REGULAR EXPRESSIONS

CMPT 379 Lecture 2b

Formal Language Theory

An **alphabet** is a set.

Each member of an alphabet is called a **character** or a **symbol**.

We typically use **Σ** to denote an alphabet.

Some examples:

$\Sigma = \{0, 1\}$ $\Sigma = \{\text{true}, \text{false}\}$ $\Sigma = \{A, C, G, T\}$

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$ $\Sigma = \{*, \#, \$, !, \S\}$

An Important Alphabet

ASCII (American Standard Code for Information Interchange) symbols are an important alphabet.

ASCII contains most symbols we use in computer languages. It has 128 symbols, including:

- The letters of the English alphabet in both lower and upper case **{a, b, …, z, A, B, …, Z}**
- The digits **{0, 1, …, 9}**
- Various punctuation, mathematical symbols, and control codes **{. ; , ! ? # = ( ) & ^ @ ~ … }**

Modern Computer Alphabets

ASCII is an old alphabet. It is actually a code specifying which numbers (from 0 to 127) correspond to which symbols.

Modern versions of the same idea include ASCII as a subset. The modern versions are called **Unicode** or **UTF** (Unicode Transformation Format). There are various versions of UTF, known as **UTF-7**, **UTF-8**, **UTF-16**, and **UTF-32**. The number after the UTF indicates how many bits correspond to a normal character. These alphabets contain characters from many different languages.

Strings

A **string** is a sequence of 0 or more symbols from some alphabet, chosen with repetitions.

The symbol **ε** denotes the string with 0 characters. It is a string over **every** alphabet.

If {0, 1} is our alphabet, then **010110** is a string. A few other strings on this alphabet are **1101**, **001**, **0**, **1**, **ε**, and **10011011101000101010010**.

If {A, C, G, T} is our alphabet, then **CATTAGGA**, **TCAAAGATC**, and **GTT** are some strings.

Formal Languages

A (formal) **language** is a set of strings over some alphabet **Σ**.

The set **{ε, a, b, aa, ab, ba, bb}** is a language over the alphabet {a, b}. This is the language consisting of strings of at most two characters.

The set **{ε, a, aa, aaa, aaaa, …}** is also a language over the same alphabet. This is the language of all strings that consist only of a's. (You don't have to use all characters in a language)

Language Formation

Given a language or two, we can form other languages from them.

We let $\mathbf{L_a}$ be the language $\mathbf{\{a\}}$, for any $\mathbf{a}$ in alphabet $\mathbf{\Sigma}$.

We let $\mathbf{L_\varepsilon}$ be the language $\mathbf{\{\varepsilon\}}$.

We let $\mathbf{L_\varnothing}$ be the language $\mathbf{\{\ \}}$.

We let $\mathbf{L_\Sigma}$ be the language $\mathbf{\{c \mid c \in \Sigma\}}$.

If $\mathbf{L_1}$ and $\mathbf{L_2}$ are languages, we let $\mathbf{L_1 \cup L_2}$ be the usual set-theoretic union of $\mathbf{L_1}$ and $\mathbf{L_2}$. For example, $\mathbf{L_a \cup L_d = \{a, d\}}$.

Concatenation of Languages

If L_1 and L_2 are languages, we let L_1L_2 be the set $\{\alpha\beta \mid \alpha \in L_1 \text{ and } \beta \in L_2\}$. Here, α and β are strings and $\alpha\beta$ denotes the string α followed by the string β.

If $L_1 = \{AC, TG\}$ and $L_2 = \{A, GCCA\}$, then

$$L_1L_2 = \{ACA, ACGCCA, TGA, TGGCCA\}$$

$L_\varepsilon L = LL_\varepsilon = L$ for any language L.

For any languages L_1, L_2, and L_3, $(L_1L_2)L_3 = L_1(L_2L_3)$.

In general $L_1L_2 \neq L_2L_1$.

Exponentiation of Languages

If **L** is a language, then $\mathbf{L^k}$ is that language concatenated **k** times with itself.

For instance, $\mathbf{L^2 = LL}$.

As special cases, $\mathbf{L^0 = L_\varepsilon}$ and $\mathbf{L^1 = L}$.

$\mathbf{L_b{}^0 = \{\varepsilon\}}$

$\mathbf{L_b{}^1 = \{b\}}$

$\mathbf{L_b{}^2 = \{bb\}}$

$\mathbf{L_b{}^3 = \{bbb\}}$

...

If $\mathbf{M = \{0, 111\}}$ then $\mathbf{M^0 = \{\varepsilon\}}$

$\mathbf{M^1 = \{0, 111\}}$

$\mathbf{M^2 = \{00, 0111, 1110, 111111\}}$

$\mathbf{M^3 = \{000, 00111, 01110, 0111111, 11100, 1110111, 1111110, 111111111\}}$

Kleene Closure

If **L** is a language, then the Kleene Closure **L*** is

$L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$

For instance, $L_d^* = \{\varepsilon, d, dd, ddd, dddd, \ldots\}$. is the language of all strings you can make with d.

$(L_a \cup L_b)^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$ is the language of all strings of a's and b's.

Regular Languages

Any language that can be constructed from the basic set definitions **L_a, L_ε, $L_\varnothing$,** and **L_Σ** and the general operations union, concatenation, and Kleene closure is called a **regular language**.

Regular languages exhibit a sort of **regularity**.

There is a more compact notation for (most) regular languages that is called **regular expressions**.

Regular Expressions

For any character **a** $\in\Sigma$, the regular expression **a** denotes the language L_a.

The regular expression ε denotes the language L_ε.

If α and β are regular expressions for languages L_α and L_β, the regular expression $\alpha + \beta$ denotes the language $L_\alpha \cup L_\beta$. $\alpha + \beta$ can also be written as $\alpha | \beta$.

If α and β are regular expressions for languages L_α and L_β, the regular expression $\alpha\beta$ denotes the language $L_\alpha L_\beta$.

If α is a regular expression for language L_α the regular expression α^* denotes the language $L_\alpha{}^*$.

Parentheses are used for clarification . Kleene closure takes precedence over concatenation which takes precedence over union. a + bc* = a + (b(c*)).

Regular Expression Examples

The regular expression **a(b + ε)a** denotes the language **{aba, aa}**.

The regular expression **a(b + c)ab** denotes the language **{abab, acab}**.

The regular expression **ab + cab** denotes the language **{ab, cab}**.

The regular expression **ab*** denotes the language **{a, ab, abb, abbb, abbbb, …}**.

The regular expression **a(b+c)(d+e)** denotes the language **{abd, abe, acd, ace}**.

Regular Expressions as Patterns

Regular expressions are often used as **patterns** to match. A string is said to **match** the pattern of the regular expression if the string is in the language represented by the regular expression.

The string **abab** matches the regular expression **a(b + c)ab**.

The string **abbb** matches the regular expression **ab***.

The string **ade** does not match the regular expression **a(b+c)(d+e)**

Notational extensions

We can add some extra notation to regular expressions without changing what languages they can represent.

If α is a regular expression, $\alpha^?$ (sometimes written as $\alpha?$) is equivalent to $(\varepsilon + \alpha)$.

If α is a regular expression, α^+ is equivalent to $\alpha\alpha^*$.

If α is a regular expression for language L_α the regular expression α^k denotes the language $L_\alpha{}^k$. k must be a constant or used once. For instance, $a^k b^k$ is not a regular expression.

Notational extensions

The expression **[abgmr]** is called a **character class** and is equivalent to **(a+b+g+m+r)**

Abbreviated ellipses (..) in a character class means all characters in the range. The expression **[ab..gm]** is equivalent to **(a+b+c+…+ g+m)**. Common constructions are **[a..z]**, **[a..zA..Z]**, and **[0..9A..F]**.

A tilde **~** or caret **^** at the beginning of a character class denotes "all characters of Σ that are **not** in."
[^a..z] means "all characters that are not lower case letters"

Notational extensions

If α and β are regular expressions, $\alpha \bowtie \beta$ means **$(\varepsilon + \alpha(\beta\alpha)^*)$**. This is a β**-**separated list of 0 or more α's. For example, **$x \bowtie ,$** means **$\varepsilon + x + x,x + x,x,x + \ldots$**, that is, a comma-separated list of zero or more x's. (uncommon)

Uses of regular expressions

In an editor, you might look for a string that matches a regular expression you type in.

In a command shell, you might use a regular expression to match all of the file names you want to use.

Basically, you can use them in most situations in which you want to match a pattern.

Later in the course, we will get into the technology behind matching regular expressions on a computer.

Uses of regular expressions

In a compiler, we use regular expressions when matching input text to tokens, which are the product of lexical analysis.

There are tools (for instance, **lex**, **flex**, **ANTLR**) which will automatically generate a lexical analyzer based on a regular-expression specification of the tokens.

We will not be using these tools, but will have a similar mechanism for parts of our lexical analyzers. They are quite easy to write by hand anyhow.

Tokens in project Milestone 1

- *integerLiteral* → [**0..9**]$^+$
- *booleanLiteral* → **true** | **false**
-
- *floatingLiteral* → ([**0..9**]$^+$ **.** [**0..9**]$^+$) (**E** (**+** | **-**)$^?$ [**0..9**]$^+$)$^?$
  3.4 0.0 0.2E4 3.15E-21 423.17E+07
- *stringLiteral* → " [^"**\n**] * "
  "hello" "" "A#c"
-
- *characterLiteral* → #a | ##[0..9#] | #[0..7]$^+$
  #Z #! ### ##3 #027
-
- *identifier* → [**a..z_**]$^+$
- *identifier* → [**a..zA..Z_ @**][**a..zA..Z_@0..9**]*
  canLit AIndex@b49 _min @max@

Tokens in project Milestone 1

- *punctuator* → *unaryOperator* | *operator* | *punctuation*
- *punctuator* → *operator* | *punctuation*
- *operator* → *unaryOperator* | *arithmeticOperator* | *comparisonOperator*
-
- *unaryOperator* → -
- *unaryOperator* → **+** | -

- *operator* → **+** | ***** | **>**
- *arithmeticOperator* → **+** | **–** | ***** | **/**
- *comparisonOperator* → **<** | **<=** | **==** | **!=** | **>** | **>=**
-
- *punctuation* → ; | , | . | **{** | **}**| **:=**
- *punctuation* → ; | **{** | **}** | **(** | **)** | **[** | **]** | **%** | **:=**
-
- *comment* → **%** [^**%\n**]* (% | **n**)

  % wrap up the loop %
  % endline comment

Brief Overview – Lexical Analysis Package of Project