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 $\leftarrow$ these are our focus

$L_2 = \{01, 0101, 010101, \ldots\}$ = those non-empty words that are of the form $01\ldots01$ 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
- $\{\varepsilon\}^n$
- $\{w_1\}\{w_2\}$

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
- $\{\varepsilon\}^n$
- $\{w_1\}\{w_2\}$

Note the difference in results between concatenation with:

- 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
- $\{\varepsilon\}^n$
- $\{w_1\}\{w_2\}$

Note the difference in results between concatenation with:

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

Is it the case that always $L_1 L_2 = L_2 L_1$? Prove or give counterexample.

Concatenation of Languages

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

Is this a monoid?

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?

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 a set, a language $L \subseteq A^*$ is given by its *characteristic function* $f_L : A^* \to \{0,1\}$ defined by: $f_L(w) \;=\; (\textbf{if } w \in L \textbf{ then } 1 \textbf{ else } 0)$.
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

We 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:

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

Moreover, $w \in L^*$ if and only if either $w = \varepsilon$ or, for some n where $1 \leq n \leq |w|$ (note $\leq$),

$$w = w_1 \ldots w_n$$

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

Star and the Empty Word

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 $1 \leq n \leq |w|$ (note $\leq$),

$$w = w_1 \dots w_n$$

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 $1 \leq n \leq |w|$ (note $\leq$),

$$w = w_1 \dots w_n$$

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 $1 \leq n \leq |w|$ (note $\leq$),

$$w = w_1 \dots w_n$$

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: $w = \square \underbrace{|\square| \dots |\square|}_{k-1} \square$

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 $1 \leq n \leq |w|$ (note $\leq$),

$$w = w_1 \dots w_n$$

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: $w = \square \underbrace{|\square| \dots |\square|}_{k-1} \square$
- Exercise: find a way to check $w \in L^*$ with polynomially many invocations of $w \in L$

Starring: $\{a, ab\}$

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

$$L^* = \{w \in A^* \mid P(w)\}$$

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 $P(w)$ that describes the language L^*, such that:

$$L^* = \{w \in A^* \mid P(w)\}$$

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

Example properties:

- does not begin with b
- does not contain bb

Conjectured property $P(w)$: there is an "a" immediately before every "b" inside w.

Proving the Property

$$L^* = \{w \in A^* \mid P(w)\}$$

where $P(w)$ is: there is an "a" immediately before every "b" occurrence inside w.
How to prove that this $P(w)$ is correct? Show two directions of set equality:

- $\{a, ab\}^* \subseteq \{w \mid P(w)\}$, that is: if w is a concatenation $w_1 ... w_n$ where each w_i is either a or ab, then, inside w, there is an "a" immediately before every "b".
- $\{w \mid P(w)\} \subseteq \{a, ab\}^*$, that is: if we have a string such that every occurrence of b has an a immediately left to it, then we can split w into some number of blocks $w_1 \dots w_n$ such that each w_i is either a or ab.

Regular Expressions

Regular Expressions

Mathematical expressions used to denote finite and infinite languages. Definition: a regular expression over language A is build inductively as follows:

- $\emptyset$, denoting the empty set of strings
- ε, denoting the language $\{\varepsilon\}$ containing only empty word
- a for $a \in A$, denoting the language with one word of length one, $\{a\}$
- $r_1 \mid r_2$ denoting the union of languages
- $r_1 r_2$ denoting concatenation of languages of r_1 and r_2
- $r*$ denoting the Kleene star of the language of r (a high priority operator)

Examples:

- $(a|ab)^*$ denoting the language $\{a, ab\}^*$
- $(a|b|c)\ (a|b|c|0|1)*$ denotes $\{a, b, c\}\{a, b, c, 0, 1\}^*$, the identifiers that start with one of the three letters a, b, c followed by a sequence of the letters or digits $0, 1$.

Example Use of Regular Expressions: grep

`grep` is a widely used command-line (terminal) tool that filters those lines that match a given pattern. Pattern can be a fixed string,

```
$ cd /etc/dictionaries-common
$ tail -n 5 words
zwieback
zwieback's
zygote
zygote's
zygotes
$ grep 'ncompat' words
incompatibilities
incompatibility
incompatibility's
incompatible
incompatible's
incompatibles
incompatibly
```

grep for clp using a regular expression

Find words that start with *c*, contain *l* and end with *p*:

```
$ grep '^c.*l.*p$' words
cantaloup
clamp
clap
claptrap
clasp
cleanup
clip
clomp
clop
clump
cowslip
```

Some notation specific to `grep`:

- `.` means any character, so `.*` means any string
- `^` means start of the line (otherwise it adds `.*` in front)
- `$` means end of the line (otherwise it adds `.*` at the end)

Another grep Example

Use '-E' so you don't have to escape union | and parentheses (,)

```
$ grep -E '^(b|c)(a|i|o)*t$' words
bait
bat
bit
boat
boot
bot
cat
coat
coot
cot
ct
```

One can also use regular expressions for syntax highlighting

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)

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

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

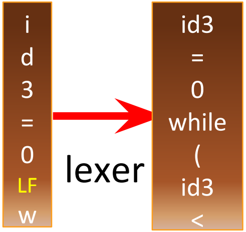
letter ::= a | b | c | … | z | A | B | C | … | Z

digit ::= 0 | 1 | … | 8 | 9

Manually Constructing Lexers by example

Stream of **Char**-s:

```
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

$\text{first}(\{a, bb, ab\}) = \{a,b\}$

$\text{first}(\{a, ab\}) = \{a\}$

$\text{first}(\{aaaaaaa\}) = \{a\}$

$\text{first}(\{a\}) = \{a\}$

$\text{first}(\{\}) = \{\}$

$\text{first}(\{\varepsilon\}) = \{\}$

$\text{first}(\{\varepsilon,ba\}) = \{b\}$

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:

$$\begin{aligned}
nullable(\emptyset) &= false \\
nullable(\varepsilon) &= true \\
nullable(a) &= false \\
nullable(e_1|e_2) &= nullable(e_1) \vee nullable(e_2) \\
nullable(e^*) &= true \\
nullable(e_1 e_2) &= nullable(e_1) \wedge nullable(e_2)
\end{aligned}$$

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.