$ by **(*)**, so by IH $w' \in \{a,ab\}^*$, so $w \in \{a,ab\}^*$.

2.2) $w = vb$. By $w \in S$, $w_{(|w|-2)} = a$, so $w = w'ab$. By **(*)**, $w' \in S$, by IH $w' \in \{a,ab\}^*$, so $w \in \{a,ab\}^*$. In any case, $w \in \{a,ab\}^*$. We proved the entire equality.

Regular Expressions

Regular Expressions

One way to denote (often infinite) languages

Regular expression = expression built only from:

- empty language $\varnothing$ (empty set of words)
- $\{\varepsilon\}$, denoted just ε (set containing the empty word)
- $\{a\}$ for $a \in A$, denoted simply by a
- union of sets of words, denoted | (some use +)
- concatenation of sets of words (dot, or not written)
- Kleene star * (repetition)

- Example: letter (letter | digit)*
  (letter,digit are shorthand sets from before)

Kleene (from Wikipedia)

Stephen Cole Kleene (January 5, 1909, Hartford, Connecticut, United States – January 25, 1994, Madison, Wisconsin) was an American mathematician who helped lay the foundations for theoretical computer science. One of many distinguished students of Alonzo Church, Kleene, along with Alan Turing, Emil Post, and others, is best known as a founder of the branch of mathematical logic known as recursion theory. Kleene's work grounds the study of which functions are computable. A number of mathematical concepts are named after him: Kleene hierarchy, Kleene algebra, the Kleene star (**Kleene closure**), Kleene's recursion theorem and the Kleene fixpoint theorem. He also **invented regular expressions**, and was a leading American advocate of mathematical intuitionism.

Regular Expressions

- Regular expressions are just a notation for some particular operations on languages

  letter (letter | digit)*

- Denotes the set

  $$\text{letter}\ (\text{letter} \cup \text{digit})^*$$

- Each **finite** language $\{w_1, \ldots, w_n\}$ can be described using regular expression $(w_1 \mid \ldots \mid w_n)$ but we can also describe many infinite languages.

Some Regular Expression Operators that can be Defined in Terms of Previous Ones

- [a..z] = a|b|...|z (use ASCII ordering)
  (also other shorthands for finite languages)
- $e^?$ (optional expression)
- e^+ (repeat at least once)
- $e^{k..*} = e^k e^*$ $e^{p..q} = e^p (\varepsilon|e)^{q-p}$
- complement: !e (A* \ e) -non-obvious, use automata
- intersection: e1 & e2 (e1 ∩ e2) = ! (!e1 | !e2)

Monadic Second-Order Logic

(Advanced)

- Quantification: we can also allow expressions with ∀ and "Monadic Second-Order Logic of Strings"

For example, the statement:

$$\{a,ab\}^* = \{w \in \{a,b\}^* \mid \forall i.\ w_{(i)}{=}b \rightarrow i > 0 \ \& \ w_{(i-1)}{=}a\}$$

can be proven automatically using tools such as:
http://www.brics.dk/mona/

Lexical Analysis

Lexical Analysis

res = 14 + arg * 3 (character stream)

Lexer gives:

“res”, “=”, “14”, “+”, “arg”, “*”, “3” (token strem)

Lexical analyzer (lexer, scanner, tokenizer) is often specified using regular expressions for each kind of token
It groups characters into tokens, maps stream to stream

- A simple lexer could represent all tokens as strings
- For efficiency and convenience we represent tokens using more structured data types

Lexical Analyzer - Key Ideas

Typically needs only *small* amount of *memory*.

It is *not difficult* to construct a lexical analyzer manually

For such lexers, we use the first character to decide on token class: first(L) = { a | aw in L }

We use *longest match rule*: lexical analyzer should eagerly accept the **longest** token that it can recognize from this point, even if this means that later characters will not form valid token.

It is possible to *automate* the construction of lexical analyzers, using a conversion of regular expressions to automata.

Tools that automate this construction are part of compiler-compilers, such as JavaCC described in the “Tiger book”.

While Language – A Program

```
num = 13;
while (num > 1) {
  println("num = ", num);
  if (num % 2 == 0) {
    num = num / 2;
  } else {
    num = 3 * num + 1;
  }
}
```

Tokens (Words) of the *While* Language

Ident ::=

letter (letter | digit)*

integerConst ::= digit digit*

regular
expressions

keywords

if else while println

special symbols

( ) && < == + - * / % ! -{ } ; ,

letter ::= a | b | c | … | z | A | B | C | … | Z

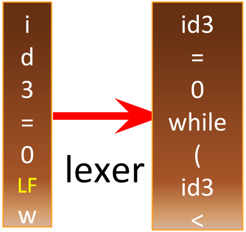
digit ::= 0 | 1 | … | 8 | 9

Manually Constructing Lexers by example

Stream of **Char**-s:

Stream of **Token**-s

```
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
}
```




```
sealed abstract class Token
case class ID(content : String) // "id3"
        extends Token
case class IntConst(value : Int) // 10
        extends Token
case object AssignEQ extends Token
case object CompareEQ
        extends Token
case object MUL extends Token // *
case  object PLUS extends Token //+
case object LEQ extends Token //'<='
case object OPAREN extends Token
case class CPAREN extends Token
case object IF extends Token
case object WHILE extends Token
case object EOF extends Token
        // End Of File
```

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

Recognizing Identifiers and Keywords

```
if (isLetter) {
  b = new StringBuffer
  while (isLetter || isDigit) {
     b.append(ch.current)
     ch.next
  }
 keywords.lookup(b.toString) {
  case None=> token=ID(b.toString)
  case Some(kw) => token=kw
 }
}
```

regular expression for identifiers:

letter (letter|digit)*

Keywords look like identifiers, but are simply indicated as keywords in language definition. Introduce a constant Map from strings to keyword tokens. If not in map, then it is ordinary identifier.

Integer Constants and Their Value

regular expression for integers:
digit digit*

```
if (isDigit) {
  k = 0
  while (isDigit) {
    k = 10*k + toDigit(ch.current)
    ch.next
  }
  token = IntConst(k)
}
```

Deciding which Token is Coming

- How do we know when we are supposed to analyze string, when integer sequence etc?
- Manual construction: use **lookahead** (next symbol in stream) to decide on token class
- compute first(e) - symbols with which e can start
- check in which first(e) current token is
- If $L \subseteq A^*$ is a language, then first(L) is set of all alphabet symbols that start some word in L

$$\text{first}(L) = \{a \in A \mid \exists v \in A^* .\ a\,v \in L\}$$

First Symbols of a Set of Words

first({a, bb, ab}) = {a,b}

first({a, ab}) = {a}

first({aaaaaaa}) = {a}

first({a}) = {a}

first({}) = {}

first({ε}) = {}

first({ε,ba}) = {b}

first of a regexp

- Given regular expression e, how to compute first(e)?
  - use automata (we will see this later)
  - rules that directly compute them (also work for grammars, we will see them for parsing) - now
- Examples of first(e) computation:
  - first(ab*) = {a}
  - first(ab*|c) = {a,c}
  - first(a*b*c) = {a,b,c}
  - first((cb|a*c*)d*e)) =
- Notion of nullable(r) - whether empty string belongs to the regular language.

Computing 'nullable' for regular expressions

If e is regular expression (its syntax tree), then $L(e)$ is the language denoted by it.
For $L \subseteq A^*$ we defined $nullable(L)$ as $\varepsilon \in L$
If e is a regular expression, we can compute $nullable(e)$ to be equal to $nullable(L(e))$, as follows:

$$\begin{array}{rcl}
nullable(\emptyset) & = & false \\
nullable(\varepsilon) & = & true \\
nullable(a) & = & false \\
nullable(e_1|e_2) & = & nullable(e_1) \lor nullable(e_2) \\
nullable(e^*) & = & true \\
nullable(e_1 e_2) & = & nullable(e_1) \land nullable(e_2)
\end{array}$$

Computing 'first' for regular expressions

For $L \subseteq A^*$ we defined: $\mathit{first}(L) = \{a \in A \mid \exists v \in A^*.\ av \in L\}$.
If e is a regular expression, we can compute $\mathit{first}(e)$ to be equal to $\mathit{first}(L(e))$, as follows:

$$
\begin{array}{rcl}
\mathit{first}(\emptyset) & = & \emptyset \\
\mathit{first}(\varepsilon) & = & \emptyset \\
\mathit{first}(a) & = & \{a\},\ \text{for } a \in A \\
\mathit{first}(e_1|e_2) & = & \mathit{first}(e_1) \cup \mathit{first}(e_2) \\
\mathit{first}(e*) & = & \mathit{first}(e) \\
\mathit{first}(e_1 e_2) & = & \mathit{if}\,(\mathit{nullable}(e_1))\ \mathit{then}\ \mathit{first}(e_1) \cup \mathit{first}(e_2) \\
 & & \mathit{else}\ \mathit{first}(e_1)
\end{array}
$$

Clarification for first of concatenation

Let e be $\mathbf{a}^*\mathbf{b}$. Then $L(e) = \{b, ab, aab, aaab, \ldots\}$
$\mathit{first}(L(e)) = \{a, b\}$

$e = e_1 e_2$ where $e_1 = a^*$ and $e_2 = b$. Thus, $\mathit{nullable}(e_1)$.

$$\mathit{first}(e_1 e_2) = \mathit{first}(e_1) \cup \mathit{first}(e_2) = \{a\} \cup \{b\} = \{a, b\}$$

It is *not correct* to use $\mathit{first}(e) =^? \ \mathit{first}(e_1) = \{a\}$.
Nor is it correct to use $\mathit{first}(e) =^? \ \mathit{first}(e_2) = \{b\}$.
We must use their union.

Converting Simple Regular Expresssions into a Lexer

regular expression	*lexercode*
$a \quad (a \in A)$	$\mathit{if}\ (\mathit{current} = a)\ \mathit{next}\ \mathit{else}\ \ldots$
$r_1 r_2$	$\mathit{code}(r_1);\ \mathit{code}(r_2)$
$r_1 \vert r_2$	$\mathit{if}\ (\mathit{current} \in \mathit{first}(r_1))$ $\quad \mathit{code}(r_1)$ $\mathit{else}\ \mathit{code}(r_2)$
$r*$	$\mathit{while}\ (\mathit{current} \in \mathit{first}(r))$ $\quad \mathit{code}(r)$

More complex cases

In other cases, a few upcoming characters ("lookahead") are not sufficient to determine which token is coming up.

Examples:
A language might have separate numeric literal tokens to simplify type checking:

- integer constants: *digit digit**
- floating point constants: *digit digit** . *digit digit**

Floating point constants must contain a period (e.g., Modula-2).

Division sign begins with same character as // comments.
Equality can begin several different tokens.

In such cases, we process characters and store them until we have enough information to make the decision on the current token.

Example of a part of a lexical analyzer

```
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}
  }
}
```

Example of a part of a lexical analyzer

```
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 if we omit ch.next?

Example of a part of a lexical analyzer

```
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 if we omit ch.next?

Lexer could generate a non-existing equality token!

White spaces and comments

Whitespace can be defined as a token, using space character, tabs, and various end of line characters. Similarly for comments.

In most languages (Java, ML, C) white spaces and comments can occur between any two other tokens have no meaning, so parser does not want to see them.

Convention: the lexical analyzer removes those "tokens" from its output. Instead, it always finds the next non-whitespace non-comment token.

Other conventions and interpretations of new line became popular to make code more concise (sensitivity to end of line or indentation). Not our problem in this course! Tools that do formatting of source also must remember comments. We ignore those.

Skipping simple comments

```
if (ch.current='/') {
  ch.next
  if (ch.current='/') {
     while (!isEOL && !isEOF) {
       ch.next
     }
  } else {
```

Skipping simple comments

```
if (ch.current='/') {
  ch.next
  if (ch.current='/') {
    while (!isEOL && !isEOF) {
      ch.next
    }
  } else {
    ch.current = DIV
  }
}
```

Skipping simple comments

```
if (ch.current='/') {
  ch.next
  if (ch.current='/') {
     while (!isEOL && !isEOF) {
       ch.next
     }
  } else {
    ch.current = DIV
  }
}
```

Nested comments: this is a single comment:

/* foo /* bar */ baz */

Solution:

Skipping simple comments

```
if (ch.current='/') {
  ch.next
  if (ch.current='/') {
    while (!isEOL && !isEOF) {
      ch.next
    }
  } else {
    ch.current = DIV
  }
}
```

Nested comments: this is a single comment:

/* foo /* bar */ baz */

Solution: use a counter for nesting depth

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Consider language with the following tokens:

ID: letter(digit | letter)*
LE: $<=$
LT: $<$
EQ: $=$

How can we split this input into subsequences, each of which in a token:

interpreters $<=$ *compilers*

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Consider language with the following tokens:

ID:	letter(digit | letter)*
LE:	$<=$
LT:	$<$
EQ:	$=$

How can we split this input into subsequences, each of which in a token:

interpreters $<=$ *compilers*

Some solutions:

ID(interpreters) LE ID(compilers) - OK, longest match rule

ID(inter) ID(preters) LE ID(compilers)

ID(interpreters) LT EQ ID(compilers)

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Consider language with the following tokens:

ID:	letter(digit | letter)*
LE:	$<=$
LT:	$<$
EQ:	$=$

How can we split this input into subsequences, each of which in a token:

interpreters $<=$ *compilers*

Some solutions:

ID(interpreters) LE ID(compilers) - OK, longest match rule
ID(inter) ID(preters) LE ID(compilers)
- not longest match: ID(inter)
ID(interpreters) LT EQ ID(compilers)

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Consider language with the following tokens:

ID:	letter(digit | letter)*
LE:	$<=$
LT:	$<$
EQ:	$=$

How can we split this input into subsequences, each of which in a token:

interpreters $<=$ *compilers*

Some solutions:

ID(interpreters) LE ID(compilers) - OK, longest match rule

ID(inter) ID(preters) LE ID(compilers)

- not longest match: ID(inter)

ID(interpreters) LT EQ ID(compilers)

- not longest match: LT

Longest match rule is greedy, but that's OK

Consider language with ONLY these three operators:

LT:	<
LE:	<=
IMP:	=>

For sequence:

$$<=>$$

lexer will first return LE as token, then report unknown token >.
This is the behavior that we expect.

This is despite the fact that one could in principle split the input into < and =>, which correspond to sequence LT IMP. But a split into < and => would not satisfy longest match rule, so we do *not* want it. Reporting error is the right thing to do here.

This behavior is not a restriction in practice: programmers we can insert extra spaces to stop maximal munch from taking too many characters.

Token priority

What if our token classes intersect?

Longest match rule does not help, because the same string belongs to two regular expressions

Examples:

- a keyword is also an identifier
- a constant that can be integer or floating point

Solution is priority: order all tokens and in case of overlap take one earlier in the list (higher priority).

Examples:

- if it matches regular expression for both a keyword and an identifier, then we define that it is a keyword.
- if it matches both integer constant and floating point constant regular expression, then we define it to be (for example) integer constant.

Token priorities for overlapping tokens must be specified in language definition.

Automating Construction of Lexers
by converting
Regular Expressions to Automata

Regular Expression to Programs

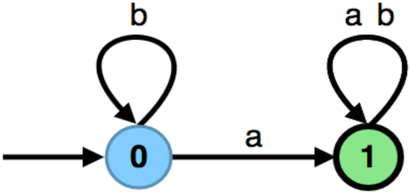
- How can we write a lexer that has these two classes of tokens:
  - a*b
  - aaa
- Consider run of lexer on: **aaaab** and on: **aaaaaa**

Regular Expression to Programs

- How can we write a lexer that has these two classes of tokens:
  - a*b
  - aaa
- Consider run of lexer on: **aaaab** and on: **aaaaaa**
- A general approach:

Finite Automaton (Finite State Machine)

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



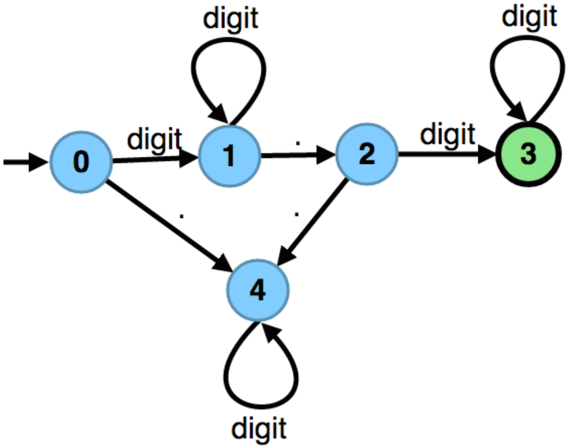
$$\delta \subseteq Q \times \Sigma \times Q,$$
$$q_0 \in Q,$$
$$F \subseteq Q$$

$$q_0 \in Q$$
$$q_1 \subseteq Q$$
$$\delta = \{ (q_0, a, q_1), (q_0, b, q_0),$$
$$(q_1, a, q_1), (q_1, b, q_1), \}$$

- 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*

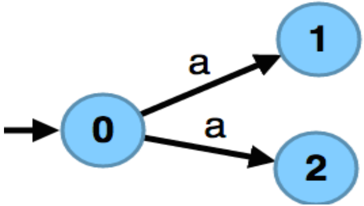


What if the decimal part is optional?

Kinds of Finite State Automata

- DFA: δ is a function : $(Q, \Sigma) \mapsto Q$
- NFA: δ could be a relation

- In NFA there is no unique next state. We have a set of possible next states.



Remark: Relations and Functions

- Relation $r \subseteq B \times C$

  $r = \{ \ldots, (b,c1) , (b,c2) ,\ldots \}$

- Corresponding function: $f : B \to 2^C$

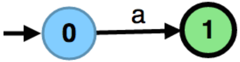
  $f = \{ \ldots (b,\{c1,c2\}) \ldots \}$

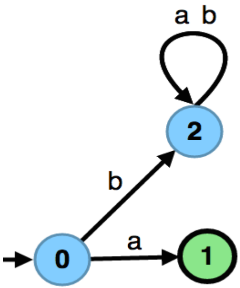
$f(b) = \{ c \mid (b,c) \in r \}$

- Given a state, next-state function returns a **set** of new states

  for *deterministic* automaton, set has *exactly 1* element

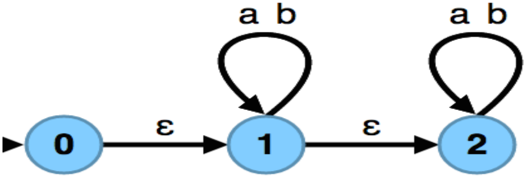
Allowing Undefined Transitions





- Undefined transitions are equivalent to transition into a sink state (from which one cannot recover)

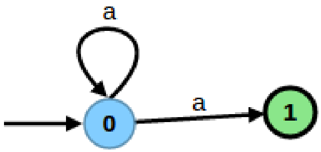
Allowing Epsilon Transitions



- Epsilon transitions:
  - traversing them does not consume anything
- Transitions labeled by a word:
  - traversing them consumes the entire word

When Automaton Accepts a Word

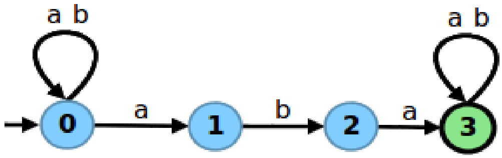
Automaton accepts a word *w* iff there **exists a path** in the automaton from the starting state to some accepting state such that concatenation of words on the path gives *w*.



- Does the automaton accept the word ***a*** ?

Exercise

- Construct a NFA that recognizes all strings over {a,b} that contain "aba" as a substring



Running NFA (without epsilons)

```
def δ(a : Char)(q : State) : Set[States] = { ... }
def δ'(a : Char, S : Set[States]) : Set[States] = {
  for (q1 <- S, q2 <- δ(a)(q1)) yield q2  // S.flatMap(δ(a))
}
def accepts(input : MyStream[Char]) : Boolean = {
 var S : Set[State] = Set(q0) // current set of states
 while (!input.EOF) {
  val a = input.current
  S = δ'(a,S)          // next set of states
 }
 !(S.intersect(finalStates).isEmpty)
}
```

NFA Vs DFA

- Every DFA is also a NFA (they are a special case)
- For every NFA there exists an equivalent DFA that accepts the same set of strings
- But, NFAs could be exponentially smaller (succinct)
- There are NFAs such that **every** DFA equivalent to it has exponentially more number of states

Regular Expressions and Automata

Theorem:

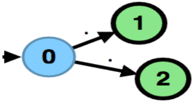
Let L be a language. There exists a regular expression that describes it if and only if there exists a finite automaton that accepts it.

Algorithms:

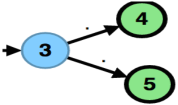
- regular expression → automaton (important!)
- automaton → regular expression (cool)

Recursive Constructions

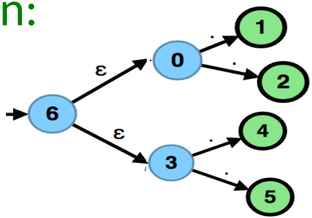
A_1:



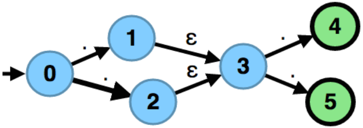
A_2:



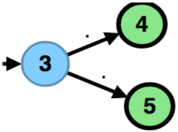
Union:



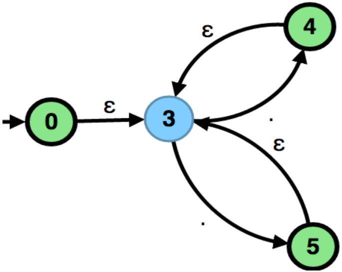
Concatenation:



Recursive Constructions

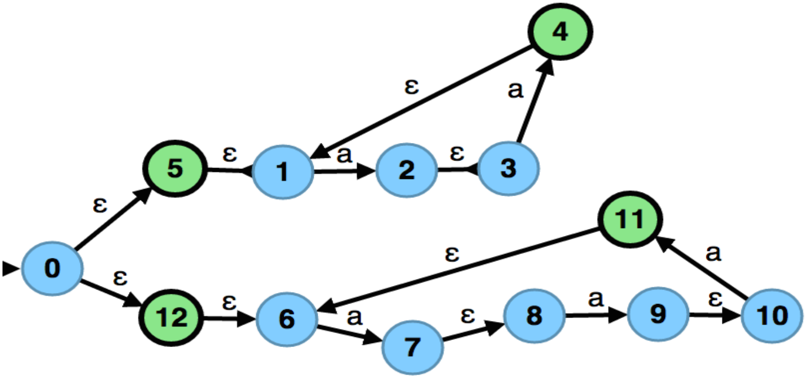


Star:



Exercise: (aa)* | (aaa)*

- Construct an NFA for the regular expression

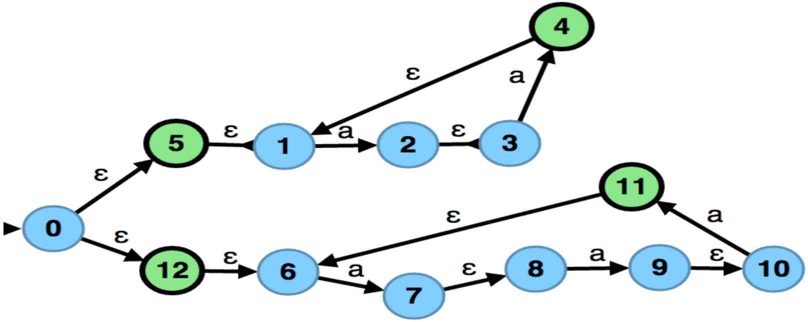


NFAs to DFAs (Determinization)

- keep track of a set of all possible states in which the automaton could be

- view this finite set as one state of new automaton

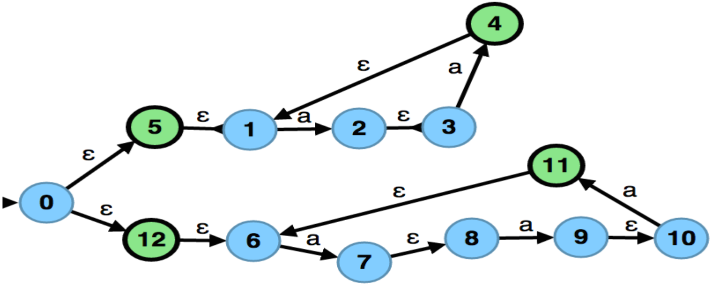
NFA to DFA Conversion



Possible states of the DFA: 2^Q

{ { } , { 0},…{12}, {0,1}, …,{0,12}, …{12, 12},
{0,1,2} …, { 0,1,2…,12 } }

NFA to DFA Conversion



Epsilon Closure

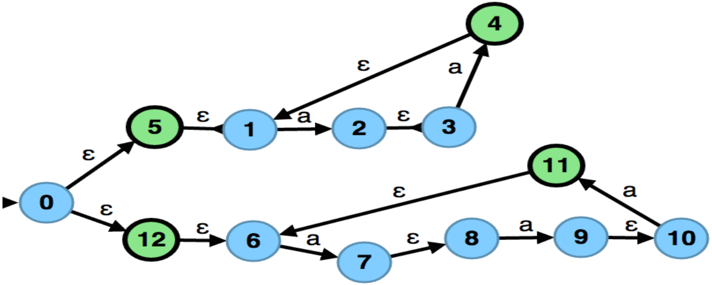
- All states reachable from a state through epsilon
- q $\in E(q)$
- If $q_1 \in E(q)$ and $\delta(q_1, \epsilon, q_2)$ then $q_2 \in E(q)$

E(0) = { } E(1) = { } E(2) = { }

NFA to DFA Conversion

- DFA: $(\Sigma, 2^Q, q_0', \delta', F')$
- $q_0' = E(q_0)$
- $\delta'(q', a) = \bigcup_{\{\exists q_1 \in q', \delta(q_1, a, q_2)\}} E(q_2)$
- $F' = \{\, q' | q' \in 2^Q, q' \cap F \neq \emptyset\}$

NFA to DFA Conversion through Examle



Clarifications

- what happens if a transition on an alphabet ‘a’ is not defined for a state ‘q’ ?
- $\delta'(\{q\}, a) = \emptyset$
- $\delta'(\emptyset, a) = \emptyset$

- Empty set represents a state in the NFA
- It is a trap/sink state: a state that has self-loops for all symbols, and is non-accepting.

Minimizing DFAs: Procedure

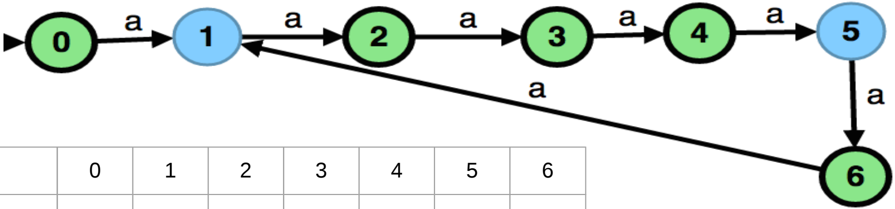
- Write down all pairs of state as a table
- Every cell in the table denotes whether the corresponding states are equivalent

	q1	q2	q3	q4	q5
q1	x	?	?	?	?
q2		x	?	?	?
q3			x	?	?
q4				x	?
q5					x

Minimizing DFAs: Procedure

- Inititalize cells (q1, q2) to false if one of them is final and other is non-final
- Make the cell (q1, q2) false, if q1 → q1’ on some alphabet symbol and q2 → q2’ on ‘a’ and q1’ and q2’ are not equivalent
- Iterate the above process until all non-equivalent states are found

Minimizing DFAs: Illustration



	0	1	2	3	4	5	6
0	x						
1		x					
2			x				
3				x			
4					x		
5						x	
6							x

Properties of Automata

Complement:

- Given a DFA A, switch accepting and non-accepting states in A gives the complement automaton A^c
- $\mathrm{L}(\mathrm{A}^c) = (\Sigma^* \setminus L(A))$

Note this does not work for NFA

Intersection: $\mathrm{L}(\mathrm{A}') = L(A_1) \cap L(A_2)$

$-A' = (\Sigma, Q_1 \times Q_2, (q_0^1, q_0^2), \delta', F_1 \times F_2)$

$-\delta'((q_1, q_2), a) = \delta(q_1, a) \times \delta(q_2, a)$

Emptiness of language, inclusion of one language into another, equivalence – they are all decidable

Exercise 0.1: on Equivalence

Prove that (a*b*)* is equivalent to (a|b)*

Sequential Hardware Circuits are Automata

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

Q – states of flip-flops, registers, etc.

Each state q_i is given by values $v : Vars \rightarrow \{0,1\}$

δ – combinational circuit that determines next state: given v compute v' according to a given logical circuit

Circuit can be exponentially smaller than graph