In [1]:

```
# setup
from IPython.core.display import display,HTML
display(HTML('<style>.prompt{width: 0px; min-width: 0px; visibility: collapse}</style>'))
display(HTML(open('rise.css').read()))

# imports
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
sns.set(style="whitegrid", font_scale=1.5, rc={'figure.figsize':(12, 6)})
```

CMPS 2200

Introduction to Algorithms

Functional Programming

Today's agenda:

- What is functional programming?
- What's the connection between functional programming and parallelism?
- How do we perform computation in functional languages?

Specifying algorithms

We need a way to specify both **what** and algorithm does and **how** it does it.

Algorithm specification: defines what an algorithm should do.

> Given a sequence A of n elements, return a sequence B such that $B[i] \leq B[j]$ for all $0 \leq i \leq j \leq n$

May also include **cost specification**, e.g. $O(n \log n)$ work and. $O(\log^2 n)$ span.

Algorithm implementation (or just **algorithm**): defines **how** an algorithm works.

This could be real, working code, or pseudo-code.

Our textbook uses a pseudo code language called **SPARC**

- based on a functional language called ML.
- suitable for parallel algorithms
- good level of abstraction for talking about parallel algorithms

When possible, we will also show Python versions of key algorithms.

Functional languages

In functional languages, functions act like mathematical functions.

Two key properties:

1. function maps an input to an output $f : X \mapsto Y$
   - no **side effects**

1. function can be treated as values
   - function A can be passed to function B

Pure function

A function is **pure** if it maps an input to an output with no **side effects.**

A computation is **pure** if all of its functions are pure.

In [2]:

```
def double(value):
    return 2 * value

double(10)
```

Out[2]:

```
20
```

We can view the `double` function as a mathematical function, defined by the mapping:

$$\{(0,0),(1,2),(2,4),\ldots\}$$

versus...

In [3]:

```
def append_sum(mylist):
    return mylist.append(sum(mylist))

mylist = [1,2,3]
append_sum(mylist)
mylist
```

Out[3]:

```
[1, 2, 3, 6]
```

This has the side effecet of changing (or *mutating*) `mylist`.

though compare with...

In [4]:

```
def append_sum(mylist):
    return list(mylist).append(sum(mylist))

mylist = [1,2,3]
append_sum(mylist)
mylist
```

Out[4]:

```
[1, 2, 3]
```

Almost all "real" computations have some side effects. Consider:

In [5]:

```
def do_sum(mylist):
    total = 0
    for v in mylist:
        total += v
    return total
```

`do_sum` has the side effect of modifying `total`. But, this effect is not visible outside of `do_sum`, due to variable scoping.

> **benign effect:** a side-effect that is not observable from outside of the function.

A function with benign effects is still considered pure.

Why is pure computation good for parallel programming?

Recall our **race condition** example:

In [6]:

```python
from multiprocessing.pool import ThreadPool

def in_parallel(f1, arg1, f2, arg2):
    with ThreadPool(2) as pool:
        result1 = pool.apply_async(f1, [arg1])  # launch f1
        result2 = pool.apply_async(f2, [arg2])  # launch f2
        return (result1.get(), result2.get())   # wait for both to finish

total = 0

def count(size):
    global total
    for _ in range(size):
        total += 1

def race_condition_example():
    global total
    in_parallel(count, 100000,
                count, 100000)
    print(total)

race_condition_example()
```

```
188980
```

The `count` function has a side-effect of changing the global variable `total`.

Heisenbugs

- Race conditions can lead to bugs that only appear, e.g., 1 out of 1000 runs of the program.
- Reference to Heisenberg uncertainty principal (the bug disappears when you study it, but reappears when you stop studying it)

More generally, if we want to parallelize two functions $f(a)$ and $g(b)$, we want the same result **no matter which order they are run in.**

> Because of the lack of side-effects, pure functions satisfy this condition.

Data Persistence

In pure computation no data can ever be overwritten, only new data can be created.

Data is therefore always **persistent** —if you keep a reference to a data structure, it will always be there and in the same state as it started.

Isn't this horribly space inefficient?

> garbage collection

Functional languages

In functional languages, functions act like mathematical functions.

Two key properties:

1. function maps an input to an output $f : X \mapsto Y$
    - no **side effects**

1. function can be treated as values
    - function A can be passed to function B

Functions as values

Many languages allow functions to be passed to other functions.

Functions as "first-class values."

In [7]:

```python
def double(value):
    return 2 * value

def double_and_sum(double_fn, vals):
    total = 0
    for v in vals:
        total += double_fn(v)
    return total

# pass the function double to the function double_and_sum
double_and_sum(double, [1,2,3])
# 1*2 + 2*2 + 3*3
```

Out[7]:

```
12
```

`double_and_sum` is called a **higher-order function**, since it takes another function as input.

Why is this useful?

In [8]:

```python
def map_function(function, values):
    for v in values:
        yield function(v)

list(map_function(double, [1,2,3]))
```

Out[8]:

```
[2, 4, 6]
```

In [9]:

```python
def square(value):
    return value * value

list(map_function(square, [1,2,3]))
```

Out[9]:

```
[1, 4, 9]
```

In [10]:

```python
list(map_function(double, map_function(square, [1,2,3])))
```

Out[10]:

```
[2, 8, 18]
```

- If we know that `function` is pure, then we can trivially parallelize `map_function` for many inputs.
- By using higher-order functions, we can define a few primitive, high-order functions that will make it easier to reason about and analyze run-time of parallel computations.

Lambda Calculus

- pure language developed by Alonzo Church in the 1930s
- inherently parallel

Consists of expressions e in one of three forms:

1. a **variable**, e.g., x
2. a **lambda abstraction**, e.g., $(\lambda\, x\ .\ e)$, where e is a function body.
3. an **application**, written (e_1, e_2) for expressions e_1, e_2.

Beta reduction

A generic way of applying a sequence of anonymous functions.

$$(\lambda\, x\ .\ e_1)e_2 \mapsto e_1[x/e_2]$$

$e_1[x/e_2]$: for every free occurence of x in e_1, substitute it with e_2.

Can read as "substitute e_2 for x in e_1."

E.g.

In [11]:

```python
# lambda functions exist in Python.
# these are anonymous functions (no names)
# Here, e_2 is a variable.
(lambda x: x*x)(10)
```

Out[11]:

```
100
```

$(\lambda\, x\ .\ x * x)\ 10 \mapsto 10 * 10 \mapsto 100$

We can also chain functions together. E.g., e_2 can be another function.

In [12]:

```python
(lambda x: x*x)((lambda x: x+2)(10))
```

Out[12]:

```
144
```

beta reduction:

$(\lambda\, x\ .\ x * x)((\lambda\, x\ .\ x + 2)\ 10) \mapsto$
$(\lambda\, x\ . x * x)\ (10 + 2) \mapsto$
$(\lambda\, x\ . x * x)\ 12 \mapsto$
$(12 * 12) \mapsto$
144

Could we have done this in any other order?

$((\lambda\, x\ .\ x * x)(\lambda\, x\ .\ x + 2))\ 10 \mapsto$
$(\lambda\, x\, .\, (x + 2) * (x + 2))\ 10 \mapsto$
$(10 + 2) * (10 + 2) \mapsto$
144

Beta reduction order

$$(\lambda\, x\ .\ e_1)e_2 \mapsto e_1[x/e_2]$$

call-by-value: (first example): e_2 is evaluated to a value first, then reduction is applied

- square$(10 + 2)$
- square(12)
- $12 * 12$
- 144
- Languages that use call-by-value are called **strict**: argument is always evaluated before applying function

call-by-need: (second example): e_2 is first copied into e_1, then e_1 is evaluated.

- square$(10 + 2)$
- $(10 + 2) * (10 + 2)$
- 144

We will be using call-by-value, which is more amenable to parallelism.

Computation via beta reductions

Computation in lambda calculus means to apply beta reductions until there is nothing left to reduce.

When there is nothing left to reduce, the expression is in **normal form**.

How can we make an infinite loop in lambda computation?

$((\lambda\, x\ .\ (x\ x))(\lambda\, x\ .\ (x\ x))) \mapsto$

$((\lambda\, x\ .\ (x\ x))(\lambda\, x\ .\ (x\ x)))$

Doing computation via lambda calculus seems limiting.

Can it compute everything we need?

Surprising result:

The Lambda calculus is equivalent to Turing Machines.

That is...

Anything that can be computed by a Turing Machine can also be computed by the Lambda calculus, and visa versa.

More details:

- CMPS/MATH 3250
- **Church-Turing Thesis**