Computer Language Processing (COMP 4901U)

Lionel Parreaux, HKUST

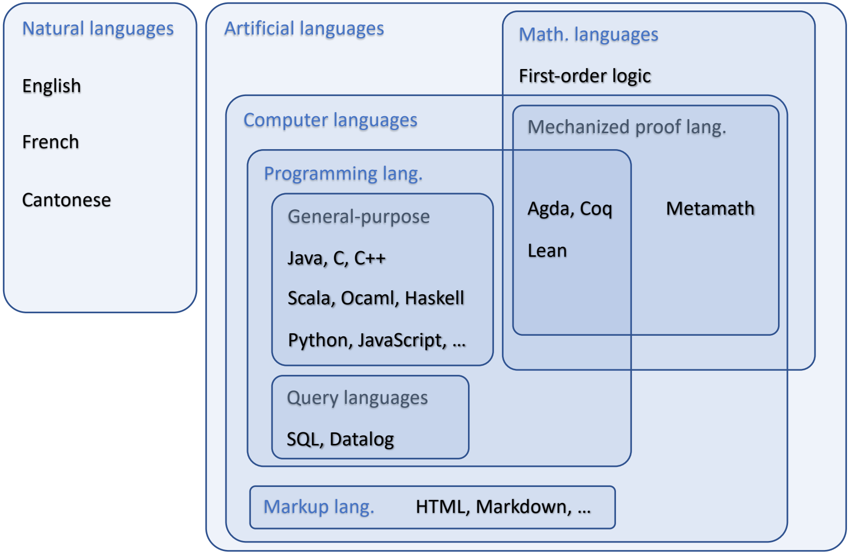
Includes contents adapted from Viktor Kuncak, EPFL.

What is COMP 4901U about?

Every aspect of processing ***languages meant to be processed***,
or ***computer languages***.

Particular focus on *programming languages*.

Computer Languages



Computer Language Processing

A **language** can be:

- natural language (English, French, . . .)
- **computer language** (Scala, Java, C, SQL, . . .)
- language for mathematics: $\forall \varepsilon . \exists \delta . \forall x . \ (|x| < \delta \Rightarrow |f(x)| < \varepsilon|)$

We can define languages mathematically as **sets of strings**

We can **process** languages: define algorithms working on strings

In this course we study algorithms to process computer languages

Interpreters and Compilers

We are particularly interested in processing general-purpose programming languages.

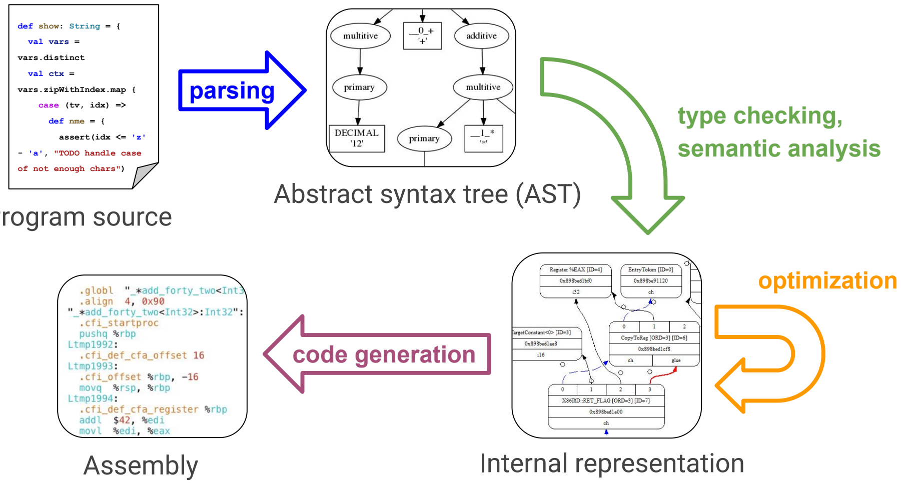
Two main approaches:

- interpreter: execute instructions while traversing the program (Python)
- compiler: traverse program, generate executable code to run later (Rust, C)

Portable compiler (Java, Scala, C#):

- compile (`javac`) to platform-independent **bytecode** (`.class`)
- use a combination of interpretation and compilation to run bytecode (`java`)
  - compile or interpret fast, determine important code fragments (inner loops)
  - **optimize** important code and swap it in for subsequent iterations

Typical Compiler Organization



```
def show: String = {
  val vars =
vars.distinct
  val ctx =
vars.zipWithIndex.map {
    case (tv, idx) =>
      def nme = {
        assert(idx <= 'z'
- 'a', "TODO handle case
of not enough chars")
```

```
  .globl  "_*add_forty_two<Int3
  .align  4, 0x90
"_*add_forty_two<Int32>:Int32":
  .cfi_startproc
  pushq %rbp
Ltmp1992:
  .cfi_def_cfa_offset 16
Ltmp1993:
  .cfi_offset %rbp, -16
  movq  %rsp, %rbp
Ltmp1994:
  .cfi_def_cfa_register %rbp
  addl  $42, %edi
  movl  %edi, %eax
```

Compilers for Programming Languages

A typical compiler processes a Turing-complete programming language and translates it into the form where it can be efficiently executed (e.g. machine code).

$$\begin{tikzcd}
\boxed{\text{Source code in a programming language}} \arrow[d, "\text{compiler}"] \\
\boxed{\text{machine code}}
\end{tikzcd}$$

- gcc, clang: map C into machine instructions
- Java compiler: map Java source into bytecodes (.class files)
- Just-in-time (JIT) compiler inside the Java Virtual Machine (JVM): translate .class files into machine instructions (while running the program)

Java compiler (javac) and JIT compiler (java)

```
class Counter {
 public static void main( ... ) {
   int i = 0; int j = 0;
   while (i < 10) {
     System.out.println(j);
     i = i + 2;
     j = j + 2*i + 1; }}}
```

$\downarrow$ `javac -g`

```
Counter.class bytecode

cafe babe 0000 0034
0018 0a00 0500 0b09
000c 000d 0a00 0e00
0f07 0010 0700 1101
```

$\xrightarrow{\texttt{java}}$

```
0
5
14
27
44
```

Inside a Java class file

```
class Counter {
 public static void main( … ) {
   int i = 0; int j = 0;
   while (i < 10) {
     System.out.println(j);
     i = i + 2;
     j = j + 2*i + 1; }}}
```

↓ javac

```
Counter.class bytecode

cafe babe 0000 0034
0018 0a00 0500 0b09
000c 000d 0a00 0e00
0f07 0010 0700 1101
```

$\xrightarrow{\text{javap -c}}$

```
 0: iconst_0
 1: istore_1
 2: iconst_0
 3: istore_2
 4: iload_1
 5: bipush 10
 7: if_icmpge 32
     …
21: iload_2
22: iconst_2
23: iload_1
24: imul
25: iadd
26: iconst_1
27: iadd
28: istore_2
29: goto 4
32: return
```

Compilers are Important

Source code (e.g. Scala, Java, C, C++, Python)

- designed to be easy **for programmers** (humans) to use
- should correspond to way programmers think and help them be productive: avoid errors, write at a **higher level**, use abstractions, interfaces

Target code (e.g. x86, arm, JVM, .NET)

- designed **to efficiently run on hardware**
- low level
- fast to execute, low power use

Compilers **bridge these two worlds**

- essential for building complex, performant software

Example Modern Compiler Technologies

Domain-aware compilers

Halide

image and array processing **DSL/compiler**

julia

numerical analysis and computational science **just-in-time compiler**

TensorFlow

XLA: **optimizing compiler** for machine learning

Futhark

purely functional data-parallel array **programming language** for the GPU

PyTorch

TorchScript **just-in-time compiler** for machine learning

tvm

deep learning **compiler stack**

Compiler Design Philosophies

Old way of building compilers

A group of guys with grey beards spend 20 years writing C code.

The result is considered final.

New way of building compilers

Open-ended, extensible *compiler frameworks* are developed as libraries.

Ported to many target architectures and new heterogeneous computing devices.

Some Skills and Knowledge Learned in the Course

- Develop a compiler for a simple functional language
  - Write a compiler from start to end
  - Generate WebAssembly code, which runs in browser or in nodejs
- Architect elegant software solutions in Scala
- Learn libraries to build compilers (e.g. parsing combinators)
  - Learn how to use *and* how to make them
- Analyze complex text formats and their semantics
- Automatically detecting errors in code:
  - type checking
  - abstract interpretation
- Foundations: regular expressions, grammars, parsing

Examples Uses of This Knowledge

- Understand how compilers work; *use* and *choose* them better
- Leverage new powerful tools for building complex software
- Design and implement your own language and compiler
- Extend existing languages through their compilers
- Analyze, process HTML pages & other computer languages
- Use extensible compiler frameworks to speed up parts of your applications
- Parse simple natural language fragments

Learning Scala

Scala is a powerful object-oriented and functional programming language, ideal for building compilers and interpreters.

Fine if you do not already know Scala.

Learn on the fly — the course material is adapted for it

Knowing Scala will probably make you a better developer.

Word-count: Java vs Scala

```java
public class WordCountJava {
    public static void main(String[] args) {
        StringTokenizer st
                = new StringTokenizer(args[0]);
        Map<String, Integer> map =
                new HashMap<String, Integer>();
        while (st.hasMoreTokens()) {
            String word = st.nextToken();
            Integer count = map.get(word);
            if (count == null)
                map.put(word, 1);
            else
                map.put(word, count + 1);
        }
        System.out.println(map);
    }
}
```

```
> runMain WordCountJava "a b a c a b"
[info] Running WordCountJava a b a c a b
{a=3, b=2, c=1}
```

```scala
object WordCountScala extends App {
 println(
   args(0)
   .split(" ")
   .groupBy(x => x)
   .map(t => t._1 -> t._2.length))
}
```

```
> runMain WordCountScala "a b a c a b"
[info] Running WordCountScala a b a c a b
Map(b -> 2, a -> 3, c -> 1)
```

Course Organization

- Lectures (~2h, mixed mode light)
  Learn general material.
- Tutorial (~2h, real-time online mode)
  Practice solving exercises.
- Lab (~2h, real-time online mode)
  Work on mini-projects and get help.

These time estimates are upper bounds.

Collaboration: Work *individually* for all mini-projects except last one.

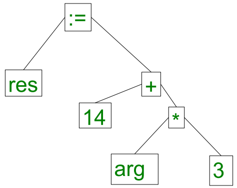
- I may ask you to explain specific parts of the code
- I use code plagiarism detection tools
- I will check whether you understand your code

Tentative course structure

- Introduction & review of formal languages
- Lexical analysis
- Syntactic analysis (parsing)
- Name analysis
- Type checking
- Type inference
- Code generation
- Optimization
- Extensible compilers and DSLs

Compilers Bridge the Source-Target Gap in Phases

characters	`res = 14 + arg * 3`
↓ lexical analyzer	
words	$\boxed{\texttt{res}}\ \boxed{\texttt{=}}\ \boxed{\texttt{14}}\ \boxed{\texttt{+}}\ \boxed{\texttt{arg}}\ \boxed{\texttt{*}}\ \boxed{\texttt{3}}$
↓ parser	
trees	Assign(res, Plus(C(14), Times(V(arg),C(3))))
↓ name analyzer	
graphs	(variables mapped to declarations)
↓ type checker	
graphs	Assign(res:Int, Plus(C(14), Times(V(arg):Int,C(3)))):Unit
↓ intermediate code generator	
intermediate code	e.g. LLVM bitcode, JVM bytecode, Web Assembly
↓ JIT compiler or platform-specific back end	
machine code	e.g. x86, ARM, RISC-V



Front End and Back End

front end

characters
↓ lexical analyzer
words
↓ parser
trees
↓ name analyzer
graphs
↓ type checker
graphs
↓ intermediate code generator
intermediate code
↓ JIT compiler or platform-specific back end
machine code   e.g. x86, ARM, RISC-V

back end

Benefits of modularity:

- do one thing in one phase
- swap different front-end: add languages
  (C or Rust, Java or Scala)
- swap different back-end: add various architectures
  (Linux on x86 and ARM)

Interpreters

$$\begin{tikzcd}
\text{characters} \arrow[d, "\text{lexical analyzer}"] & & \\
\text{words} \arrow[d, "\text{parser}"] & & \\
\text{trees} \arrow[d] & & \text{program input} \arrow[ll] \\
\text{program result} & &
\end{tikzcd}$$

Comparison to a compiler:

- same front end: front end techniques apply to interpreters
- no back end: compute result using trees and graphs

Program Trees are Crucial for Interpreters and Compilers

We call a program tree **Abstract Syntax Tree** (AST)

- All serious programming language implementations use ASTs

Structure of trees:

- Nodes represent arithmetic operations, statements, blocks
- Leaves represent constants, variables, methods

Representation of trees:

- Classes in object-oriented languages
- Algebraic data types in functional languages like Haskell, ML

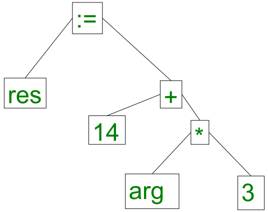
  (Scala is a mix of both!)

A Simple AST Definition in Scala

```
enum Expr:
  case C(n: Int) // constant
  case V(s: String) // variable
  case Plus(e1: Expr, e2: Expr)
  case Times(e1: Expr, e2: Expr)

enum Statement:
  case Assign(id: String, e: Expr)
  case Block(s: List[Statement])

val program = Assign("res", Plus(C(14), Times(V("arg"), C(3))))
```



Transforming Text Into a Tree

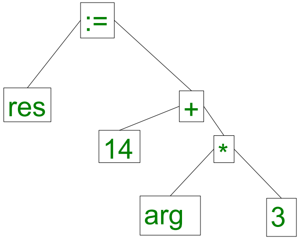
characters `res = 14 + arg * 3`

↓ lexical analyzer

words $\boxed{\texttt{res}}\ \boxed{\texttt{=}}\ \boxed{\texttt{14}}\ \boxed{\texttt{+}}\ \boxed{\texttt{arg}}\ \boxed{\texttt{*}}\ \boxed{\texttt{3}}$

↓ parser

trees Assign(res, Plus(C(14), Times(V(arg),C(3))))



First two phases:

1. lexical analyzer (lexer): sequence of characters → sequence of words
2. syntax analyzer (parser): sequence of words → tree

We will study *linear-time algorithms* for these problems.

We start with the underlying *theory of formal languages*.

Definition of Words in Set Theory

Let A be an alphabet $\{\text{a}, \text{b}, \text{c}, \ldots\}$

We define words of length n, denoted A^n, as follows:

$A^0 = \{\varepsilon\}$ (only one word of length zero, always denoted ε)

For $n > 0$, $A^n = \{\, \text{a}w \mid w \in A^{n-1} \,\}$

A non-empty word is just a letter followed by a smaller word.
We usually write single-letter words like ‘aε’ as just ‘a’.

Example: $w = \mathbf{1011}$, to be understood as $w = \mathbf{1}(\mathbf{0}(\mathbf{1}(\mathbf{1})))$
(we’ll see that parenthesization does not matter)

We sometimes refer to letters by index. $w_{(0)} = \mathbf{1}$ $w_{(1)} = \mathbf{0}$ $w_{(2)} = \mathbf{1}$ $w_{(3)} = \mathbf{1}$

Set of all words: $A^* = \bigcup_{n \geq 0} A^n$

which means: $w \in A^*$ if and only iff there exists n such that $w \in A^n$.

Word Equality

Words are equal when they are both empty, or when they are formed of the same letter followed by equal sub-words.

Let $u, v \in A^*$. Then $u = v$ if and only if either

1. $u = \varepsilon$ and $v = \varepsilon$; or
2. $u = \mathrm{a}u'$ and $v = \mathrm{a}v'$ where $u' = v'$ for some a, u', v'

Words as Inductive Structures

Theorem (Structural induction for words)

Given a property on words $P : A^* \to \{true, false\}$
If $P(\varepsilon)$ *and if, for every letter* $a \in A$ *and every* u, $P(u)$ *implies* $P(a \cdot u)$,
then $\forall u \in A^*.\ P(u)$.

Words as Scala Lists

```
enum List[A]: // A is the alphabet
  case Nil()
  case Cons(head: A, tail: List[A])

// Example:
val w = List.Cons('a', List.Cons('b', List.Nil()))

println(w) // prints Cons(a,Cons(b,Nil()))
```

Words as Scala Lists — Adding Methods

```scala
enum List[A]:
   case Nil()
   case Cons(head: A, tail: List[A])

   def length: Int = this match
      case Nil() ⇒ 0
      case Cons(h, t) ⇒ 1 + t.length

import List.*

// Example:
val w = Cons('a', Cons('b', Nil()))

println(w.length) // prints 2
```

Words as Scala Lists — Appension Shorthand

```scala
enum List[A]:
   case Nil()
   case Cons(head: A, tail: List[A])
import List.*

// Example:
val w = Cons('a', Cons('b', Nil()))

extension [A](x: A) def append(xs: List[A]): List[A] = Cons(x, xs)

val w = 'a'.append('b'.append(Nil()))

// Symbolic name for append is '::'
extension [A](x: A) def ::(xs: List[A]): List[A] = Cons(x, xs)

val w = 'a' :: 'b' :: Nil() // :: is right-associative
```

Concatenation

Concatenation is a fundamental operation on words, and denotes putting the words of one word after another. For example, concatenating words 01 and 10, denoted $01 \cdot 10$, results in the word 0110.

Definition

$$u \cdot v = \begin{cases} v & \text{if } u = \varepsilon \\ \mathsf{a}(u' \cdot v) & \text{if } u = \mathsf{a}u' \end{cases}$$

Note: it follows that $w \cdot \varepsilon = w$ and $\varepsilon \cdot w = w$. Also, $\mathsf{a} \cdot w = \mathsf{a}w$.

Often, by abuse of notation, we write just uv instead of $u \cdot v$.

Concatenation in Scala

```scala
enum List[A]:
  case Nil()
  case Cons(head: A, tail: List[A])

  def ++(that: List[A]): List[A] = this match
    case Nil() => that
    case Cons(h, t) => Cons(h, t ++ that)

val v = 1 :: 2 :: 3 :: Nil() // 123
val w = 9 :: 8 :: Nil() // 98

assert(
  v ++ w == 1 :: 2 :: 3 :: 9 :: 8 :: Nil() // 12398
)
```

Associativity of Concatenation

Theorem

For all $u, v, w \in A^*$, $\qquad u \cdot (v \cdot w) = (u \cdot v) \cdot w$

Proof?

Associativity of Concatenation

Theorem

For all $u, v, w \in A^*$, $\qquad u \cdot (v \cdot w) = (u \cdot v) \cdot w$

Proof?

By induction on u.

Case $u = \varepsilon$. Then $u \cdot (v \cdot w) = v \cdot w = (u \cdot v) \cdot w$ because $u \cdot v = v$.

Associativity of Concatenation

Theorem

For all $u, v, w \in A^*$, $\qquad u \cdot (v \cdot w) = (u \cdot v) \cdot w$

Proof?

By induction on u.

Case $u = \varepsilon$. Then $u \cdot (v \cdot w) = v \cdot w = (u \cdot v) \cdot w$ because $u \cdot v = v$.

Case $u = \mathrm{a}u'$. Then $u \cdot (v \cdot w) = \mathrm{a}u' \cdot (v \cdot w)$...
But how to show this is the same as $(u \cdot v) \cdot w = (\mathrm{a}u' \cdot v) \cdot w$?
By induction, we only know that $u' \cdot (v \cdot w) = (u' \cdot v) \cdot w$.

Associativity of Concatenation

Theorem

For all $u, v, w \in A^*$, $\qquad u \cdot (v \cdot w) = (u \cdot v) \cdot w$

Lemma

For all $\mathrm{a} \in A,\ u, v \in A^*$, $\qquad \mathrm{a}(u \cdot v) = \mathrm{a}u \cdot v$

Proof of lemma: by definition of concatenation.

$$\mathrm{a}u \cdot v = \begin{cases} v & \text{if } \mathrm{a}u = \varepsilon \\ \mathrm{a}'(u' \cdot v) & \text{if } \mathrm{a}u = \mathrm{a}'u' \end{cases} \qquad = \mathrm{a}'(u' \cdot v) = \mathrm{a}(u \cdot v)$$

Proof of theorem: by induction on u.
Case $u = \varepsilon$. Then $u \cdot (v \cdot w) = v \cdot w = (u \cdot v) \cdot w$ because $u \cdot v = v$.
Case $u = \mathrm{a}u'$. Then $u \cdot (v \cdot w) = \mathrm{a}u' \cdot (v \cdot w) = \mathrm{a}(u' \cdot (v \cdot w))$ by the lemma
and $(u \cdot v) \cdot w = (\mathrm{a}u' \cdot v) \cdot w = \mathrm{a}(u' \cdot v) \cdot w = \mathrm{a}((u' \cdot v) \cdot w)$ by applying the lemma twice
and by induction, $u' \cdot (v \cdot w) = (u' \cdot v) \cdot w$.

Free Monoid of Words

The neutral element and associativity law imply that the structure $(A^*, \cdot, \varepsilon)$ is an algebraic structure called *monoid*. The monoid of words is called the *free monoid*. Word monoid satisfies, among others, the following additional properties (which do not hold in all monoids).

Theorem (Left cancellation law)

For every three words $u, v, w \in A^*$, if $wu = wv$, then $u = v$.

Theorem (Right cancellation law)

For every three words $u, v, w \in A^*$, if $uw = vw$, then $u = v$.

Reversal

Reversal of a word is a word of same length with same symbols but in the reverse order.
Example: the reversal of the word 011, denoted $(011)^{-1}$, is the word 110.

Definition

Given $w \in A^*$, its reversal $w^{-1} = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ w'^{-1} \cdot \mathrm{a} & \text{if } w = \mathrm{a} \cdot w' \end{cases}$

From definition it follows that $\varepsilon^{-1} = \varepsilon$ and that $a^{-1} = a$ for all $a \in A$.

Theorem

For all $u, v \in A^*$, $(u^{-1})^{-1} = u$ *and* $(uv)^{-1} = v^{-1}u^{-1}$.

Every law about words has a dual version.
Here is the dual of induction principle, where we peel of last elements.

Theorem (Structural induction for words (dual))

Given a property of words $P : A^* \to \{true, false\}$, *if* $P(\varepsilon)$ *and, if for every letter* $a \in A$ *and every* u, *if* $P(u)$ *then* $P(u \cdot a)$, *then* $\forall u \in A^*. P(u)$.

Prefix, Postfix, and Slice

Definition

Let $u, v, w \in A^*$ such that $uv = w$. We then say that *u is a prefix of w* and that *v is a suffix of w*.

Definition

Given a word $w \in A^*$ and two integers p, q such that $0 \le p \le q \le |w|$, the *$[p, q)$-slice of w*, denoted $w_{[p,q)}$, is the word u such that $|u| = q - p$ and $u_{(i)} = w_{(p+i)}$ for all i where $0 \le i < q - p$.

Theorem

Let $w \in A^*$ and $u = w_{[p,q)}$ where $0 \le p \le q \le |w|$. Then the exist words $x, y \in A^*$ such that $|x| = p$, $|y| = |w| - q$, and $w = xuy$.

Theorem

Let $w, u, x, y \in A^*$ and $w = xuy$. Then $x = w_{[0,|x|)}$, $u = w_{[|x|,|x|+|u|)}$ and $v = w_{[|x|+|u|,|w|)}$.

Slice in Scala

$w \in A^*$, $0 \le p \le q \le |w|$, $[p,q)$-*slice of* w, denoted $w_{[p,q)}$, is u such that $|u| = q - p$ and $u_{(i)} = w_{(p+i)}$ for all i where $0 \le i < q - p$.

```
def slice(i: Int, j: Int): List[T] = {
   require(0 <= i && i <= j && j <= length)

   this match
     case Nil() ⇒ Nil()
     case Cons(h,t) ⇒
       if i == 0 && j == 0 then Nil()
       else if i == 0 then Cons(h, t.slice(0, j-1))
       else t.slice(i-1, j-1)

  } ensuring (_.size == j - i)

  // i.e.:
  }.ensuring(res ⇒ res.size == j - i)
```