### Lecture #2: Functions, Expressions

**Administrative**
- Reader with discussion and other materials available at Vick Copy (Euclid and Hearst).
- Sign yourself up on Piazza. See course web page: http://inst.cs.berkeley.edu/~cs61a
- Be sure to get an account form next week in lab, and provide registration data.

**Announcement:** We're trying to hire a new lecturer. There will be two candidates coming Jan. 27–28 (Josh Hug) and Feb. 3–4 (John DeNero), and you can help evaluate them! For both days:
  - Mon 01:00pm-02:00pm “Big ideas” talk (in Woz)
  - Tue 11:45am-12:45pm Undergrad student lunch on northside (meet in 777 Soda)
  - Tue 01:00pm-02:00pm Demo Class talk (in 380 Soda for Josh, Woz for John)
  - UG Tue 02:00pm-02:45pm Open Session after demo class (same rooms)

**Recap**
- From last lecture: **Values** are data we want to manipulate and in particular,
- **Functions** are values that perform computations on values.
- **Expressions** denote computations that produce values.
- Today, we'll look at them in some detail at how functions operate on data values and how expressions denote these operations.
- As usual, although our concrete examples all involve Python, the actual concepts apply almost universally to programming languages.

### Functions
- Something like `abs` denotes or evaluates to a function.
- To depict the denoted function values, we sometimes use this notation: `abs(x):` and `add(a, b):`
  - Idea: The opening on the left takes in values and one on the right delivers results.
  - The (green) formal parameter names—such as `x`, `a`, `b`—show the number of parameters (inputs) to the function.
  - The list of formal parameter names gives us the function’s signature—in Python, this is the number of arguments.
  - For our purposes, the blue name is simply a helpful comment to suggest what the function does.
  - (Python actually maintains this intrinsic name and the parameter names internally, but this is not a universal feature of programming languages, and, as you’ll see, can be confusing.)

### Functions: Lambda
- I’m often going to use a more venerable notation for function values: `λ x: ≪ | x | ≫` and `λ a, b: ≪ the sum of a and b ≫`
  - Formal parameters go to the left of the colon.
  - The part to the right of the colon is an expression that indicates what value is produced.
  - I’ll use `≪ · · · ≫` expressions to indicate non-Python descriptions of values or computations.
  - In Python, you can denote simple function values like this: `lambda a, b: ≪ the sum of a and b ≫` which evaluates to `λ a, b: ≪ the sum of a and b ≫`
  - (Well, OK: the `≪ · · · ≫` isn’t really Python, but I’ll use it as a placeholder for some computation I’m not prepared to write.)

### Calling Functions (I)
- The fundamental operation on function values is to **call** or **invoke** them, which means giving them one value for each formal parameter and having them produce the result of their computation on these values:
  - `-5 ≫ abs(number):` and `5`
  - `(29, 13) ≫ add(left, right)` and `42`

### Call Expressions
- A call expression denotes the operation of calling a function.
- Consider `add(2, 3):`
  - `add(Operator, Operand 0, Operand 1)`
  - The operator and the operands are all themselves expressions (recursion again).
  - To evaluate this call expression:
    - Evaluate the operator (let’s call the value `C`). It must evaluate to a function.
    - Evaluate the operands (or actual parameters) in the order they appear (let’s call these values `P_0` and `P_1`)
    - Call `C` with parameters `P_0` and `P_1`
Calling a Function (I): Substitution

- Once we have the values for the operator and operands, we must still actually evaluate the call.
- A simple way to understand this (which will work for simple expressions) is to think of the process as substitution.
- Once you have a value:
  \[ \lambda a, b: \text{sum of } a \text{ and } b \]
  and values for the operands (let's say 2 and 3),
- substitute the operand values for the formal parameters, replacing the whole call with
  \[ \text{sum of } 2 \text{ and } 3 \]
- which in turn evaluates to 5.

Example: From Expression to Value

Let's evaluate the expression \[ \text{mul(add(2, mul(0x4, 0x6)), add(0x3, 005))} \].

In the following sequence, values are shown in boxes.

Everything outside a box is an expression.

- \[ \text{mul(add(2, mul(0x4, 0x6)), add(0x3, 005))} \]
- \[ \lambda a, b: \text{sum of } a \text{ and } b \]
- \[ \lambda a, b: \text{sum of } a \text{ and } b \] (add(2, mul(0x4, 0x6)), add(0x3, 005))
- \[ \lambda a, b: \text{sum of } a \text{ and } b \] (add(2, mul(0x4, 0x6)), add(0x3, 005)) (2)
- \[ \lambda a, b: \text{sum of } a \text{ and } b \] (add(2, mul(0x4, 0x6)), add(0x3, 005)) (4) 5
- \[ \lambda a, b: \text{sum of } a \text{ and } b \] (add(2, mul(0x4, 0x6)), add(0x3, 005)) (4) 5
- \[ \lambda a, b: \text{sum of } a \text{ and } b \] (add(2, mul(0x4, 0x6)), add(0x3, 005)) (4) 5
- \[ \lambda a, b: \text{sum of } a \text{ and } b \] (add(2, mul(0x4, 0x6)), add(0x3, 005)) (4) 5
- \[ \lambda a, b: \text{sum of } a \text{ and } b \] (add(2, mul(0x4, 0x6)), add(0x3, 005)) (4) 5
- \[ \lambda a, b: \text{sum of } a \text{ and } b \] (add(2, mul(0x4, 0x6)), add(0x3, 005)) (4) 5
- \[ \lambda a, b: \text{sum of } a \text{ and } b \] (add(2, mul(0x4, 0x6)), add(0x3, 005)) (4) 5
- \[ \lambda a, b: \text{sum of } a \text{ and } b \] (add(2, mul(0x4, 0x6)), add(0x3, 005)) (4) 5
- ... \[ \lambda a, b: \text{sum of } a \text{ and } b \] (add(2, mul(0x4, 0x6)), add(0x3, 005)) (4) 5
- ... 208

Puzzle I

Evaluate \[ (\lambda a: \lambda b: a + b)(1)(3) \]
- First, must understand how it's grouped:
  \[ (\lambda a: \lambda b: a + b)(1)(3) \]
- ... and then you get into long, technical explanations about how the second "2" is really in a different language than the first, and actually is just another notation for some mystical Platonic "2" that is floating off somewhere.
- I'll just try to be practical and distinguish values from literals by surrounding values in a boxes: the value of 2 is \[ [2] \]
- One way to see the distinction between literals and values: the literals 0x10 and 16 are obviously different, but both denote the same value: \[ [16] \]

Puzzle I (cont'd.)

- \[ (\lambda a: \lambda b: a + b)(1)(3) \]
- \[ \lambda a: \lambda b: a + b \]
- \[ (\lambda b: 1 + b)(3) \]
- \[ \lambda b: 1 + b \]
- \[ 1 + 3 \]
- \[ 4 \]

Impure Functions

- The functions so far have been pure: their output depends only on their input parameters' values, and they do nothing in response to a call but compute a value.
- Functions may do additional things when called besides returning a value.
- We call such things side effects.
- Example: the built-in print function:
  \[ \text{print}(-5) \]
  \[ \rightarrow \text{display text '}-5'\]
- Displaying text is print's side effect. It's value, in fact, is generally useless (always the null value).
- For this lecture (at least), I'll use \[ '!' \] ("lambda bang") to denote function values with side effects.
Example: Print

What about an expression with side effects?
1. \( \text{print}(1), \text{print}(2) \)
2. \( \lambda x: \langle \text{print } x \rangle (\lambda x: \langle \text{print } x \rangle (1), \text{print}(2)) \)
3. \( \lambda x: \langle \text{print } x \rangle (\text{None}, \text{print}(2)) \)
   and print 1.
4. \( \lambda x: \langle \text{print } x \rangle (\text{None}, \lambda x: \langle \text{print } x \rangle (2)) \)
5. \( \lambda x: \langle \text{print } x \rangle (\text{None}, \text{None}) \)
   and print 2.
6. \( \text{None} \)
   and print 'None None'.