Lexical vs. Dynamic Scoping

- Lexical: free variables bound at compile time.
- Dynamic: free variables bound at run time.

Output:

```
int y = 3;
int f(int x) {
    return x + y;
}
int g(int y) {
    return f(2);
}
main() { print(g(4)); }
```

```
lexical scoping:
Output: f(2) = 2 * y = 2 * 4 = 8
```

```
dynamic scoping:
Output: f(2) = 2 * y = 2 * 3 = 6
```

Obliq: Overview

- Distributed Lexical Scoping
- Interpreted
- Untyped
- Built on top of Modula3 Network Objects
- Object Oriented
- Garbage Collection

Obliq Objects

- “Prototypical Approach”
- have a state
- are local to a site
- example:

```
c => { x => 3,
        foo => meth(a, b) b end,
        x => alias y of o2 end;
    }
```
Encoding Inheritance

- Inheritance:
  ```java
  Class A{
  int i;
  ...
  }
  Class B extends A{
  int j;
  ...
  }
  ```

Aliasing / References

- `a.x := alias of b.y;`
  - Operations on a.x are redirected to b.y
  - a.x now acts as a local stub
  - Like "remote pointers"
  - Redirect a to b:
    - Alias for all fields of a

Special Field Attributes

- `protected`:
  - Prohibits external updates, cloning and aliasing

- `serialized`:
  - Like synchronized in Java
  - At most one thread operating on a field at a time

Example 1: Serialized Queue (1)

- Standard example for concurrent programming
- Two methods:
  - `read` - reads and removes an item from the queue
  - `write` - puts an item on the queue
- A condition:
  - `nonEmpty` - guard to prevent read from empty queue

Example 1: Serialized Queue (2)

```java
let queue = {
  (let nonEmpty = condition();
  var q = [];
  [ protected, serialized, 
    write =>
      meth(e, elem);
      q := q @ [elem];
      signal(nonEmpty);
      end;
  ];
  let q0 = q[0]
  q := q[1 for #(q)-1]
  q0;
  end;
};
read =>
  meth(e)
  watch nonEmpty
  until #(q)>0
  end;
  let q0 = q[0]
  q :=
  q[1 for #(q)-1]
  q0;
  end;
};
```

Network Objects

- Export (ObjA, "myObj")
- Import "myObj"
- Method Calls
- Site 1
- Site 2
Example 2: Agent Migration (1)

- Idea:
  - write a program (agent)
  - put it into a network
  - disconnect
  - agent travels around in the network and collects information
  - come back online
  - collect agent

Example 2: Agent Migration (2)

- Problem:
  - to collect information, an agent needs some kind of storage

- Obliq Solution:
  - one object can be attached to another
  - active computations transmitted together with an evaluation stack (=> strong mobility)

Example 2: Agent Migration (3)

```
let state = { ... }; let agent = proc(state, arg) ... end;

// migrate using a computation engine
let atSite1 = net_importEngine("E1@Site1")
atSite1{ proc(arg)
  agent(copy(state), arg)
end)
```

Example 2: Agent Migration (4)

- Before Migration:
  - State
  - agent

- After Migration:

```
Network Objects

- Server:
  - net_export("obj", Namer, site1Obj)

- Client:
  - let site1Obj = net_import("obj", Namer);
  - site1Obj.opA(args);
  - site1Obj.opB(site1Obj);
```

Conclusion

- consistent way of dealing with distribution
- at the cost of high traffic volume on network
- Obliq as ‘testing suite’ for network-object libraries
Network Objects (2)

- Execution Engines:
  - Server:
    ```
    net_exportEngine(E1@Site1", Namer, arg);
    ```
  - Client:
    ```
    let atSite1 = 
    net_importEngine(E1@Site1", Namer); 
    atSite1(proc(arg) 3+2 end);
    ```