Class 21: More on Functions: Macros, Lazy evaluation, Built-ins, and Operators

SI 413 - Programming Languages and Implementation

Dr. Daniel S. Roche

United States Naval Academy

Fall 2011

Homework Review

```
new f := lambda a {
  new g := lambda b { ret := b + b/2; };
  new h := lambda c {
    new x := a*c;
    ret := lambda d { ret := g(d) < x; };
  };
  ret := h;
};
new foo := f(3)(4);
write foo(8);
```

- Draw the frames and closures, then show how GC by reference counting and GC by mark-and-sweep would work.

Different kinds of functions

The code `f(5)` here is definitely a function call:

```
int f(int x) { return x + 6; }

int main() {
  cout << f(5) << endl;
  return 0;
}
```

Different kinds of functions

The code `f(5)` here is definitely a function call:

```
int f(int x) { return x + 6; }

int main() {
  cout << f(5) << endl;
  return 0;
}
```

- *What else is a function call?*

Operators

Say we have the following C++ code:

```
int mod (int a, int b) {
  return a - (a/b)*b;
}
```

What is the difference between

`23 % 5`

and

`mod(23, 5)`

Are Operators Functions?

It's language dependent!

- **Scheme**: Every operator is clearly just like any other function. Yes, they can be re-defined at will.
- **C/C++**: Operators are functions, but they have a *special syntax*. The call `x + y` is *syntactic sugar* for either **operator+**`(x, y)` or `x.`**operator+**`(y)`.
- **Java**: Can't redefine operators; they only exist for some built-in types. So are they still function calls?

Built-ins

A *built-in function* looks like a normal function call, but instead makes something special happen in the compiler/interpreter.

- Usually system calls are this way.
  C/C++ are an important exception!
- What is the difference between a built-in and a library function?

Built-ins

A *built-in function* looks like a normal function call, but instead makes something special happen in the compiler/interpreter.

- Usually system calls are this way.
  C/C++ are an important exception!
- What is the difference between a built-in and a library function?
  Library functions are still *written in the language*.

Macros

Recall that C/C++ has a *preprocessor* stage that occurs before compilation.
These are the commands like **#include**, **#ifndef**, etc.

#define defines a *macro*. It corresponds to textual substitution *before* compilation.

Constant Macros

Here's an example of a basic macro that you might see somewhere:

The program

```
#define PI 3.14159

double circum (double radius)
{ return 2*PI*radius; }
```

gets directly translated by the preprocessor to

```
double circum (double radius)
{ return 2*3.14159*radius; }
```

before compilation!

Macro Issues #1

What if we wrote the last example differently:

```
#define PI 3.14159
#define TWOPI PI + PI

double circum (double radius)
{ return TWOPI*radius; }
```

Macro Issues #1

What if we wrote the last example differently:

```
#define PI 3.14159
#define TWOPI PI + PI

double circum (double radius)
{ return TWOPI*radius; }
```

```
double circum (double radius)
{ return 3.14159 + 3.14159*radius; }
```

Probably not what you wanted!

Function-like Macros

We can also do things like this in C++:

```
#define CIRCUM (radius) 2*3.14159*radius

...
cout << CIRCUM(1.5) + CIRCUM(2.5) << endl;
...
```

gets translated to

```
...
cout << 2*3.14159*1.5 + 2*3.14159*2.5 << endl;
...
```

(still *prior to compilation*)

Macro Issues #2

What if we made the following function to print out the larger number:

```
#define PRINTMAX (a,b) \
  if (a >= b) {cout << a << endl;} \
  else {cout << b << endl;}
```

This will work fine for `PRINTMAX(5,10)`,
but what happens with the following:

```
int x = 5;
PRINTMAX(++x, 2)
```

Macro Issues #2

What if we made the following function to print out the larger number:

```
#define PRINTMAX (a,b) \
  if (a >= b) {cout << a << endl;} \
  else {cout << b << endl;}
```

This will work fine for `PRINTMAX(5,10)`,
but what happens with the following:

```
int x = 5;
PRINTMAX(++x, 2)
```

Prints 7!

Thoughts on Macros

- The advantage is SPEED - pre-compilation!
- Notice: no types, syntactic checks, etc.
  — *lots of potential for nastiness!*
- The literal text of the arguments is pasted into the function wherever the parameters appear.
  This is called *call by name*.
- The **inline** keyword in C++ is a compiler suggestion that may offer a compromise.
- Scheme has a very sophisticated macro definition mechanism
  — allows one to define "special forms" like **cond**.

Argument evaluation

Question: When are function arguments evaluated?

So far we have seen two options:

- **Applicative order**: Arguments are evaluated *just before the function body is executed*. This is what we get in C, C++, Java, and even SPL.
- **Call by name**: Arguments are evaluated *every time they are used*. (If they aren't used, they aren't evaluated!)

Lazy Evaluation

(Sometimes called *normal order evaluation*)

Combines the best of both worlds:

- Arguments are not evaluated *until they are used*.
- Arguments are only evaluated *at most once*.

(Related idea to *memoization*.)

Lazy Examples

Note: lazy evaluation is great for functional languages (why?).

- Haskell uses lazy evaluation for *everything*, by default.
  Allows wonderful things like infinite arrays!

- Scheme lets us do it manually with *delayed evaluation*,
  using she *built-in special forms* `delay` and `force`.

Class outcomes

You should know:

- How operators compare with normal functions
- How built-ins compare with normal functions
- What macros are, why we might want to use them, and what dangers they bring.
- The difference between the three argument evaluation options: applicative order, call by name, and lazy evalutation

You should be able to:

- Perform simple macro translations of programs
- Trace program execution using any of the three argument evaluations schemes above