In [3]:

```
# 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

SPARC

Today's agenda:

- Overview of SPARC language
- Foundation of cost model framework

Why are we learning another language?

- allows us to specify parallel programs concisely
- allows us to analyze runtime of parallel programs
  - particularly for nested recursion
  - recall the recursive fork-join approach to sum an array (lec 2)

SPARC

- based on Standard ML
- functional language

Example SPARC program

$$
\begin{array}{l}
\textbf{let} \\
\quad x = 2 + 3 \\
\quad f(w) = (w * 4, w - 2) \\
\quad (y, z) = f(x - 1) \\
\textbf{in} \\
\quad x + y + z \\
\textbf{end}
\end{array}
$$

binding: associate entities (data or code) with identifiers.

let expression:

let
b^+
in
e
end

Expression e is applied using the bindings defined inside **let**.

expression *e*: describes a computation

- **evaluating** an expression produces its value

$$x = 2 + 3 = 5$$
$$f(4) \rightarrow (16, 2)$$
$$x + y + z = 5 + 16 + 2 = 23$$

value: irreducible unit of computation

- e.g.: $\mathbb{N}$, *true*, -, *and*
- *functions* are also values (it is a functional language)

SPARC supports lambda functions like:

$\texttt{lambda}\, x \,.\, x + 1$

$\texttt{lambda}\, (x, y) \,.\, x$

What do these do?

In [11]:
```
f1 = lambda x: x+1
f1(10)
```

Out[11]:
```
11
```

In [12]:
```
f2 = lambda x,y : x
f2(10,20)
```

Out[12]: 10

In [51]:
```
f2(100, 200)
```

Out[51]: 100

Function application

> A function application, $e_1 e_2$, applies the function generated by evaluating e1 to the value generated by evaluating e2.

E.g.,

- if e_1 evaluates to function $f(x)$
- e_2 evaluates to value v
- apply f to v by substituting v in for x

$\texttt{lambda}\ ((x,y)\ .\ x/y)\ (8,2)$

evaluates to 4

Composition

sequential composition: (e_1, e_2)

parallel composition: $(e_1 \mid\mid e_2)$

e.g.

$\texttt{lambda}\ (x,y).\ (x*x, y*y)$

vs

$\texttt{lambda}\ (x,y).\ (x*x \mid\mid y*y)$

In [44]:
```
def compose(g, f):
    """
    Returns a **function** that composes f and g
    """
    return lambda x: g(f(x))   # different from just: g(f(x))

def meter2cm(d):
    return d * 100

def cm2inch(d):
    return d / 2.54


# how many inches in a meter?
meter2inch = compose(meter2cm, cm2inch)
meter2inch(1)
```

Out[44]: 39.370078740157474

scoping and recursion

$x(p) = e$

vs

$x = \texttt{lambda}\, p.\, e$

When can x be referenced from e?

x is only visible from e when defined via the binding $x(p) = e$

This enables recursive expressions...

What does this do?

$$
\begin{array}{l}
\texttt{let} \\
\quad f(i) = \texttt{if}\ (i < 2)\ \texttt{then}\ i\ \texttt{else}\ i * f(i-1) \\
\texttt{in} \\
\quad f(5) \\
\texttt{end}
\end{array}
$$

In [6]:

```
factorial = lambda i: i if i < 2 else i*factorial(i-1)
factorial(5)
```

Out[6]:

```
120
```

Binary tree

We can also define datatypes recursively like:

$$\texttt{type}\ \mathit{tree} = \mathit{Leaf}\ \texttt{of}\ \mathbb{Z} \mid \mathit{Node}\ \texttt{of}\ (\mathit{tree}, \mathbb{Z}, \mathit{tree})$$

$$
\begin{array}{l}
\mathit{find}\,(t, x) = \\
\quad \texttt{case}\ t \\
\quad \mid \mathit{Leaf}\, y \Rightarrow x = y \\
\quad \mid \mathit{Node}\,(\mathit{left}, y, \mathit{right}) \Rightarrow \\
\quad\quad \texttt{if}\ x = y\ \texttt{then} \\
\quad\quad\quad \texttt{return true} \\
\quad\quad \texttt{else if}\ x < y\ \texttt{then} \\
\quad\quad\quad \mathit{find}\,(\mathit{left}, x) \\
\quad\quad \texttt{else} \\
\quad\quad\quad \mathit{find}\,(\mathit{right}, x)
\end{array}
$$

In [7]:

```
# translated into python...
class Tree:
```

```python
    def __init__(self, key, left=None, right=None):
        self.left = left
        self.key = key
        self.right = right
        self.is_leaf = left is None and right is None

t = Tree(4,
        Tree(2,
             Tree(1),
             Tree(3)
            ),
        Tree(5,
             Tree(6),
             Tree(7)
            )
        )

def find(t, x):
    print('find t=%d x=%d' % (t.key, x))
    if t.is_leaf:
        return t.key == x
    else:
        if x == t.key:
            return True
        elif x < t.key:
            return find(t.left, x)
        else:
            return find(t.right, x)

find(t, 7)
```

```
find t=4 x=7
find t=5 x=7
find t=7 x=7
```

Out[7]:

```
True
```

Pattern matching

Pattern matching is a way to do typical `if .. else` statements:

$$\begin{aligned}
&\mathit{find}\,(t, x) = \\
&\quad \texttt{case}\ t \\
&\quad \mid \mathit{Leaf}\, y \Rightarrow x = y \\
&\quad \mid \mathit{Node}\,(\mathit{left}, y, \mathit{right}) \Rightarrow \ldots
\end{aligned}$$

- Match t against each of the cases.
- When a match is found, evaluate the right hand side of $\Rightarrow$

What does this do?

$$\texttt{lambda}\ x\,.\,(\texttt{lambda}\ y\,.\,f(x, y))$$

Currying

> Convert a function of n variables into a sequence of functions with 1 argument each.

Why?

- Get specialized functions from more general functions by using composition.
- DRY: no need to repeat function arguments
- Lambda calculus: can define a programming language that only allows functions of one argument
  - easier for proofs!

E.g.,

$f(x, y) = x + y^2$

$\texttt{lambda}\, x\,.\,(\texttt{lambda}\, y\,.\,f(x, y))(10)(20) \rightarrow$
$\texttt{lambda}\, y\,.\,f(10, y)(20) \rightarrow$
$\texttt{lambda}\, y\,.\,(10 + y^2)(20) \rightarrow$
$10 + 20^2 \rightarrow$
410

In [9]:

```
def curry(f):
    """
    Given a function f of two variables,
    return a function g that binds the first variable
    and returns a function of the second variable.
    """
    def g(x):       # nested function 1
        def h(y):   # nested function 2
            return f(x, y)
        return h # bind x and return function of y
    return g

def f(x,y):
     return x + y**2

print(f)
print(curry(f))          # returns f'n g. input: x, output function of y
print(curry(f)(10))      # returns f'n h. input: y, output f(10,y)
print(curry(f)(10)(20)) # returns f(10,20)
print(curry(f)(10)(3))   # returns f(10,3)
```

```
<function f at 0x10444c488>
<function curry.<locals>.g at 0x10444c598>
<function curry.<locals>.g.<locals>.h at 0x10444c730>
410
19
```

Next lecture we will see how SPARC will allow us to analyze the cost of an algorithm.