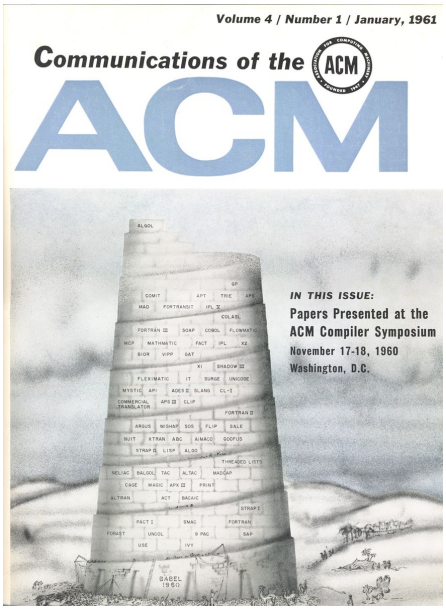
# Programming Languages

CSE 3302, Summer 2019
Lambdas




## *Lambdas (and Java)*

## Lambda Calculus

- The *lambda calculus* is a *formal system* in mathematics for expressing computation based on *functional abstraction* and *variable binding and substitution*.
- It is a *universal* model of computation that can be used to simulate any Turing machine.
- Invented by Alonzo Church in the 1930s as part of his research into the foundations of mathematics.

https://en.wikipedia.org/wiki/File:Alonzo_Church.jpg

## Lambdas in Programming Languages

- Generally equated with the idea of *anonymous* functions
  - … used as arguments to a *higher order function*.
    - That is, a *function* that operates on *functions*.
  - … used for constructing the result of a higher order function.
- Introduced in programming by Lisp (1958).
- Now ubiquitous!
  - C (non-standard extension of GCC), C++, C#, Java, Python, …

## [ Lambda Examples—C ]

- "Lambdas" do not exist in standard C, but GCC has a couple of non-standard extensions that get one kind of close.
  - *Nested Functions*
    - https://gcc.gnu.org/onlinedocs/gcc/Nested-Functions.html
  - *Statements and Declarations in Expressions*
    - https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html
- With these two extensions, "anonymous" functions are possible.

## [ Lambda Examples—C ]

- The "`lambda`" macro creates a block `{}` inside an expression statement `()`.
- Inside the block, a function is declared using the (dummy) name `_fn___`.
- A pointer to that function is returned.

C

```c
#include <stdio.h>

//--------------------------------------
#define lambda(return_type, function_body) \
({ return_type _fn___ function_body _fn___; })

//--------------------------------------
int main()
{
  int (*max)(int, int) =
    lambda(
      int,
      (int x, int y) {
        return x > y ? x : y;
      } );

  printf( "%d\n", max(4, 5) );
}
```

## [ Lambda Examples—C++ ]

- `auto` indicates the type should be deduced, as with a generic.
- The brackets [] mark the 'capture' list: objects visible from the environment.
- The parentheses mark the lambda's argument list.
- Available since C++11, extended in C++14.

*C++*

```
#include <iostream>

//---------------------------------------
auto maxi = [](auto x, auto y)
{
  return x > y ? x : y;
};

//---------------------------------------
using namespace std;

int main()
{
  cout << maxi( 4, 5 ) << endl;
}
```

## [ Lambda Examples—Python ]

- Lambdas in Python ① are somewhat limited as the body is restricted to a single expression.
- Python prefers that one declare a named function ② in the same scope.

①

```
def _main() :
  maxi = lambda x, y : x if x > y else y

  print( maxi( 4, 5 ) )

if __name__ == '__main__' :
  _main()
```

②

```
def _main() :
  def maxi( x, y ) :
    return x if x > y else y

  print( maxi( 4, 5 ) )

if __name__ == '__main__' :
  _main()
```

## [ *Is* `lambda` *Pythonic?* ]

- Python's `lambda` is often looked down on as not "Pythonic".
  - That is, not good Python programming style.
- Why not?
- Well, even without `lambda`, Python is already a dynamic, flexible language that supports first class functions.
- Many `lambda` use cases in other languages already have natural ways of expression in Python without `lambda`.
- Using `lambda` is not necessarily *bad*, but don't *misuse* it.

## [ *Is* `lambda` *Pythonic?* ]

***Q***: What feature of Python are you least pleased with?

***A***: Sometimes I've been too quick in accepting contributions, and later realized that it was a mistake. One example would be some of the functional programming features, such as `lambda` functions. `lambda` is a keyword that lets you create a small anonymous function; built-in functions such as `map`, `filter` and `reduce` run a function over a sequence type, such as a list.

In practice, it didn't turn out that well. Python has only two scopes: local and global. This makes writing `lambda` functions painful, because you often want to access variables in the scope where the `lambda` was defined, but you can't because of the two scopes. There's a way around this, but it's something of a kludge. Often it seems much easier in Python just to use a `for` loop instead of messing around with `lambda` functions. `map` and friends work well only when a built-in function that does what you want already exists.

**Guido van Rossum**
Creator of Python and Python's BDFL from 1995 to 2018

https://en.wikipedia.org/wiki/Guido_van_Rossum https://www.linuxjournal.com/article/2959

## [ *Is* `lambda` *Pythonic?* ]

- That interview is old (November 1998) and Python has evolved greatly since then.
  - E.g., there are now *nested scopes*, which resolves Guido's main objection.
- Interestingly, when Python moved from version 2 to version 3 (December 2008), `lambda` was *not* removed.
- Trey Hunner has a *great* discussion of this topic:

  https://treyhunner.com/2018/09/stop-writing-lambda-expressions/

## Java Lambda

- Java *lambdas* have two main purposes,
  - They enable the implementation of a method interface using an expression representation.
    - Though one can write the body of a lambda as a code block.
  - They increase the utility of collections in Java by making it much easier to iterate through, filter, and select data from a collection.
    - There are other ways to do these things in Java, but lambdas are a much better solution.

# Java Lambda

- Some examples of lambda in Java,
  - `() -> 42`
    - Takes no arguments and returns 42 (always!).
  - `( String s ) -> { System.out.println( s ); }`
    - Takes one argument and prints it out. The body of this lambda is a *code block* and not an expression. Returns *nothing*.
  - `( Integer i ) -> -i`
    - Takes one argument and returns its negation.
  - `( Integer i, Integer j ) -> i+j`
    - Takes two arguments and returns their sum.

# Java Lambda Examples

- Notice how the lambdas are defined and used.
- For no arguments, `.get()` to execute.
- For no return, `.accept()` to execute.
- For supplying arguments, `.apply(...)` to execute.

```
import java.util.function.Consumer;
import java.util.function.Supplier;
import java.util.function.UnaryOperator;
import java.util.function.BinaryOperator;

public class sampleLambda {
 static Supplier<Integer>
   forty_two = () -> 42;

 static Consumer<String>
   dump = ( String s ) -> { System.out.println( s ); };

 static UnaryOperator<Integer>
   negate = ( Integer i ) -> -i;

 static BinaryOperator<Integer>
   add = ( Integer i, Integer j ) -> i+j;

 public static void main( String[] args ) {
   System.out.format( "forty_two() is %d\n", forty_two.get() );
   dump.accept( "Dumped String" );
   System.out.format( "negate( 1 ) is %d\n", negate.apply( 1 ) );
   System.out.format( "add( 2, 3 ) is %d\n", add.apply( 2, 3 ) );
 }
}
```

# Java Lambda Examples

- The `sampleLambda` example compiles fine and gets just the answers we expect.

```
dalioba@Hoong:~/Desktop/Lambdas$ javac sampleLambda.java
dalioba@Hoong:~/Desktop/Lambdas$ java sampleLambda
forty_two() is 42
Dumped String
negate( 1 ) is -1
add( 2, 3 ) is 5
dalioba@Hoong:~/Desktop/Lambdas$
```

# Recursive Java Lambda Examples

- One *cannot* use (directly) the name of a lambda in its initializer, as in ①.
- The name `wrong` is not visible until *after* the declaration.
- Use a *qualified* name instead, as in ②.

①

```
import java.util.function.UnaryOperator;

public class wrongLambda {
 static UnaryOperator<Integer>
   wrong = ( Integer i ) -> i == 0 ? 0 : i + wrong.apply( i-1 );

 public static void main( String[] args ) {
   System.out.format( "wrong( 10 ) is %d\n", wrong.apply( 10 ) );
 }
}
```

②

```
import java.util.function.UnaryOperator;

public class okLambda {
 static UnaryOperator<Integer>
   ok = ( Integer i ) -> i == 0 ? 0 : i + okLambda.ok.apply( i-1 );

 public static void main( String[] args ) {
   System.out.format( "ok( 10 ) is %d\n", ok.apply( 10 ) );
 }
}
```

## Recursive Java Lambda Examples

- The `wrongLambda` case gets a compile time error because it's trying to use a name that is not yet available.

```
dalioba@Hoong:~/Desktop/Lambdas$ javac wrongLambda.java
wrongLambda.java:5: error: self-reference in initializer
   wrong = ( Integer i ) -> i == 0 ? 0 : i + wrong.apply( i-1 );
                                             ^
1 error
dalioba@Hoong:~/Desktop/Lambdas$
```

## Recursive Java Lambda Examples

- The `okLambda` case works because it uses a *qualified* name that *is* available.

```
dalioba@Hoong:~/Desktop/Lambdas$ javac okLambda.java
dalioba@Hoong:~/Desktop/Lambdas$ java okLambda
ok( 10 ) is 55
dalioba@Hoong:~/Desktop/Lambdas$
```