Name Binding

- **Binding**: an association between something in a program and a property
  - **Static binding**: the association occurs *before* run-time
  - **Dynamic binding**: the association occurs *during* run-time
- **Name binding**: associating an entity with a name
- A language in which bindings happen as *early* as possible tends to allow for more efficient code
  - A *compiled* language, in which the compiler can bind names and optimize memory layout, can often run much faster
- Late bindings provide *flexibility*
  - An *interpreted* language may not make any binding decisions until run-time, which can be powerful, but also have significant time-costs

Scope

- **Name scope**: all the statements that can access a name, and use/affect its bound value
- **Static scoping**: at compile time, based on program text
  - Also known as *lexical* scoping
- **Dynamic scoping**: at run time, based on state of program
  - This is much rarer, but does exist in e.g. Perl, Common Lisp

Static Scoping

- A *symbol table* is a data structure used by a compiler that allows it to keep track of each declared name and its bindings
- A *dictionary* is a data structure that associates names and declarations of type, value, etc.

Algorithm for determining variable bindings:
1. When a scope is entered, push empty dictionary onto a stack
2. When a scope is exited, pop the stack
3. When a declaration occurs: bind name to declaration, storing that binding in the dictionary at top of the stack
4. When a variable reference occurs: search the top-most dictionary for the variable and it’s declaration.
   - If not found, repeat going down the stack from top to bottom
   - If not found in the bottom-most dictionary: “unresolved symbol”.

Friday, 1 Mar. 2019

Programming Languages (CS 421/521)

M. Allen, 01 Mar. 19
Static Scoping

- In static scoping, the symbol tables for each code block are constructed once, at compile time.
- The referencing environment for each name is thus made available before the program executes, using the stack-of-dictionaries approach.

```
1. int i, j;
2. void fun(int x) {
3.    int i = 1;
4.    int y = 2;
5.    j = i + x + y;
6. }
7. void main() {
8.    int x = 5;
9.    int y = 3;
10.   fun(x + y);
11. }
```

Note that the global scope here contains not just the global variables (i, j), but also the method/function names (fun, main).
- This assumes we are using a language like Java, where methods can call one another freely, no matter the order in which they are defined.

```
Global: <i, 1>, <j, 1>, <fun, 2>, <main, 7>
```

Static Scoping

- Here, each block has its own associated dictionary and referencing environment.
  - We represent these as bindings between the identifiers and the lines where they are declared/defined.

```
Global: <i, 1>, <j, 1>, <fun, 2>, <main, 7>
fun: <x, 2>, <i, 3>, <y, 4>
main: <x, 8>, <y, 9>
```

Static analysis of scope nesting and dictionary stacks gives us bindings.
- For example, `main()` occurs in the context of the dictionary stack:

```
main: <x, 8>, <y, 9>
Global: <i, 1>, <j, 1>, <fun, 2>, <main, 7>
```

This means that the symbol table for `main()` is simply the union:

```
main: <x, 8>, <y, 9>, <i, 1>, <j, 1>, <fun, 2>, <main, 7>
```
Static Scoping

On the other hand, `fun()` occurs in the context:

```plaintext
fun: <x, 2>, <i, 3>, <y, 4>
Global: <i, 1>, <j, 1>, <fun, 2>, <main, 7>
```

This means that the symbol table for `fun()` includes some local name bindings:

```plaintext
fun: <x, 2>, <i, 3>, <y, 4>, <j, 1>, <fun, 2>, <main, 7>
```

Dynamic Scope

Thought by many to have been a mistake, and relatively rare

- Common LISP uses dynamic scope
- Perl allows variables to be declared to have dynamic scope

Determined by the calling sequence of program units, not the static layout of the program

- Name bound to most recent declaration given execution history
- A dictionary stack is constructed as the program executes and maintained as long as a containing block of code is in operation

Scoping Stack ≠ Runtime Stack!

When `fun()` is actually called inside of `main()`, the runtime stack will keep track of all methods in execution (below)

This is important, but does not affect the dictionary stack for scoping (above), which is determined at compile time

```plaintext
fun running...
main running...
Global running...
```

Even though `fun()` is called inside of `main()`, the dictionary for its bindings does not inherit bindings `<x, 8>` and `<y, 9>`

Again, we assume a language like Java, where the scopes of `fun()` and `main()` are disjoint, or non-nested
Example: Dynamic Scoping

- The basic block dictionaries are built exactly the same way as before.
- Now, however, we build the symbol tables at run-time.

Symbol tables now depend upon how the calling stack appears as required at run-time.

For example, the stack for \( f_2() \), when it is called at line 13, inside \( \text{main}() \), looks like:

This means that the symbol table is simply the same as the Global one.

These two different tables lead to different effects.

The program, which in a static context always terminates with \( i = 4, j = 4 \)
now terminates with \( i = 4, j = 3 \).
Combining Scoping Techniques: Perl

- In Perl, a subroutine can declare the scope of variables to use either of the two scoping approaches:
  1. **Static**: using the `my` keyword
     - Such a variable is local to the subroutine itself, and is not visible to other functions called from within it.
  2. **Dynamic**: using the `local` keyword
     - Confusingly, such a variable can act non-locally, as it is visible to any other functions that called from within the subroutine.

A Perl Example

```
my $x = "Outer";
sub inside {
    print "inside: $x";
}
sub staticVar {
    my $x = "Static";
    print "static: $x";
    &inside
}
sub dynamicVar {
    local $x = "Dynamic";
    print "dynamic: $x";
    &inside
}
&inside &staticVar &dynamicVar
```

Exercise: Name Scoping

- Given this code, give the dictionary stack (line-address entries) for:
  1. Each variable at line 8, if $B()$ is called from line 14, using **static** scope
  2. Each variable at line 8, if $B()$ is called from line 14, using **dynamic** scope
  3. Each variable at line 8, if $B()$ is called from line 5, using **dynamic** scope

Next Week

- **Topic**: Closures, bindings, semantics
- **Reading**: Text, 3.1–3.3, 3.6, 4.1–4.3
- **Homework 02**: Friday, 08 March, 5:00 PM on D2L
- **Midterm Exam**: Friday, 15 March, in class
  - Open book, open notes
  - Practice exam available by Friday, 08 March
- **Office Hours**: Wing 210
  - Monday: 9:00 AM – 10:30 AM
  - Tuesday: 3:00 PM – 4:00 PM
  - Wednesday: 9:00 AM – 10:30 AM
  - No office hours Thursday
  - Friday: 9:00 AM – 10:30 AM