CS131 - Week 3

UCLA Spring 2019
TA: Kimmo Karkkainen

Today

- Currying & function types recap
- Grammars & derivations recap
- Old homework example (similar to HW2)
- Java Memory Model (brief introduction)

Currying & Types recap

Currying

- What is currying?

Currying

- What is currying?
- Currying = Function with multiple arguments can be expressed as multiple functions with one argument each, e.g.:

```
# let sum a b = a + b;;
val sum : int -> int -> int = <fun>

# let sum = fun a -> (fun b -> a + b);;
val sum : int -> int -> int = <fun>
```

Currying

- In your homework, make_matcher takes one argument and returns a matcher function that takes two more arguments -> Same as having one function that takes three arguments!

Partial application

- What is partial application?

Partial application

- What is partial application?
- Partial application = Providing only part of the arguments, thereby fixing the values in the function and creating a function with fewer arguments, e.g.:

```
# let sum = fun a -> (fun b -> a + b);;
val sum : int -> int -> int = <fun>

# let sum1 = sum 1;;
val sum1 : int -> int = <fun>

# sum1 2;;
- : int = 3
```

Partial application problems

What is the type of the following expression?

```
let rec f a = function
  | x when x mod a = 0 -> true
  | _ -> false;;
```

Partial application problems

What is the type of the following expression?

```
let rec f a = function
  | x when x mod a = 0 -> true
  | _ -> false;;
```

And after partial application? Also, what does this function do?

```
let g = f 2;;
```

Partial application problems

What is the type of the following expression?

```
let f a b = a b;;
```

Partial application problems

What is the type of the following expression?

```
let f a b = a b;;
```

How about after partial application? Also, what does this function do?

```
# f (fun x -> x + 1);;
```

Grammar recap

Derivation recap

- Recap of the top-down parsing technique:
  How to derive x y z x?

Grammar:

A -> x B x
B -> y
B -> y z

Derivation recap

- Recap of the top-down parsing technique:
  How to derive x y z x?

Grammar:

A -> x B x
B -> y
B -> y z

Current symbols:	**Apply rule:**
A	A -> x B x
x B x	B -> y
x y x	**BACKTRACK**
x B x	B -> y z
x y z x	**Match!**

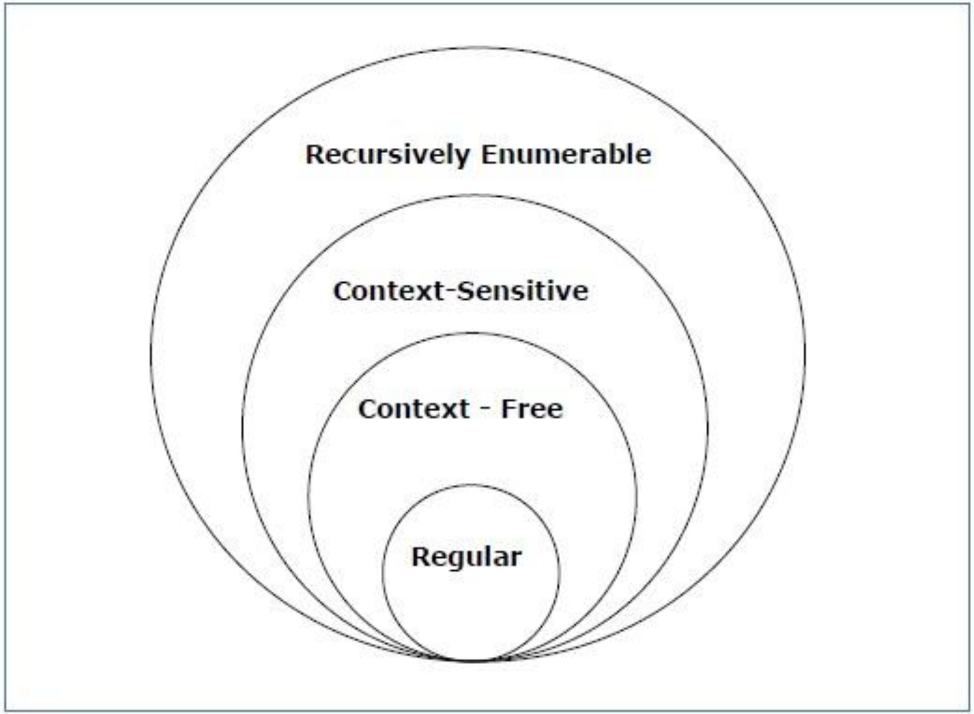
Old homework example

Old homework example

- You can use code from an old homework as the starting point for your solution: http://web.cs.ucla.edu/classes/winter19/cs131/hw/hw2-2006-4.html
- Task: Build a pattern matcher for genetic sequences
    - Similar to regular expressions
- Genetic sequence consists of letters: A, C, G, T (adenine, thymine, cytosine, and guanine
    - E.g. AGGTCAGTTACAATTGCTT...
    - These are the only allowed symbols in our language

Context-Free vs Regular Grammar

- Your homework deals with context-free grammars, the old homework example uses a regular grammar
- Regular grammars only allow rules of type S -> aS | a | ε
- Context-free grammars allow e.g. palindromes: S -> aSa | b
- Discussed more in CS181



Patterns

- *Frag [symbol list]*
  - Match a list of symbols, e.g. *Frag [C;T;G]* matches *[C;T;G]*
- *Junk k*
  - Matches up to *k* symbols, e.g. *Junk 1* matches *[], [A], [C], [T], [G]*
- *Or [pattern list]*
  - Matches any pattern in the list, e.g. *Or [Frag[C;T]; Frag[A;G]]* matches *[C;T]* and *[A;G]*
- *List [pattern list]*
  - Matches a concatenation of patterns, e.g. *List [Frag[A]; Junk 1; Frag[G]]* matches *[A;G], [A;A;G], [A;C;G], [A;T;G], [A;G;G]*
- *Closure pattern*
  - Matches a pattern 0 or more times, e.g. *Closure (Or [Frag[A];Frag[B]])* matches [], [A], [B], [A;A], [A;B], [B;B], [B;A], and so on

Pattern matching

- How to match *AAGC* using the following pattern?
- List [

```
    Frag [A];
    Or [
        Frag[T];
        Junk 2;
    ];
    Frag [G; C]
]
```

Pattern to context-free grammar - example

How would you express the following using a context-free grammar?

Frag [A; T; G; C]

Pattern to context-free grammar - example

How would you express the following using a context-free grammar?

Or [Frag [A; T]; Frag [G; C]; Frag [G; T]]

Pattern to context-free grammar - example

How would you express the following using a context-free grammar?

Closure (Frag [“A”; “T”])

Pattern to context-free grammar - example

How would you express the following using a context-free grammar?

```
List
[
  Frag [A; T];
  Junk 1;
]
```

make_matcher

- *make_matcher pattern* returns a matcher function for *pattern*
- Matcher takes a fragment and an acceptor, returns whatever the acceptor returns
- Similar to your matcher, except instead of a context-free grammar, it matches more limited patterns

make_matcher

```
let rec make_matcher = function
  | Frag frag -> make_appended_matchers match_nucleotide frag
  | List pats -> make_appended_matchers make_matcher pats
  | Or pats -> make_or_matcher make_matcher pats
  | Junk k -> match_junk k
  | Closure pat -> match_star (make_matcher pat)
```

Matching *Frag*

Frag [A; G; T]

```
  | Frag frag -> make_appended_matchers match_nucleotide frag
```

```
let match_nucleotide nt frag accept =
  match frag with
    | [] -> None
    | n::tail -> if n == nt then accept tail else None
```

Matching *Frag*

Frag [A; G; T]

```
  | Frag frag -> make_appended_matchers match_nucleotide frag
```

```
let match_nucleotide nt frag accept =
  match frag with
    | [] -> None
    | n::tail -> if n == nt then accept tail else None
```

```
let make_appended_matchers make_a_matcher ls =
  let rec mams = function
    | [] -> match_empty
    | head::tail -> append_matchers (make_a_matcher head) (mams tail)
  in mams ls
```

Matching *Frag*

Frag [A; G; T]

```
  | Frag frag -> make_appended_matchers match_nucleotide frag
```

```
let match_nucleotide nt frag accept =
  match frag with
    | [] -> None
    | n::tail -> if n == nt then accept tail else None
```

```
let make_appended_matchers make_a_matcher ls =
  let rec mams = function
    | [] -> match_empty
    | head::tail -> append_matchers (make_a_matcher head) (mams tail)
  in mams ls
```

```
let append_matchers matcher1 matcher2 frag accept =
  matcher1 frag (fun frag1 -> matcher2 frag1 accept)
```

Matching *Frag*

- Make_appended_matchers creates a chain of matchers
- Assuming our matcher tries to match Frag [A; G; T] with input [A; G; T]:

match_nucleotide A
Compare the first input symbol A to A

"acceptor"
Call next match_nucleotide with the remaining input symbols [G; T]

match_nucleotide G
Compare the first input symbol G to G

"acceptor"
Call next match_nucleotide with the remaining input symbols

match_nucleotide T
Compare the first input symbol T to T

"acceptor"
Call next match_empty with the remaining input symbols

match_empty
Call the original acceptor with the remaining input symbols

Original acceptor
Check that the suffix is acceptable

Matching *List*

List [Frag [A; G]; Junk 2]

```
| List pats -> make_appended_matchers make_matcher pats
```

- Same *make_appended_matchers* as previously, this time just used with *make_matcher* itself

```
let rec make_matcher = function
  | Frag frag -> make_appended_matchers match_nucleotide frag
  | List pats -> make_appended_matchers make_matcher pats
  | Or pats -> make_or_matcher make_matcher pats
  | Junk k -> match_junk k
  | Closure pat -> match_star (make_matcher pat)
```

Matching *Or*

Or [Frag [A; C]; Frag [G; T]]

```
| Or pats -> make_or_matcher make_matcher pats
```

```
let rec make_or_matcher make_a_matcher = function
  | [] -> match_nothing
  | head::tail ->
      let head_matcher = make_a_matcher head
      and tail_matcher = make_or_matcher make_a_matcher tail
      in fun frag accept ->
            let ormatch = head_matcher frag accept
            in match ormatch with
                  | None -> tail_matcher frag accept
                  | _ -> ormatch
```

Matching *Junk*

Junk 2

```
| Junk k -> match_junk k
```

```
let rec match_junk k frag accept =
  match accept frag with
    | None ->
        (if k = 0
        then None
        else match frag with
              | [] -> None
              | _::tail -> match_junk (k - 1) tail accept)
    | ok -> ok
```

Matching *Closure*

Closure (Frag [A; C])

```
  | Closure pat -> match_star (make_matcher pat)
```

```
let rec match_star matcher frag accept =
  match accept frag with
    | None ->
        matcher frag
            (fun frag1 ->
              if frag == frag1
              then None
              else match_star matcher frag1 accept)
    | ok -> ok
```

Old homework example

- Note the differences to your homework:
  - You need to match arbitrary context-free grammars, so the solution is more abstract
    - No need for specialized functions for all the different cases
  - Your parser should return the parse tree instead of what the acceptor returns
- Consider especially how make_appended_matchers and make_or_matcher are related to your homework
  - In what situation are you trying to match ***x and y*** vs ***x or y***?

Java

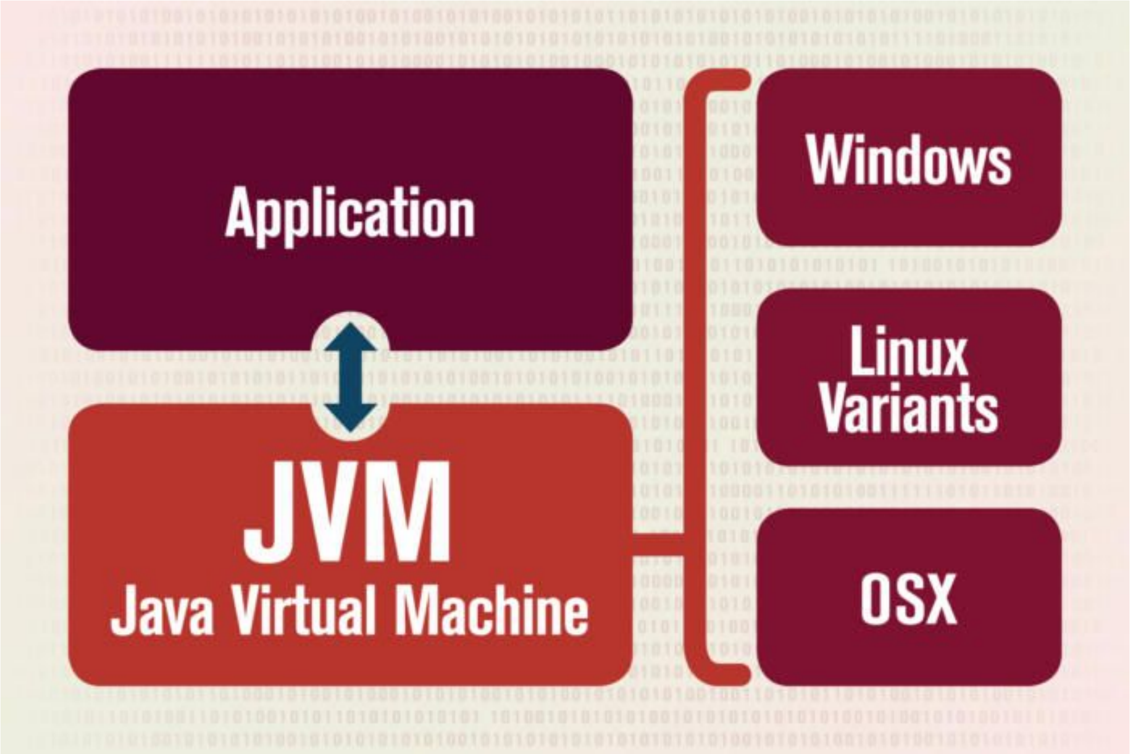
HW3

- HW3 due May 6th (2.5 weeks from now)
- Task: Comparing different ways of synchronizing multithreaded code
- Java and HW3 will be covered in more detail in 2 weeks, today just a brief introduction to Java Memory Model

Java Introduction

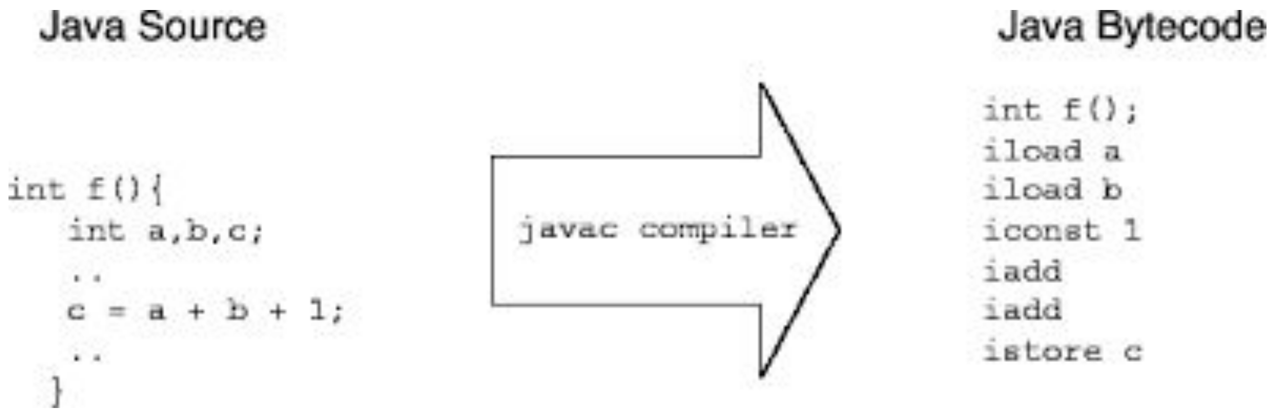
- General-purpose, object-oriented language
- One of the most popular programming languages
    - #2 most popular language on Github
    - #1 in Tiobe index
- Code compiled into bytecode and runs in a virtual machine

Java Virtual Machine (JVM)



Java Bytecode

- A compromise between compiled and interpreted code:
    - Platform independence
        - Compiled code runs on one specific platform (OS & CPU architecture)
    - Performance
        - Interpreted code is difficult to optimize



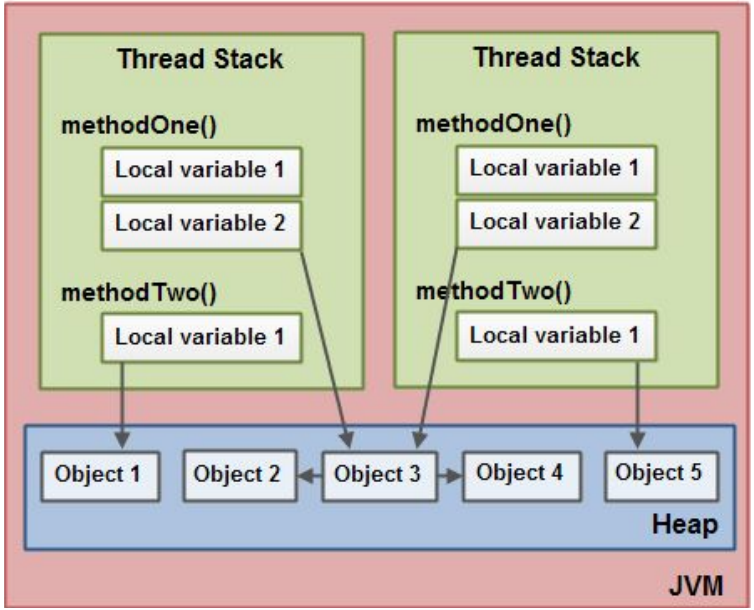
Hello, World!

```
public class HelloWorld {
	public static void main(String[] args) {
		System.out.println("Hello, World");
	}
}
```

Java Memory Model

Java Memory Model

- Defines how threads interact through memory
  - I.e. How multithreaded programs can behave in different situations



Java Memory Model

- “As-if-serial” semantics used within one thread
    - Compiler can change your code in any way as long as the result of execution is the same
    - E.g. **y = 1; x = 2;** vs **x = 2; y = 1;**
- Reasoning across multiple threads more challenging -> needs input from the programmer
    - Java provides multiple ways to set constraints on the order of execution

Problems with Concurrency

- What can go wrong with the following code?

Thread 1:

```
x = 5;
finished = true;
```

Thread 2:

```
while (!finished) { }
doSomething(x);
```

(+ many other problems, will be discussed more later...)

Synchronized keyword (Monitors)

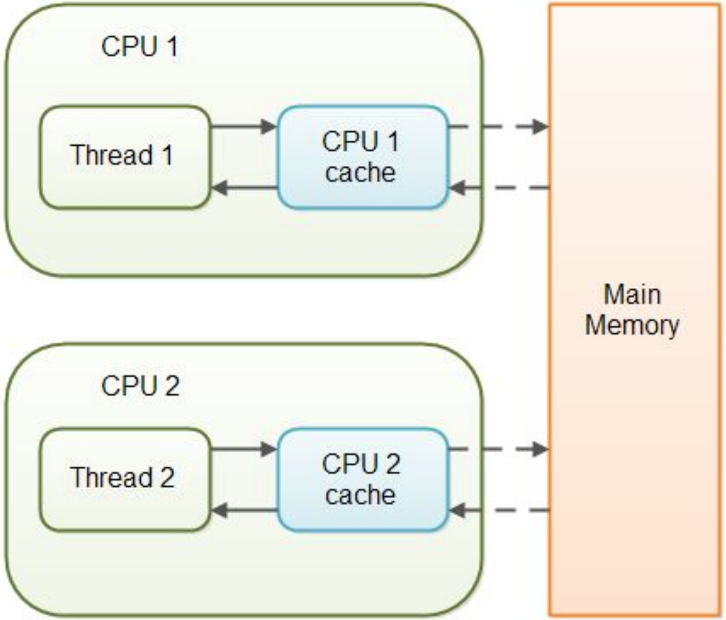
- If one thread is executing a synchronized method, all other threads have to wait before entering synchronized methods
  - Lock within one object - applies to all synchronized methods in that object

```java
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

Volatile

- Defining a variable *volatile* guarantees that other threads will see the changes immediately

```
public class SharedObject {
        public volatile int counter = 0;
}
```



Other synchronization approaches

- Atomic variables
- Locks
- VarHandle
- …

We'll discuss these more after the midterm

Questions?