Structure of Programming Languages – Lecture 7

CS 636 – 536

May 18, 2009

1. Names and Binding
    - Naming Things
    - Names, Scope, and Objects

2. Binding Names to Objects
    - Dynamic Name Binding

Good names / Bad names

What does it do?

```
typedef struct X {char next; struct X* prior;} xType;
typedef xType* myType;

    myType storIt(myType temp, char jxq){
        myType jqx, first;
        first = temp;
        jqx = temp->prior;
        while (jqx != NULL) {
            if (jqx->next == jxq) jqx = NULL;
            else { first = jqx;  jqx = jqx->prior; }
        }
        return first;
    }
```

Good names / Bad names

What does it do?

```
typedef struct CELL {char data; struct CELL* next;} Cell;
typedef Cell* List;

    List storIt(List head, char key){
        List scan, prior;
        prior = head;
        scan = head->next;
        while (scan != NULL) {
            if (scan->data == key) scan = NULL;
            else { prior = scan;  scan = scan->next; }
        }
        return prior;
    }
```

Names and Objects

In a program, names get bound to objects.

- Names of **static variables** are bound to objects at load time.
- **Declared names** are bound to objects when the declaration is executed.
- Reference **parameter names** are bound to arguments when the function is called.
- In some interpreted languages, a name is bound to an object by an assignment operator.

Names $\neq$ Objects

It is not realistic to require every objects to have a different name.

- Inventing new names becomes more and more difficult as programs get large.
- We need to combine modules written by different people.

The correspondence between names and objects is not "nice".

- We often use the same name in multiple scopes.
- In OO languages, functions have many methods with the same name.
- Call by reference: Two names for one object.
- `new Cell()` or `(cons x y)`: an object with no name.
- Parameter in a recursive function: one name with 0..n objects.

Scope

- **local** names are those defined within a particular function or scope.
- **relatively global** names are those defined within a scope that encloses the local scope.
- **global** names are defined in the current module and are visible throughout the module.
- **external** names are defined in a different module and imported by the current module.
- Some languages support **packages**, which are sets of modules that share a common namespace.

Lexical and Dynamic Scoping

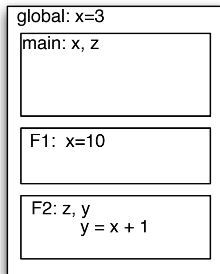
Dynamic scoping is a mistake made in LISP but not in modern languages.

- **Lexical scoping**: a non-local name is interpreted to mean the closet enclosing declaration in the printed program.
- **Dynamic scoping** : A name is interpreted to mean the most recent binding for that name on the stack.

When does it make a difference?

- No difference for a recursive function.
- No difference for references to local names.
- Different when calling a function that is outside the local scope, where the function refers to a name defined in the local scope, and that name is also defined relatively-global to the function being called.

Lexical and Dynamic Scoping



Dynamic Structure

1. main called F1
2. F1 called F2.
3. F2 uses x.

Dynamic scoping:
x means F1's x.

Lexical scooping:
x means global x.

Stack Diagram

F2	y	4 or 11
	z	3
F1	x	10
main	x	17
	z	2
global	x	3

Ambiguous Names

Two-part names have evolved to handle most cases of ambiguous names:

- Let each identifier have two parts: a primary name and a scope or namespace name.
- Identifiers written with only the primary name belong to the current context or scope.
- A scope indicator must be written before the primary name to refer to an external object.
  - Java: `n = Math.random();` (Class.static function name)
  - Java: `super.paintComponent(g);` (Call base class function)
  - C++: `StuData::print(out)` (Call base class function)
  - C++: `::close();` (Call a global function)

Dynamic Name Binding

Interpreted languages generally use dynamic name binding.

- Any name can be bound to any object at any time.
- The type is part of the object, not part of the name.
- Compile-time type checking cannot be done, (run-time checking can be done, but...).

Compiled languages are able to detect errors that dynamically typed languages cannot.

- Errors having to do with consistency.
- Errors due to having two conceptual types mapped onto the same structure.
- Const and final errors.

Static and Dynamic Binding

Language	Symbol Table			Run-time Memory
	name	type	binding	storage object
C and Java	sum	double	→	3.1415928
	count	int	→	17
FORTH	name	code field	parameter field	
	5count	variable	17	
LISP Python	name	binding	type tag / storage object	
	pi	→	rational	22 / 7
	sum	→	number	3.1415928