ECE326

PROGRAMMING LANGUAGES

Lecture 23 : Introduction to Functional Programming

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2020

Functional Programming

- Writing declarative expressions
  - No statements, everything is an expression
  - Program forms a tree of expressions
- Functions are first class citizens
- Uses higher order functions instead of control flow
  - No iteration, only recursion
- Non-strict (lazy) evaluation
  - Do not evaluate unless value is needed

Strict Evaluation

- Also known as eager or greedy evaluation
- Expression evaluated as soon as it's assigned

```
x = 2 + 3   # 2 + 3 is evaluated before assigned to x
```

- Evaluation goes from left to right, inside out
  - Function arguments evaluated before function
- Used by most imperative programming languages
  - Including Python, Rust, C++

Non-Strict Evaluation

- Also known as lazy evaluation or call-by-need
- Only evaluate if value is needed

```
# this will not crash with lazy evaluation!
x = len([2, 3, 4/0, 5])
```

Calculating length of a list does not require knowing the values within the list.

- Other examples
  - Short circuit evaluation

```
True or foo() # foo will never be called
```

  - Lazy iterable
    - Produces values on demand. e.g. range, zip, map, etc. in Python

Anonymous Function

- In Rust, closures are also anonymous
  - Name of the closure is unspecified
- In Python, anonymous functions are called lambdas
  - Can take any number of arguments, but only one expression
  - Closure can be named or nameless in Python

```
>> func = lambda x: x + 1
>> func(2)
3

>> full_name = lambda first, last: "%s %s"%(first, last)
>> full_name("Hello", "World")
'Hello World'
```

Higher Order Function

- `map`
  - Applies function, e.g. *P*(x), to all elements
  - Returns a list of results *in the same order*

```
# function: P(x)
def map(function, iterable):
    output = []
    for element in iterable:
        output.append(function(element))
    return output

# map is a lazy iterable. Must put in list to see result.
list(map(lambda x: x*x, range(1, 5)))
# [1, 4, 9, 16]
```

Higher Order Function

- `filter` function:
  - From input iterable, returns only elements that satisfies *F*(x)
    - i.e. when *F*(x) returns true
    - Original ordering is preserved

```
# function: F(x)
def filter(function, iterable):
    output = []
    for element in iterable:
        if function(element): # if F(x) is true
            output.append(element)
    return output

list(filter(lambda x: x%2 != 0, range(10)))
# [1, 3, 5, 7, 9]
```

Pure Function

- Output solely determined by input to function
  - Same return value for same arguments

```
x = 5
def foo():          # return value of foo() changes if x changes
    return x        # therefore foo is impure
```

  - Cannot have *side effects*
- Side effect
  - Changing states outside of local environment
    - E.g. modifying non-local variables, perform I/O, etc.

```
printf("%d", 5); // printf is impure because it writes to console
```

Pure Functional Programming

- Example: Haskell
- A subset of functional programming
  - All functions must be *pure*
  - All variables must be *immutable*
- Referential transparency
  - Replacing expression by its corresponding value does not change program behaviour
    - Guaranteed from a pure function
  - Enables compiler optimization
    - E.g. memoisation, parallel computing

Constant Expression

- Can be evaluated at compile time

```
int a = 5 + 7; // compiler would generate a = 12 directly
```

- constexpr keyword
  - Declares a compile-time variable, function, or class
    - May not exist at runtime (unlike constant variables)
  - Variable
    - Can only be initialized constant expression
  - Function
    - Arguments must only be constant expression

Constexpr Function

- Tells compiler to evaluate function at compile time
- Can significantly increase compile time
  - Compiler must ensure computation cannot crash itself
    - Performs extensive type-checking
    - E.g. Array out of bound check
- C++11
  - Restrictive on what's allowed in a constexpr function
    - No loops – must rely on recursion
    - Exactly one return statement allowed in body
    - No local variables, arguments only

Constexpr Function

```
constexpr int factorial(int n) {
    return n <= 1 ? 1 : (n * factorial(n - 1));
}

/* lexicographical comparison of two constant strings */
/* returns positive if a > b, negative if a < b, 0 if equal */
constexpr int constcmp(const char * a, const char * b) {
    return (a[0] == '\0') ? (a[0] - b[0]) :
    ( (a[0] == b[0]) ? constcmp(a+1, b+1) : a[0] - b[0] );
}

constcmp("he", "hello") // -108
constcmp("hello", "hell") // 111 (ASCII for o)
```

Constexpr Function

- Can be turned into a runtime function
  - Depends on compiler implementation
  - Depends on function argument
    - Becomes a runtime function when given a runtime argument

```
template<int X>           // template argument only accepts
void print_const() {      // compile-time constant values
    cout << X << endl;
};

print_const<constcmp("hello", argv[0])>();
error: ‘argv’ is not a constant expression

// runtime code generated because argv[0] is not constexpr
cout << constcmp("hello", argv[0]) << endl; // 58
```

Compile-Time Function

- Useful for pre-calculating values
  - E.g. crc64 hash of constant strings
- Can be used in conjunction with templates
- Referentially transparent
  - Does not have side effects
- Haskell does this *a lot*
  - Compile-time evaluation
  - The entire program may be optimized down to constants

Constexpr Array

```
constexpr int sum_recursive(const int a[], unsigned n) {
    return (n == 0) ? 0 : a[0] + sum_recursive(a+1, n-1);
}

template<int N>
constexpr int sum(const int (&a)[N]) {
    return sum_recursive(a, N);
}

template<int N>
constexpr char midchar(const char (&s)[N]) {
    return s[(N-1)/2];  // N includes null character
}

constexpr auto c = midchar("goodbye"); // c = 'd'
constexpr auto i = sum({1, 2, 3, 4});  // i = 10
```

Constexpr Class

- Its instances can be compile-time objects
  - Same restrictions apply to methods, but can use members

```
class Rectangle {
    int _h, _w;
public:
    // a constexpr constructor
    constexpr Rectangle (int h, int w) : _h(h), _w(w) {}
    constexpr int area () { return _h * _w; }
};

constexpr Rectangle rekt(10, 20); // compile-time
print_const<rekt.area()>();       // 200

Rectangle rect(5, argc);          // runtime Rectangle
cout << rect.area() << endl;      // 5 (if argc == 1)
```