Computer Language Processing (CS-320)

Viktor Kuncak, EPFL

https://lara.epfl.ch/w/cc

Computer Language Processing = ?

A **language** can be:

- natural language (English, French, . . .)
- **computer language** (Scala, Java, C, SQL, . . .)
- language used to write mathematical statements: $\forall \varepsilon . \exists \delta . \forall x . \ (|x| < \delta \Rightarrow |f(x)| < \varepsilon|)$

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

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).

$$\boxed{\text{Source code in a programming language}}$$

$$\downarrow \text{compiler}$$

$$\boxed{\text{machine code}}$$

- 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; }}}
```

↓ 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

```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; }}}
```

↓ javac

```
Counter.class bytecode

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

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** 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

Some Skills and Knowledge Learned in the Course

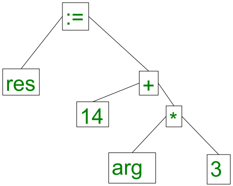
- Develop a compiler for a functional language
  - Write a compiler from start to end
  - Generates Web Assembly
  - generated code runs in browser or in nodejs
- libraries (e.g. parsing combinators) to build compilers: using and making them
- Analyze complex text
- Automatically detecting errors in code:
  - type checking
  - abstract interpretation
- (byte)code generation
- Foundations: automata, regular expressions, grammars, parsing

Examples of the Use of This Knowledge

- understand how compilers work, use them and choose them better
- gain experience with building complex software
- build compiler for your next great language
- extend language with a new construct you need
- adapt existing compiler to new target platform (e.g. embedded CPU or graphics processor)
- regular expression handling in editors and search tools
- analyze HTML pages
- process complex input boxes in your applications (make own spreadsheet software, expression evaluators)
- process LaTeX, build computer algebra system or a proof assistant
- parse simple natural language fragments

Compilers Bridge the Source-Target Gap in Phases

characters	`res = 14 + arg * 3`
↓ lexical analyzer	
words	`res` `=` `14` `+` `arg` `*` `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

back end

intermediate code
↓ JIT compiler or platform-specific back end
machine code e.g. x86, ARM, RISC-V

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

characters
$\downarrow$ lexical analyzer
words
$\downarrow$ parser
trees $\longleftarrow$ program input
$\downarrow$
program result

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)

- a language implementation today that does *not* use AST-s is a joke

Structure of trees:

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

Representation of trees:

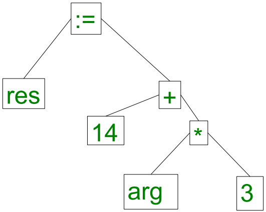
- classes in object-oriented languages
- algebraic data types in functional languages like Haskell, ML

A Simple AST Definition in Scala

```
abstract class Expression
case class C(n: Int) extends Expression // constant
case class V(s: String) extends Expression // variable
case class Plus(e1: Expression, e2: Expression) extends Expression
case class Times(e1: Expression, e2: Expression) extends Expression

abstract class Statement
case class Assign(id:String, e:Expression) extends Statement
case class Block(s: List[Statement]) extends Statement

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



Transforming Text Into a Tree

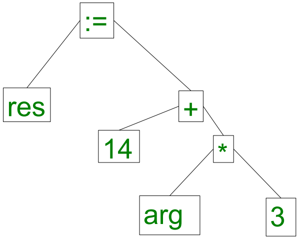
characters `res = 14 + arg * 3`

↓ lexical analyzer

words `res` `=` `14` `+` `arg` `*` `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*.