Compilers
CSCI 4020U
This course is about communication. In particular, we are interested in the communication with machines and communication between machines.
Symbols and Strings

- An *alphabet* is a collection of *symbols*.
- Some alphabets are small
  - Morse code has two symbols
  - The English alphabet has 26 symbols.
- Some alphabets are quite large.
  - Literate Chinese alphabet has over 4000 symbols.
Symbols and Strings

- No matter how large an alphabet is, it must be finite.
- We cannot encode all meaningful messages using single symbols because there are at least infinitely many meaning messages.
- We need strings.
Some Definitions

An alphabet $\Sigma$ is an *finite set* of symbols.

A string $s$ is a *finite sequence* of symbols from some alphabet.

A language $L$ is a set of strings. Most useful languages are infinite sets.

The set of all possible strings from an alphabet $\Sigma$ is denoted $\Sigma^*$. 
Fact

$L$ is a language $\Rightarrow L \subseteq \Sigma^*$
More Definitions

A string with zero symbols is written $\varepsilon$ (epsilon).

A string is valid with respect to a language $L$ if $s \in L$. Otherwise, we say that $s$ is not valid w.r.t. $L$. 
Example: English vs French

$\Sigma_{EN} = \{ a, b, c, \ldots A, B, C, \ldots Z \} \cup \text{Puntunations}$

$\Sigma_{FR} = \{ a, à, b, c, d, e, è, é, f, \ldots z, A, À, B, C, D, E, \ldots Z \}$

$\cup \text{Puntunations}$

Bonjour à tous $\in$ French, so it’s a valid French string.

Bonjour à tous $\notin$ English, so it’s not a valid English string.
Communication

Sender

Idea

Encode

Common language

Understanding

Decode

Receiver
Challenges

Bootstrapping communication

- How do we have the sender and receiver agree on using a common language?
- What is the language used to communicate the language to be used?

Expressiveness of language

- What type of messages are permitted by the common language used between the sender and the receiver?
- Are there ideas that cannot be communicated by the language?
Computation

Turing Machine

- Tape contains strings of symbols.
- Control logic is generating strings of a specific language.
- The control logic itself is defined as a string in the language understood by the Universal Turing Machine.

Lambda Calculus

- Expressions are strings of a language.
- String rewriting rules generate more strings in the language.
Programming

Sender = Programmer

- **Idea** is an algorithm to solve some problem.
- **Message** is the source code of a particular *language* that implements the algorithm.

Receiver = Computer

- **The decoder** turns the message (source code) into computational instructions (also of some language)
- **Understanding** is the actual computation triggered by the programmer.
Some languages we care about

Python Language

```python
year_names = []
with open(filename, 'r') as baby_file:
    lines = baby_file.readlines()
    for line in lines:
        if '<h3 align="center">Popularity' in line:
            year = re.search('(\d{4})', line)
            print(year.group(0))
            continue

        rank_info = re.search('<td>(\d+)</td><td>(\w*)</td><td>(\w*)</td>', line)
        if rank_info is not None:
            print(rank_info.group(0))
            rank, boy, girl = rank_info.group(1), rank_info.group(2), rank_info.group(3)
            year_names.extend([boy + ' ' + rank,
                                girl + ' ' + rank])

extract_names = [year] + sorted(year_names)
print(extracted_names)

extract_names('baby1990.html')
```
Some languages we care about

Clojure

```clojure
(defn bfs [G vtx Q visitf vals]
  ;; Recursive implementation of breadth-first search
  (letfn [(bfs [G vtx Q visitf vals]
    (let [adj (get G vtx)]
      (if (and (empty? adj)
          (empty? Q))
        (conj vals (visitf (:val vtx)))
        (let [Q (into Q adj)]
          (recur G (peek Q) (pop Q) visitf (conj vals (visitf (:val vtx)))))))]
    (bfs G vtx Q visitf vals)))
```
Some languages we care about

C

```c
/* CP CHECK END RIGHT */
/* %%Function:CpCheckEndRight %%Owner:chie */
CP CpCheckEndRight(cp, cpAnchor, pflss, psel, psty, fExtend)
CP cp;
CP cpAnchor;
struct FLSS *pflss;
struct SEL *psel;
int *psty;
BOOL fExtend;
{

    /* check for special case: insert point will not be placed to the right of an end of paragraph */
    int chBreak = pflss->chBreak;
    if (cp == pflss->cpxMac && (chBreak == chEop ||
        ((*hwdxCur)->fPageView && chBreak == chSect) || chBreak == chTable
        || chBreak == chCR) &&
            (psel->fSelAtPara || (*psty <= stySent &&
                !fExtend || cp < cpAnchor/* backward extension */)))
```
Some languages we care about

JVM Bytecode

```
aload_0
new #3 <acceptanceTests/treecase_personOK/Main$A>
dup
new #8 <java/lang/Object>
dup
invokespecial #10 <java/lang/Object.<init>>
new #12 <java/lang/Integer>
dup
iconst_2
invokespecial #14 <java/lang/Integer.<init>>
invokespecial #17 <acceptanceTests/treecase_personOK/Main$A.<init>>
new #12 <java/lang/Integer>
dup
iconst_1
invokespecial #14 <java/lang/Integer.<init>>
invokespecial #17 <acceptanceTests/treecase_personOK/Main$A.<init>>
getstatic #20 <java/lang/System.out>
new #3 <acceptanceTests/treecase_personOK/Main$A>
dup
ew #8 <java/lang/Object>
dup
invokespecial #10 <java/lang/Object.<init>>
new #12 <java/lang/Integer>
dup
iconst_2
invokespecial #14 <java/lang/Integer.<init>>
invokespecial #17 <acceptanceTests/treecase_personOK/Main$A.<init>>
invokespecialvirtual #26 <java/io/PrintStream.println>
return
```
Some languages we care about

OP Code for CPU
The journey of source code

Graph adjacency list

```clojure
(extend-type AdjacencyList
  Graph
  (out-degree [vtx graph]
    (count (graph vtx)))
  (in-degree [vtx graph]
    (count (for [v graph :when (some #([:end vtx]) v)] v)))
  (bfs [G vtx Q visitf vals]
    ;; Recursive implementation of breadth-first search
    (letfn [(bfs [G vtx Q visitf vals]
      (let [adj ((:al G) vtx)]
        (if (and (empty? adj)
          (empty? Q))
          (conj vals (visitf (:val vtx)))
          (let [Q (into Q adj)]
            (recur G (peek Q) (pop Q) visitf (conj vals (visitf (:val vtx))))))))
      (bfs G vtx Q visitf vals))))
```
The journey of source code

```
{extend-type AdjacencyList
  Graph
  (out-degree [vtx graph]
    (count (graph vtx)))
  (in-degree [vtx graph]
```
The journey of source code

Abstract syntax tree (AST)

Compilation

JVM Bytecode

```
aload_0
1 new #3 <acceptanceTests/treeset_person0K/Main4A>
4 dup
5 new #8 <java/lang/Object>
8 dup
9 invokespecial #10 <java/lang/Object.<init>>
12 new #12 <java/lang/Integer>
15 dup
16 icast_2
```
The journey of source code

```
aload_0
new #3 <acceptanceTests/treeseet_personOK/Main:A>
dup
new #8 <java/lang/Object>
dup
invokespecial #10 <java/lang/Object.<init>>
new #12 <java/lang/Integer>
dup
iconst_2
```

JVM Bytecode

Code Generation

CPU Specific Instructions
Languages everywhere

- How many types of tokens should we have?
- What are the tokens?

Regular Languages
Languages everywhere

- What are the valid sequences of tokens?
- How should we organize tokens into semantic groups?

Context Free Languages
About this course

Languages and Parsing

1. Regular Languages
2. Regular Expression and Automata
3. Context Free Languages
4. Parse Trees
5. Parsing Algorithms

Programming

1. Computation by programming in interpreted languages
2. Interpreter construction
3. Computation by programming in compiled languages
4. Three address bytecode
5. Compiler construction
About this course

1. We use Antlr, a parser generator library for Java.

2. We program in Java.

3. We program in Kotlin.

4. We provide cloud-based computing environment.