Language

Definition

A *language* over alphabet A is a set $L \subseteq A^*$. Example for $A = \{0,1\}$:

- 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 ← 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 $L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$

Example: $\{\varepsilon, a, aa\} \cdot \{b, bb\} = \{b, bb, ab, abb, aab, aabb\}$

The definition above states that $w \in L_1 L_2$ if and only if there is one or more ways to split w into 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

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

Is it the case that always $L_1L_2 = L_2L_1$? Prove or give counterexample.

Concatenation of Languages

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

Is this a monoid? (Is there a neutral element? Which law needs to hold? Does it hold?)

Concatenation of Languages

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

Is this a monoid? (Is there a neutral element? Which law needs to hold? Does it hold?)

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 $f_L : A^* \to \{0,1\}$ defined by $f_L(w) = 1$ for $w \in L$ and $f_L(w) = 0$ for $w \notin L$.
Here we use the `contains` field as the characteristic function and build the language $L_2 = \{(01)^n \mid n \geq 1\}$.

```
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 = Lang(f)
val test = L2.contains(0::1::0::1::Nil()) // true
```

Representing Language Concatenation

W can use code to express concatenation of computable languages.

```
def concat(L1: Lang[A], L2: Lang[A]): Lang[A]= {
  def f(w: List[A]) = {
    val n = w.length
    def checkFrom(i: BigInt) = {
      require(0 <= i && i <= n)
      (L1.contains(w.slice(0, i)) && L2.contains(w.slice(i, n))) ||
      (i < n && checkFrom(i + 1))
    }
    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)

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)

- words whose all contiguous blocks of b-s have even length

Can L^* be finite for some L? If so, describe all such L

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)

- words whose all contiguous blocks of b-s have even length

Can L^* be finite for some L? If so, describe all such L

- $\{\varepsilon\}^* = \{\varepsilon\}$, $\emptyset^* = \{\varepsilon\}$, for all others L has a word of length ≥ 1, so L^* is infinite

Star and the Empty Word

Concatenating with an empty word has no effect, so we have the following:

$$L^* = (L \setminus \{\varepsilon\})^* = \{\varepsilon\} \cup \bigcup_{n \geq 1} (L \setminus \{\varepsilon\})^n$$

Moreover, $w \in L^*$ 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 $1 \leq i \leq n$.

Star and the Empty Word

Concatenating with an empty word has no effect, so we have the following:

$$L^* = (L \setminus \{\varepsilon\})^* = \{\varepsilon\} \cup \bigcup_{n \geq 1} (L \setminus \{\varepsilon\})^n$$

Moreover, $w \in L^*$ 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 $1 \leq i \leq n$.

- we omit ε because it leaves concatenation the same
- we can assume $n \leq |w|$ because all blocks have length at least one

Star and the Empty Word

Concatenating with an empty word has no effect, so we have the following:

$$L^* = (L \setminus \{\varepsilon\})^* = \{\varepsilon\} \cup \bigcup_{n \geq 1} (L \setminus \{\varepsilon\})^n$$

Moreover, $w \in L^*$ 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 $1 \leq i \leq n$.

- we omit ε because it leaves concatenation the same
- we can assume $n \leq |w|$ because all blocks have length at least one

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

Star and the Empty Word

Concatenating with an empty word has no effect, so we have the following:

$$L^* = (L \setminus \{\varepsilon\})^* = \{\varepsilon\} \cup \bigcup_{n \geq 1} (L \setminus \{\varepsilon\})^n$$

Moreover, $w \in L^*$ 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 $1 \leq i \leq n$.

- we omit ε because it leaves concatenation the same
- we can assume $n \leq |w|$ because all blocks have length at least one

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

- try all possible ways of splitting w
- if $k = |w|$, for each point between the letters of w you can decide to split there or not, so there are 2^{k-1} ways to split: $w = \square \underbrace{|\square| \dots |\square|}_{k-1} \square$
- there is a better way - see exercises

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.

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.

Properties:

- does not being with b
- does not contain bb

in one phrase: there is an "a" before every "b"

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

Is bb inside L* ?

Question: Is it the case that

L*={ w | 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

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

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 ∅ (empty set of words)
- {ε}, denoted just ε (set containing the empty word)
- {*a*} for *a* ∈ 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

  letter (letter ∪ 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*

keywords
    if else while println

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

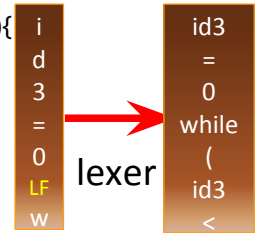
letter ::= a | b | c | … | z | A | B | C | … | Z

digit ::= 0 | 1 | … | 8 | 9

Manually Constructing Lexers by example

Stream of **Char-**s:

```
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
}
```



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 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.