# Programming Languages: Recitation 2

Goktug Saatcioglu

02.07.2019

## 1 Scoping

- Static scoping: Q: What is it? A: Determined at compile time by having the compiler look at the program structue (i.e. AST, sub-trees, traverse the tree).
- Dynamic scoping: Q: What is it? A: Determined by looking at the most recent binding of a variable at run-time (i.e we need to traverse the stack everytime a variable is accessed).
- Certain rules are left to the language implementation e.g.

```
val x = 0;
{
  statement1;
  {
    val x = 1;
    statement2;
  }
  val x = 1;
}
```

  when I refer to `x` in `statement1` where am I referring to? Answers:

  - (Java-esque): $x = 0$
  - JavaScript: Bindings are valid for the whole block, thus what is $x$ in `statement1`? A: undefined.
  - Scala: Bindings are valid for the whole block, thus we get a compile-time error (we are making a forward declaration).

- Mutually recursive definitions (go over if time is left). e.g.

```
struct Manager {
  struct* Employee;
}
struct Employee {
  struct* Manager
}
```

  This is ok for Java but not ok for C++. Q: What is the fix? A: Declare one of them at the very top, e.g. `struct Employee;`.

- Consider the following example:

```
def f() = {
  println(x);
}
def g() = {
  var x = 1;
  f();
}
g()
```

Q: What happens with static scoping? A: Compile-time error. Q: What happens with dynamic scoping? A: Prints out 1.

- Consider the following example:

```
var x = 1;
def f() = {
  println(x);
}
def g() = {
  var x = -1;
}
def h() = {
  g();
  f();
}
def i() = {
  var x = 0;
  h();
  println(x);
}
```

Note: Show the solution by either drawing the tree representation of the program or the stack frames depending on whether static/dynamic. "This is the best way to approach these problems until it becomes easy." Q: What happens with static scoping? A: Prints 1 and 0. Q: What happens with dynamic scoping? A: Prints 0 and 0.

- Dynamic scoping always traverses the stack and this is inefficient (and it might not have smthn, i.e. run-time error). Static scoping compile time, lookups can be determined making it much more efficient and easier to reason about.

- Good example for dynamic scoping.

```
def f() = {
  println(x)
}
def g(y: Int) = {
  if (y > 0)
    f()
  else {
    x = 1
    f()
  }
}
```

- No modern languages have dynamic scoping implemented: it is a bad idea.
- Frame pointer, stack pointer, offsets to determine the placeholders to access variables for static scoping.

# 2 Memory Management

- The "classical" implementation is to use chunks/blocks of memory and ask the os for the memory, i.e. malloc() → ask os → os has a free list (linked list) of memory blocks → os finds a free block (best fit, first fit is easiest) → os allows the use of memory → deallocate → OS adds block to the end of the list. This leads to fragmentation. External fragementation: you have in principle enough space but it is not contigious so you cannot satisfy the request. Internal fragmentation: You request 9 bytes but the smallest block is 16 bytes, so it gives you 16 bytes but now 7 bytes are wasted.
- The buddy system. Split blocks into two until we find the smallest largest available block. When deallocating each block has a buddy and if the buddy is also free then we combine to get back larger blocks. Use a binary tree to accompish the task: top is the whole block, lower levels are smaller splits. Q: Do we still have fragmentation? A: Yes for internal, minimize external though.
- C++ manual memory management and garbage collection.
- Example below. Assume you have 64 bytes of total memory.

```
typdef struct node {  // 20 bytes
  int val;            // 4 bytes
  Node* left;         // 8 bytes
  Node* right;        // 8 bytes
}
node* root = malloc(sizeof(node));   // 24 bytes
                                     // 4 bytes internal fragmentation
root->val = 1;
root->left = null;
root->right = null;
node* child = malloc(sizeof(node));  // 24 bytes (48 bytes total used)
                                     // 8 bytes internal fragmentation
child->val = 0;
child->left = null;
child->right = null;
root->left = child;
free(child);                         // freed 24 bytes (24 bytes total used)
                                     // possibility of external fragmentation
                                     // Q: can I request 40 bytes? A: depends
// root->left->val = -1;             // What's wrong with this call?
```

Draw the memory layout representation for this program. (Solution: Struct contents are stored contigiously in memory but multiple structs may begin at different starting addresses.)