Paul Graham: "Beating the Averages"

Recap of denotational semantics

Announcements

- **Homework posted**
- **TA's office hours posted**
- **Should have started learning Haskell by now**

Binding

- **Find all the identifiers in the following program (see slide)**
- **When is the binding made? Some examples:**
  - Language definition time: "int" in Java
    - 32 bits (from –2^31 to +2^31 – 1)
  - Language implementation time: "int" in C
    - at least 16 bits (from –2^16 to +2^16 – 1)
  - Compile-time: class definitions in Java
  - Run-time: variable values
    - Values of numbers in Fortran (can assign 3 to be 4)
    - Functions and class definitions in Common Lisp (can redefine at run-time)
- **Sometimes we talk of static and dynamic binding**
  - Dynamic: run-time
  - Static: before run-time
- **Static scope of variables**

```
int main()
{
        int x, y;
        scanf("%d %d", &x, &y);
        {
                int temp;
                temp = x;
                x = y;
                y = temp;
        }
        printf("%d %d", x, y);
        return 0;
}
```

  - Could we have added "temp" to the printf statement? No!
  - Reason: The scope of temp is limited to the inner block
    - What does that mean?
    - Inside the inner block, the name "temp" is bound to a certain variable (i.e., a certain memory position).
    - Outside the block, there is no such binding: "temp" does not refer to anything there.
  - In the C program above, can we find the binding at compile-time?
    - Yes, keep "nesting" outward from where it is used until you find the block in which it is declared.

    - This is called static scope.
  - Another example (C, but Java would be very similar)

```
int foo(int a)
{
      int x;
      x = a;
      return x * x;
}

int bar (int a)
{
      int x;
      return x * x;
}

int main()
{
      return foo(3) * bar(4);
}
```

  - How many variables are in the program?
  - What does the program return?
    - Calls foo, which returns x * x = 9
    - Calls bar, which returns x * x
      - Which variable is x?
      - The scope of x in foo is limited to foo
      - The scope of x in bar is limited to bar
      - Same name, but because of scope rules, they are two different variables.
      - The value of x in bar is "indeterminate"—it could be anything!
- **Dynamic scope of variables**
  - Consider the following program:

```
class Dynamic {
    static int addb(int a) {
        return a + b;
    }

    static int intermediate(int x) {
        return addb(x);
    }

    static void caller1() {
        int b = 3;
        System.out.println(intermediate(7));
    }

    static void caller2() {
        int b = 2;
```

```
        System.out.println(intermediate(2));
    }

    public static void main(String args[]) {
        caller1();
        caller2();
    }
}
```

- What does the program print?
- Is it even a valid Java program?
  - addb seems to be using an undeclared variable
- This is a made-up variant of Java with dynamic scope
  - The variable b refers to is determined at run time
  - Instead of just looking at the static structure of the program, we look at its dynamic structure
  - At the time addb is executing, look at its callers to find a variable called b
- Prints 10 and 4
- Languages with dynamic scope: Emacs Lisp, TECO, APL, SNOBOL4
- Common Lisp has both static and dynamic scope
- Dynamic scope useful for writing extensible systems such as Emacs. Example from Stallman:
  - The standard system has a function B
  - The extension consists of functions A and C, and the variable FOO
  - A binds FOO and calls B, which calls C
    - C is an editor operation, e.g., indenting the line
    - FOO could be the number of characters to indent blocks of source code
    - C needs access to FOO
  - Alternative: Passing FOO as a formal argument to B and C
    - B was written before C, and expected to call some other function (e.g., insert a tab) that does not need an argument like foo
    - Need to change all editor commands to accept an extra argument, just in case an extension needs it.
    - Major pain.
  - Alternative: Global variables
    - Global variable, FOO
    - Instead of binding FOO, A fakes it by assigning to the global variable and saving the previous value somewhere so it can be restored.
    - Pollutes global namespace and interferes with multithreading
- But only using dynamic scope makes programs hard to debug
  - People are much better at reasoning about the static structure of the program (which, incidentally, corresponds to the indentation structure of the source code) than at reasoning about the call graph at run time
- Some languages, such as Common Lisp, have both static and dynamic scope
  - Static scope is the default, and is used for most purposes

- Dynamic scope ("special variables") available for the special cases where they are useful.

Passing arguments to functions

- **Call by reference (call by sharing)**
  - Example

```
void foo(int b)
{
        b = 2;
}

int main()
{
        int a = 1;
        foo(a);
        return a;
}
```

  - With call by reference, what does the program return? 2
  - The b in foo is the same variable as the a in main
  - What if foo is called with an expression?

```
int main()
{
        int a = 1;
        foo(a + 1);
        return a;
}
```

  - Two things can happen:
    - a can remain unchanged (1)
    - error
  - Call by reference is standard in Fortran
- **Call by copy**
  - Formal parameter is a separate local variable in caller
  - Call by value (copy-in)
    - Value of actual parameter copied to the formal parameter
    - Data flow from caller to callee, but not the other way around
  - Call by result (copy-out)
    - At return, value of formal parameter copied to actual parameter
    - Data flow from callee to caller, but not the other way around
  - Call by value-result (copy-in/copy-out)
    - Value of actual parameter copied to the formal parameter
    - At return, value of formal parameter copied to actual parameter
    - Data flow both ways
- **Call by name**
  - Textual substitution
    - Example:

- What does the program do?

```
temp = i;
i = a[i];
a[i] = temp;
```

- The second a[i] is different form the first a[i]. Probably not what was intended!

- The formal parameter denotes a location in the environment of the caller
  - Example:

```
int c; /* global variable */

swap(int a, b) {
        int temp;
        temp = a;
        a = b;
        b = temp;
        c++;
}

main() {
        int c, d;
        …
        swap(c, d);
        …
}
```

  - What does the program do?

```
int c;
{
        int c, d;
        int temp;
        temp = c;
        c = d;
        d = temp;
        c++;
}
```

  - Ends up incrementing the c that is local to main, not the global c.

- **Parameter passing conventions in different languages**
  - Call-by-name standard in ALGOL 60 (but call-by-value also available)
  - Simula 67 provides call by value, call by reference, and call by name.
  - C++, Pascal, Modula-2 provides call by value and call by reference.
  - Ada has all three versions of call-by-copy (in, out, in out)
  - C uses call-by-value, but pointers make it possible to fake call-by-reference.
  - Lisp uses call-by-value.