Formal languages

Let \Sigma be a set of characters (an *alphabet*).

A *language* over \Sigma is a set of strings of characters drawn from \Sigma

e.g., English language: Alphabet = characters, Language = Sentences.

Another example: Alphabet = ASCII character set, Language = C programs

Meaning function L maps syntax to semantics. e.g., L(e) = M

Example: e could be a regular expression, and M would be the set of strings that it represents

Meaning of regular expressions:

```
L(\epsilon) = { "" }
L('c') = { "c" }
L(A + B) = L(A) U L(B)
L(AB) = {ab | a \in L(A) and b \in L(B)}
L(A*) = U(i>= 0)L(A^i)
```

Significance of the meaning function: separates syntax from semantics (we can change syntax without changing semantics; different syntax good for different languages/environments). Also, allows for redundancy: multiple syntaxes for the same semantics, e.g., some are concise while others are more efficient (optimization!)

Roman numerals vs. decimal number system: the meanings of both number systems are same, but the notation is extremely different

Example of redundancy: L(0*) = L(0 + 0*) = L(\epsilon + 00*) = ...

Thus L is a many-to-one function, and this is the basis of optimization. L is never one-to-many

Lexical Specifications

```
Keyword: "if" or "else" or "then" or ...
       : 'i''f' + 'e''l''s''e' + 't''h''e''n' + ...
       : 'if' + 'else' + 'then' + ... (shorthand)
```

Integers = non-empty string of digits

```
digit = '0' + '1' + '2' + ...
Integer = digit.digit*    (ensures that there is at least one digit, also represented as digit+)
```

Identifier = strings of letters or digits, starting with a letter

```
letter = 'a' + 'b' + ... + 'z' + 'A' + ... + 'Z' (shorthand: [a-zA-Z])
Identifier = letter(letter + digit)*
```

Whitespace = a non-empty sequence of blanks, newlines, and tabs

```
Whitespace = (' ' + '\n' + '\t')+
```

Example of a real language: PASCAL

```
digit = '0' + '1' + '2' + ...
digits = digit+
opt_fraction = ('.' digits) + \epsilon
opt_exponent = ('E'('+' + '-' + \epsilon) digits) + \epsilon
num = digits opt_fraction opt_exponent
```

Given a string *s* and a regular expression *R*, does *s \in L(R)*?

Some additional notation (for conciseness)

```
A+ = AA*
A | B = A + B
A? = A + \epsilon
[a-z] = 'a' + .. + 'z'
[^a-z] = complement of [a-z]
```

Lexical analysis

1. First step: write a regular expression for each token class (e.g., Keyword, Identifier, Number, ...)
2. Construct a giant regular expression that is formed by

   ```
   R = Keyword + Identifier + Number + ...
   or R = R1 + R2 + R3 + R4 + ....
   ```

3. For input `x1...xn`

   For 1 <= i <= n, check
   x1...xi \in L(R)

4. If success, we know that x1...xi \in L(Rj) for some j
5. Remove x1...xi from the input and go to step 3

This algorithm has some ambiguities.

1. How much input is used?

   ```
   x1...xi \in L(R)
   x1...xj \in L(R)
   i != j
   ```

   e.g., '=' vs. '=='. The answer is that we should always take the longer one. This is called "Maximal Munch". This is how we process the English language as well, for example (this is how humans work). E.g., if we see the word 'inform', we do not read it as 'in' + 'form' but as a single word 'inform'.
2. Which token is used?

   ```
   x1...xi \in L(Rj)
   x1...xi \in L(Rk)
   ```

   e.g., Keywords = 'if' + 'else' + ...
   Identifiers = letter(letter + digit)*

   Should we treat 'if' as a keyword or an identifier?

   Answer: use a priority order. e.g., Keywords higher priority than Identifiers

3. What if no rule matches?

   ```
   x1...xi \notin L(R)
   ```

   This will cause the program to terminate, as the program does not know what to do.

   Answer: do not let this happen. Create another token class called *Error* and put it last in the priority order.

   ```
   Error = all strings not in the lexical specification
   ```

Summary: a simple lexical analysis can be achieved using regular expressions + a few rules to resolve ambiguities and errors.

Good algorithms are known to do such lexical analysis (subject of future lectures)

- Require only single pass over the input
- Few operations per character (table lookup)

DFAs

- Only one possible transition for any input from any state
- No epsilon moves
- Only one path possible for any input

NFAs

- Potentially multiple possible transition for any input from any state
- Allow epsilon moves
- Accept if any path accepts

Conversion from NFA to DFA

- \epsilon-closure of an NFA state = all NFA states that are reachable through \epsilon moves from that state
- Have one state for each subset of NFA states: NFA can be exponentially smaller than the DFA!
- Execution of an NFA will always yield a subset of NFA states at the current execution point. Use the DFA state corresponding to this subset in the DFA path.
- While this conversion is exponential:
  - Typical number of states in the NFA are quite small
  - Exponential blow-up is worst-case. Typical blow-ups are much smaller.

Conversion from Regular language to NFA

- \epsilon : start state is an accepting state
- 'c' : consume one character to reach an accepting state from the start state
- A + B : add a new start state, and add \epsilon moves from the start state to the start states of A and B resp. Similarly, add epsilon moves from the accepting states of A and B to the accepting state of the new automaton
- AB : connect the accepting state of A to the start state of B through an \epsilon move.
- A* : add a start state, a loop state. Add \epsilon moves from start state to loop state, and from loop state to final accepting state. add \epsilon moves from loop state to A's start state and from A's accepting state to the loop state.
-

Implementing an automaton

- A DFA can be implemented as a 2D table T
  - One dimension : state
  - Second dimension : Input character
  - For every transition Si --a--> Sj, define T[i, a] = j

-
```
i = 0
state = 0
      while (input[i]) {
  state = T[state, input[i++]];
}
```

- Single pass, two lookups per input character (one for the input, another for the table)
- More space-efficient implementation of the table T are possible: e.g., share identical rows. Identical rows are very common, and so this results in significant compaction in the table's storage size. *Con*:

Extra pointer de-reference during table lookup. Space vs. time tradeoff

- Even more space-efficient implementation: Use a table for the NFA (not the DFA): In this case, an element of the table would be a set of states. The table will be relatively small, because the number of states is limited by the number of NFA states and the size of the input alphabet. However, simulating the NFA is now more expensive, because we have to deal with sets of states (instead of single states). For each transition, we have to look at all the current set of possible states, to generate the next set of possible states. Again, this is a Space vs. Time tradeoff

In practice, lexical analysis give you configuration options to choose between DFAs and NFAs-based implementations (for the user to choose between space-time tradeoffs).