Theory of Computational

Arithmetic Expressions:

- Suppose we ask ourselves what constitutes a valid arithmetic expression or AE for short.
- The alphabet for this language is
- Σ = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /, (,)}

Arithmetic Expression AE:

- Obviously, the following expressions are not valid:

  (3 + 5) + 6) 2(/8 + 9) (3 + (4-)8)
- The first contains unbalanced parentheses; the second contains the forbidden substring /; the third contains the forbidden substring -).
- Are there more rules? The substrings // and */ are also forbidden.
- Are there still more?
- The most natural way of defining a valid AE is by using a **recursive definition**, rather than a long list of forbidden substrings.

Recursive Definition of AE:

- **Rule 1:** Any number (positive, negative, or zero) is in AE.
- **Rule 2:** If x is in AE, then so are

  (i) (x)

  (ii) -x (provided that x does not already start with a minus sign)
- **Rule 3:** If x and y are in AE, then so are

  (i) x + y (if the first symbol in y is not + or -)

  (ii) x - y (if the first symbol in y is not + or -)

  (iii) x * y

  (iv) x / y

  (v) x ** y (our notation for exponentiation)
- The above definition is the most natural, because it is the method we use to recognize valid arithmetic expressions in real life.
- For instance, we wish to determine if the following expression is valid:

  (2 + 4) * (7 * (9 - 3)/4)/4 * (2 + 8) - 1
- We do not really scan over the string, looking for forbidden substrings or count the parentheses.
- We actually imagine the expression in our mind broken down into components:

  Is (2 + 4) OK? Yes Is (9 - 3) OK? Yes Is 7 * (9 - 3)/4 OK? Yes, and so on.

- Note that the recursive definition of the set AE gives us the possibility of writing 8/4/2, which is ambiguous, because it could mean 8/(4/2) = 4 or (8/4)/2 = 1.
- However, the ambiguity of 8/4/2 is a problem of *meaning*. There is no doubt that this string is a word in AE, only doubt about what it means.
- By applying Rule 2, we could always put enough parentheses to avoid such confusion.
- The recursive definition of the set AE is useful for proving many theorems about arithmetic expressions, as we shall see in the next few slides.

Defining Languages by Another New Method:
Regular Expressions:

- ❖ **Defining Languages by Another New Method**
- ❖ **Formal Definition of Regular Expressions**
- ❖ **Languages Associated with Regular Expressions**
- ❖ **Finite Languages Are Regular**
- ❖ **How Hard It Is to Understand a Regular Expression**
- ❖ **Introducing EVEN-EVEN**

❖ Language-Defining Symbols:

- We now introduce the use of the Kleene star, applied not to a set, but directly to the letter x and written as a superscript: x*.
- This simple expression indicates some sequence of x's (may be none at all):

  $\mathbf{x}^* = \Lambda$ or x or x^2 or x^3…

  = x^n for some n = 0, 1, 2, 3, …
- Letter **x** is intentionally written in boldface type to distinguish it from an alphabet character.
- We can think of the star as an unknown power. That is, **x*** stands for a string of x's, but we do not specify how many, and it may be the null string .
- The notation x* can be used to define languages by writing, say L_4 = language (x*)
- Since x* is any string of x's, L_4 is then the language of all possible strings of x's of any length (including Λ).
- We should not confuse x* (which is a **language-defining symbol**) with L_4 (which is the **name** we have given to a certain language).
- Given the alphabet = {a, b}, suppose we wish to define the language L that contains all words of the form one *a* followed by some number of *b*'s (maybe no *b*'s at all); that is

  L = {a, ab, abb, abbb, abbbb, …}

- Using the language-defining symbol, we may write
  L = language (ab*)
- This equation obviously means that L is the language in which the words are the concatenation of an initial a with some or no b's.
- From now on, for convenience, we will simply say **some** b**'s** to mean **some or no** b**'s**. When we want to mean **some positive number of** b**'s**, we will explicitly say so.
- We can apply the Kleene star to the whole string ab if we want:
  (ab)* = Λ or ab or abab or ababab…
- Observe that
  $(ab)^* \neq a^*b^*$
- Because the language defined by the expression on the left contains the word abab, whereas the language defined by the expression on the right does not.
- If we want to define the language L1 = {x; xx; xxx; …} using the language-defining symbol, we can write
  L1 = language (xx*)
  which means that each word of L1 must start with an x followed by some (or no) x's.
- Note that we can also define L1 using the notation + (as an exponent) introduced in Chapter 2:
  L1 = language(x^+)
- which means that each word of L1 is a string of some positive number of x's.

Plus Sign:

- Let us introduce another use of the plus sign. By the expression
  x + y
  where x and y are strings of characters from an alphabet, we mean **either** x **or** y.
- Care should be taken so as not to confuse this notation with the notation + (as an exponent).

Example:

- Consider the language T over the alphabet
  Σ = {a; b; c}:
- T = {a; c; ab; cb; abb; cbb; abbb; cbbb; abbbb; cbbbb; …}
- In other words, all the words in T begin with either an a or a c and then are followed by some number of b's.
- Using the above plus sign notation, we may write this as
  T = language ((a+ c)b*)

Example:

- Consider a finite language L that contains all the strings of a's and b's of length three exactly:
  L = {aaa, aab, aba, abb, baa, bab, bba, bbb}
- Note that the first letter of each word in L is either an a or a b; so are the second letter and third letter of each word in L.
  - Thus, we may write: L = language((a+ b)(a + b)(a + b))
  - or for short, L = language($(a+b)^3$)

Example:

- In general, if we want to refer to the set of all possible strings of a's and b's of any length whatsoever, we could write language ((a+ b)*)
- This is the set of **all possible strings** of letters from the alphabet Σ = {a, b}, **including the null string**.
- This is powerful notation. For instance, we can describe all the words that begin with first an a, followed by anything (i.e., as many choices as we want of either a or b) as: a(a + b)*

❖ Formal Definition of Regular Expressions:

- The set of **regular expressions** is defined by the following rules:
- Rule 1: Every letter of the alphabet Σ can be made into a regular expression by writing it in **boldface, Λ** itself is a regular expression.
- Rule 2: If r_1 and r_2 are regular expressions, then so are:
  (i) (r_1)
  (ii) r_1r_2
  (iii) $r_1 + r_2$
  (iv) r_1^*
- Rule 3: Nothing else is a regular expression.

Note: If r_1 = aa + b then when we write r_1^* , we really mean $(r_1)^*$, that is $r_1^* = (r_1)^*$ = (aa + b)*

Example:

- Consider the language defined by the expression: (a + b)*a(a + b)*
- At the beginning of any word in this language we have
  *(a + b)**, which is any string of *a*'s and *b*'s, then comes an *a*, then another any string.
- For example, the word abbaab can be considered to come from this expression by 3 different choices:
  (Λ)a(bbaab) or (abb)a(ab) or (abba)a(b)

- This language is the set of all words over the alphabet $\Sigma = \{a, b\}$ that have at least one a.
- The only words left out are those that have only b's and the word Λ.
  These left out words are exactly the language defined by the expression b*.
- If we combine this language, we should provide a language of all strings over the alphabet $\Sigma = \{a, b\}$. That is, $(a + b)^* = (a + b)^*a(a + b)^* + b^*$

Example:

- The language of all words that have at least two a's can be defined by the expression: $(a + b)^*a(a + b)^*a(a + b)^*$
- Another expression that defines all the words with at least two a's is
  $b^*ab^*a(a + b)^*$
- Hence, we can write: $(a + b)^*a(a + b)^*a(a + b)^* = b^*ab^*a(a + b)^*$

where by the equal sign we mean that these two expressions are **equivalent** in the sense that they describe the same language.

Example:

- The language of all words that have at least one a and at least one b is somewhat trickier. If we write
  $(a + b)^*a(a + b)^*b(a + b)^*$

then we are requiring that an a must precede a b in the word. Such words as ba and bbaaaa are not included in this language.

- Since we know that either the a comes before the b or the b comes before the a, we can define the language by the expression
  $(a + b)a(a + b)b(a + b) + (a + b)b(a + b)a(a + b)$

Note that the only words that are omitted by the first term $(a + b)^*a(a + b)^*b(a + b)^*$ are the words of the form some b's followed by some a's. They are defined by the expression bb^*aa^*

Example:

- We can add these specific exceptions. So, the language of all words over the alphabet $\Sigma = \{a, b\}$ that contain at least one a and at least one b is defined by the expression: $(a + b)a(a + b)b(a + b) + bb^*aa^*$
- Thus, we have proved that

$$(a + b)^*a(a + b)^*b(a + b)^* + (a + b)^*b(a + b)^*a(a + b)^*$$
$$= (a + b)^*a(a + b)^*b(a + b)^* + bb^*aa^*$$

Example:

- In the above example, the language of all words that contain both an a and ab is defined by the expression (a + b)*a(a + b)*b(a + b)* + bb*aa*

- The only words that do not contain both an a and ab are the words of all a's, all b's, or Λ.
- When these are included, we get everything. Hence, the expression
  (a + b)*a(a + b)*b(a + b)* + bb*aa* + a* + b*

defines all possible strings of a's and b's, including (accounted for in both a and b).

- Thus: (a + b)* = (a + b)*a(a + b)*b(a + b)* + bb*aa* + a* + b*

Example:

- The following equivalences show that we should not treat expressions as algebraic polynomials:
  (a + b)* = (a + b)* + (a + b)*
  (a + b)* = (a + b)* + a*
  (a + b)* = (a + b)*(a + b)*
  (a + b)* = a(a + b)* + b(a + b)* + Λ
  (a + b)* = (a + b)*ab(a + b)* + b*a*
- The last equivalence may need some explanation:
  – The first term in the right hand side, (a + b)*ab(a + b)*, describes all the words that contain the substring ab.
  – The second term, b*a* describes all the words that do not contain the substring ab (i.e., all a's, all b's, Λ, or some b's followed by some a's).

Example:

- Let V be the language of all strings of a's and b's in which either the strings are all b's, or else an a followed by some b's. Let V also contain the word Λ. Hence, V = {Λ, a, b, ab, bb, abb, bbb, abbb, bbbb, …}
- We can define V by the expression: b* + ab* where Λ is included in b*.
- Alternatively, we could define V by (Λ + a)b*

which means that in front of the string of some b's, we have either an a or nothing.

- Hence, (Λ + a)b* = b* + ab*
- Since b* = Λb*, we have (Λ + a)b* = b* + ab*

which appears to be distributive law at work.

- However, we must be extremely careful in applying distributive law. Sometimes, it is difficult to determine if the law is applicable.

Product Set:

- If S and T are sets of strings of letters (whether they are finite or infinite sets), we define the **product set** of strings of letters to be

ST = {all combinations of a string from S concatenated with a string from T in that order}

Example:

- If S = {a, aa, aaa} and T = {bb, bbb} then
  ST = {abb, abbb, aabb, aabbb, aaabb, aaabbb}
- Note that the words are not listed in lexicographic order.
- Using regular expression, we can write this example as
  (a + aa + aaa)(bb + bbb) = abb + abbb + aabb + aabbb + aaabb + aaabbb

Example:

- If M = {Λ, x, xx} and N = {Λ, y, yy, yyy, yyyy, …} then
- MN ={Λ, y, yy, yyy, yyyy,…x, xy, xyy, xyyy, xyyyy, …xx, xxy, xxyy, xxyyy, xxyyyy, …}
- Using regular expression (Λ + x + xx)(y*) = y* + xy* + xxy*

❖ Languages Associated with Regular Expressions:

Definition:

- The following rules define the **language associated** with any regular expression:
- Rule 1: The language associated with the regular expression that is just a single letter is that one-letter word alone, and the language associated with Λ is just {Λ}, a one-word language.
- Rule 2: If r_1 is a regular expression associated with the language L_1 and r_2 is a regular expression associated with the language L_2, then:

(i) The regular expression $(r_1)(r_2)$ is associated with the product L_1L_2, that is the language L_1 times the language L_2: language$(r_1r_2) = L_1L_2$

(ii) The regular expression $r_1 + r_2$ is associated with the language formed by the union of L_1 and L_2: language$(r_1 + r_2) = L_1 + L_2$

(iii) The language associated with the regular expression $(r_1)^*$ is L_1^*, the Kleene closure of the set L_1 as a set of words: language$(r_1^*) = L_1^*$

❖ Finite Languages Are Regular:

Theorem 5:

- **If** L **is a finite language (a language with only finitely many words), then** L **can be defined by a regular expression. In other words, all finite languages are regular.**

Proof

- Let L be a finite language. To make one regular expression that defines L, we turn all the words in L into boldface type and insert plus signs between them.
- For example, the regular expression that defines the language
  L = {baa, abbba, bababa} is baa + abbba + bababa

- This algorithm only works for finite languages because an infinite language would become a regular expression that is infinitely long, which is forbidden.

❖ <u>How Hard It Is To Understand A Regular Expression</u>:

Let us examine some regular expressions and see if we could understand something about the languages they represent.

<u>Example:</u>

- Consider the expression
  (a + b)*(aa + bb)(a + b)* =(arbitrary)(double letter)(arbitrary)
- This is the set of strings of a's and b's that at some point contain a double letter.

Let us ask, "What strings do not contain a double letter?" Some examples are
Λ; a; b; ab; ba; aba; bab; abab; baba; …

- The expression (ab)* covers all of these except those that begin with b or end with a. Adding these choices gives us the expression: (Λ + b)(ab)*(Λ + a)
- Combining the two expressions gives us the one that defines the set of all strings
  (a + b)*(aa + bb)(a + b)* + (Λ + b)(ab)*(Λ + a)

<u>Examples:</u>

- Note that (a + b*)* = (a + b)*

since the internal * adds nothing to the language. However,

(aa + ab*)* ≠ (aa + ab)*

since the language on the left includes the word *abbabb*, whereas the language on the right does not. (The language on the right cannot contain any word with a double b.)

Example

- Consider the regular expression: (a*b*)*.
- The language defined by this expression is all strings that can be made up of factors of the form a*b*.
- Since both the single letter a and the single letter b are words of the form a*b*, this language contains all strings of a's and b's. That is,
  (a*b*) = (a + b)*
- This equation gives a big doubt on the possibility of finding a set of algebraic rules to reduce one regular expression to another equivalent one.

❖ <u>Introducing EVEN-EVEN:</u>

- Consider the regular expression E = [aa + bb + (ab + ba)(aa + bb)*(ab + ba)]*
- This expression represents all the words that are made up of *syllables* of three types: $type_1$ = aa
  $type_2$ = bb
  $type_3$ = (ab + ba)(aa + bb)*(ab + ba)

- Every word of the language defined by E contains an **even number of** a**'s and an even number of** b**'s**.
- All strings with an **even number of** a**'s and an even number of** b**'s** belong to the language defined by E.

Algorithms for EVEN-EVEN:

- We want to determine whether a long string of a's and b's has the property that the number of a's is even and the number of b's is even.

Algorithm 1: Keep two binary flags, the a-flag and the b-flag. Every time an a is read, the a-flag is reversed (0 to 1, or 1 to 0); and every time a b is read, the b-flag is reversed. We start both flags at 0 and check to be sure they are both 0 at the end.

Algorithm 2: Keep only one binary flag, called the type$_3$-flag. We read letter in two at a time. If they are the same, then we do not touch the type$_3$-flag, since we have a factor of type$_1$ or type$_2$. If, however, the two letters do not match, we reverse the type$_3$-flag. If the flag starts at 0 and if it is also 0 at the end, then the input string contains an even number of a's and an even number of b's.

- If the input string is

(aa)(ab)(bb)(ba)(ab)(bb)(bb)(bb)(ab)(ab)(bb)(ba)(aa) then, by Algorithm 2, the type$_3$-flag is reversed 6 times and ends at 0.

- We give this language the name EVEN-EV EN. so, EVEN-EV EN ={Λ, aa, bb, aaaa, aabb, abab, abba, baab, baba, bbaa, bbbb, aaaaaa, aaaabb, aaabab, …}