# Actor Thinking

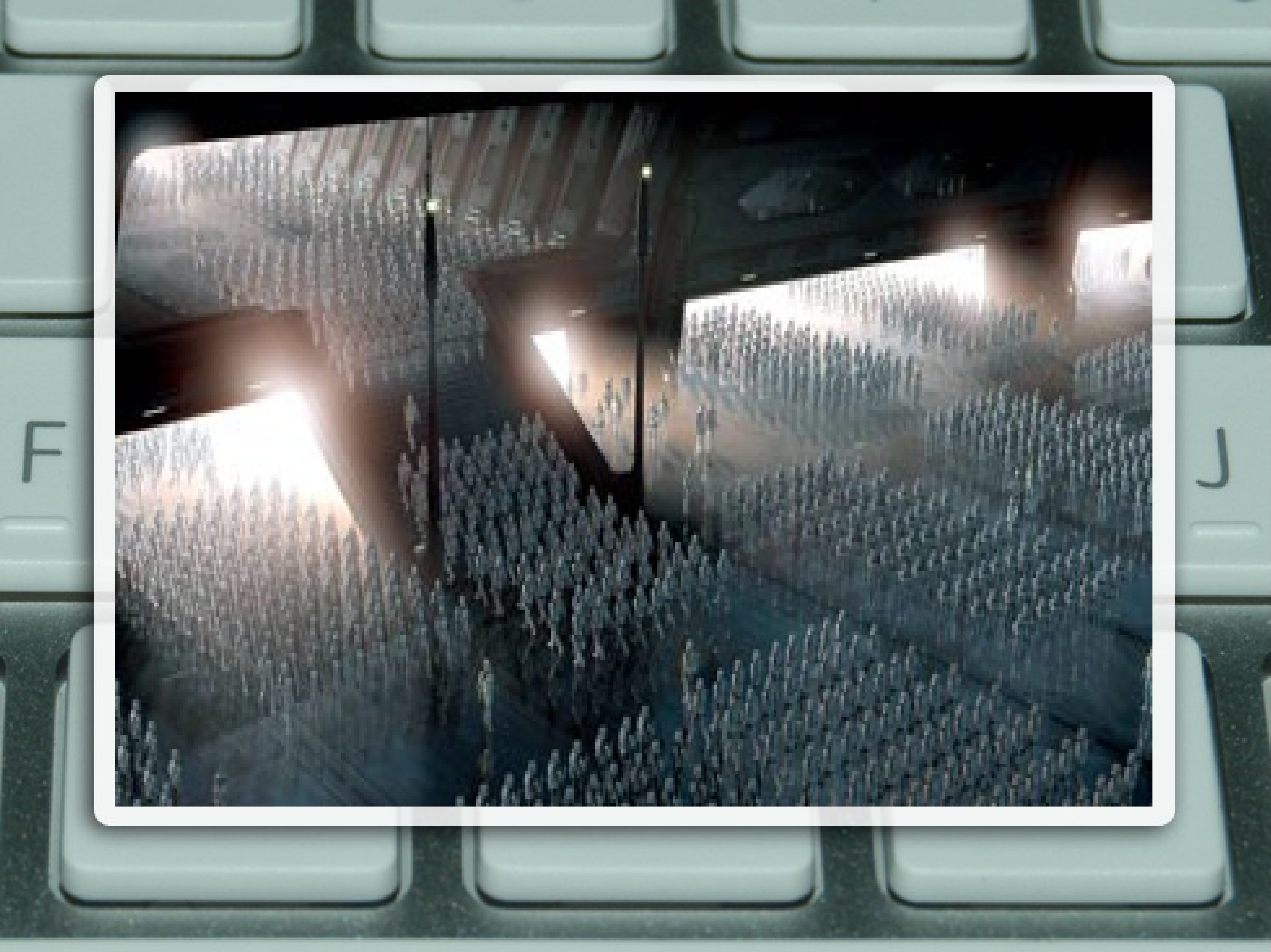
Dale Schumacher
twitter: @dalnefre

QCon/SF 2010-11

# Conway's Law

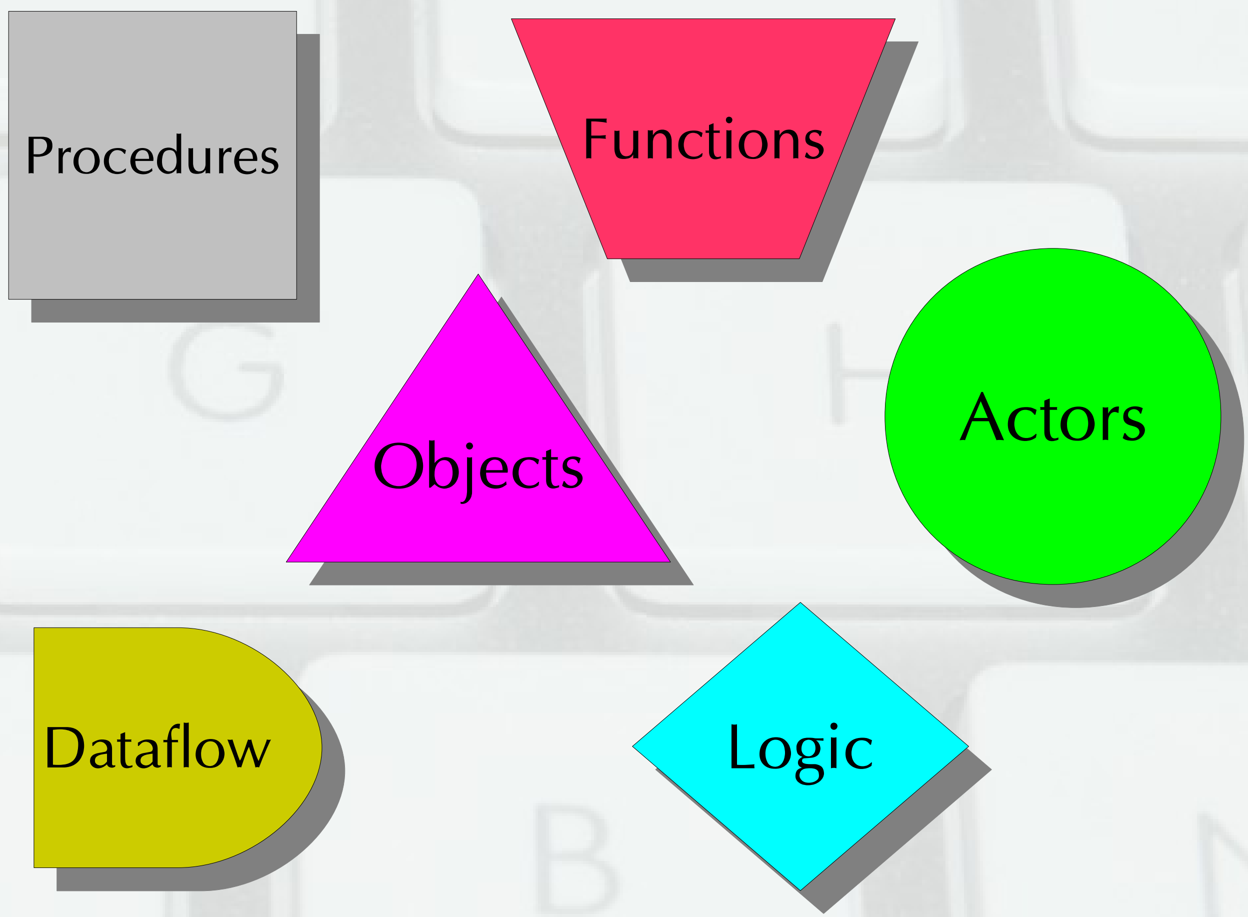
"... organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations."

–M. Conway (1968)

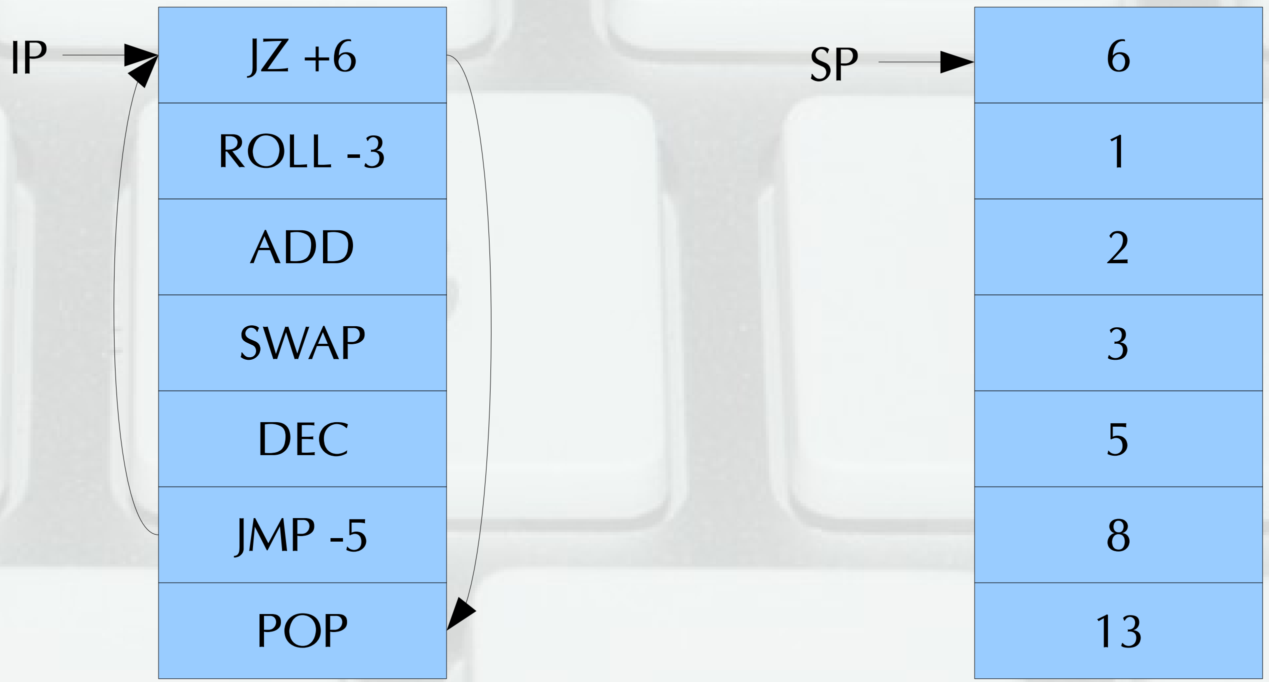

F
J


Copyright © 2001 New Line Cinema Productions, Inc.

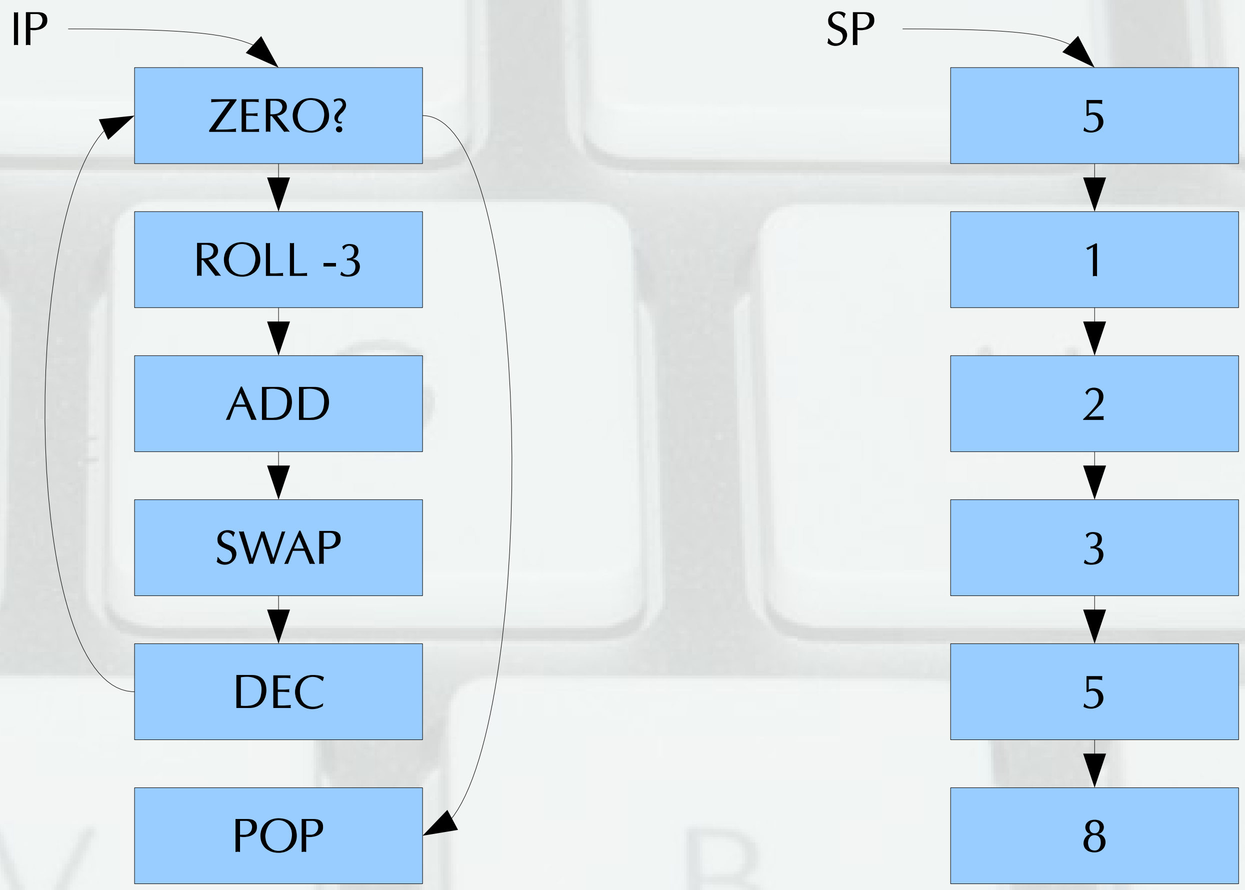
# Models of Computation



# Sequential Stack Machine



# Linked Stack Machine



# Actors and Functions

"Hewitt had noted that the actor model could capture the salient aspects of the lambda calculus; Scheme demonstrated that the lambda calculus captured nearly all salient aspects (excepting only side effects and synchronization) of the actor model."

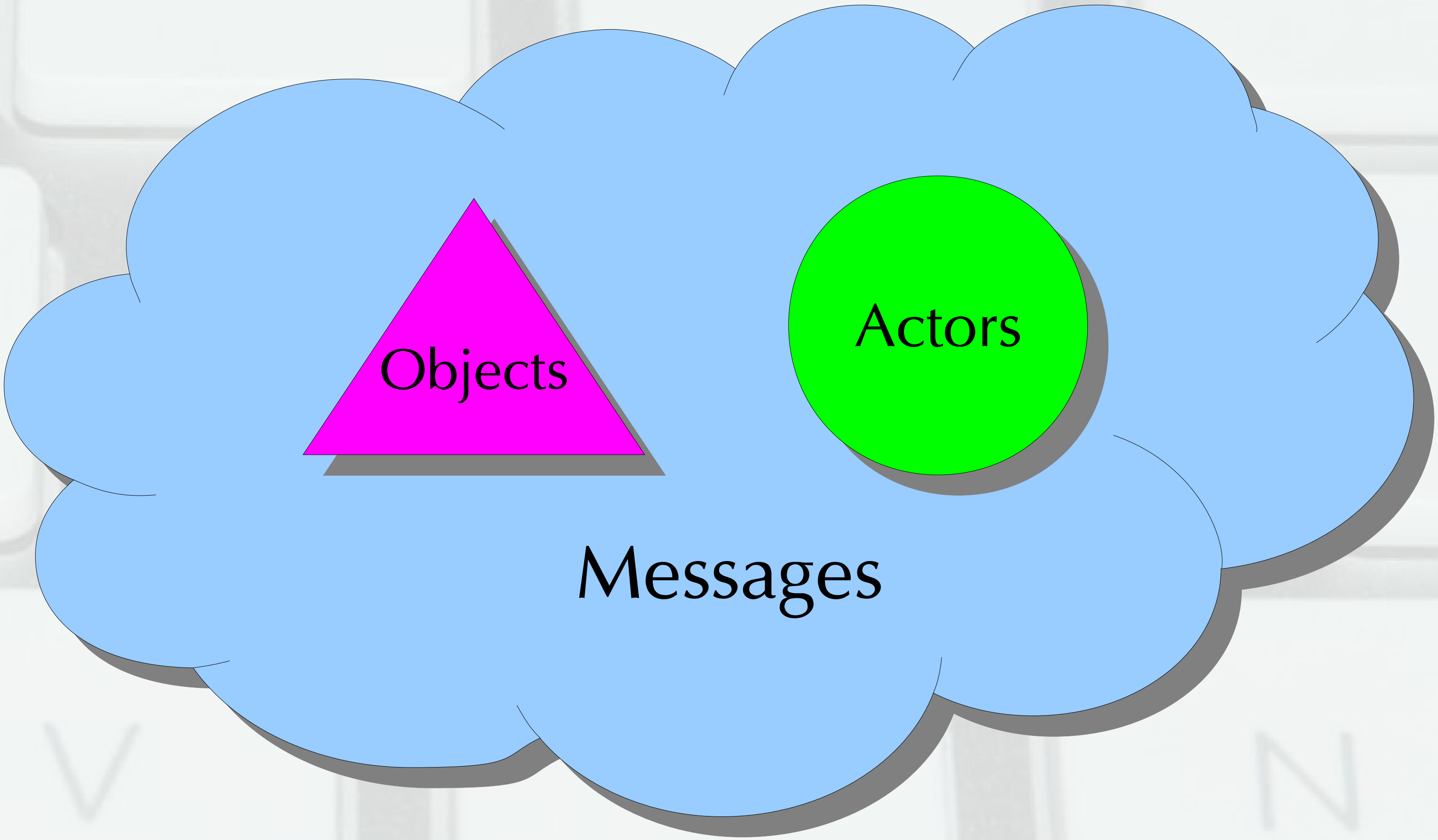
–G. Steele and R. Gabriel (1993)

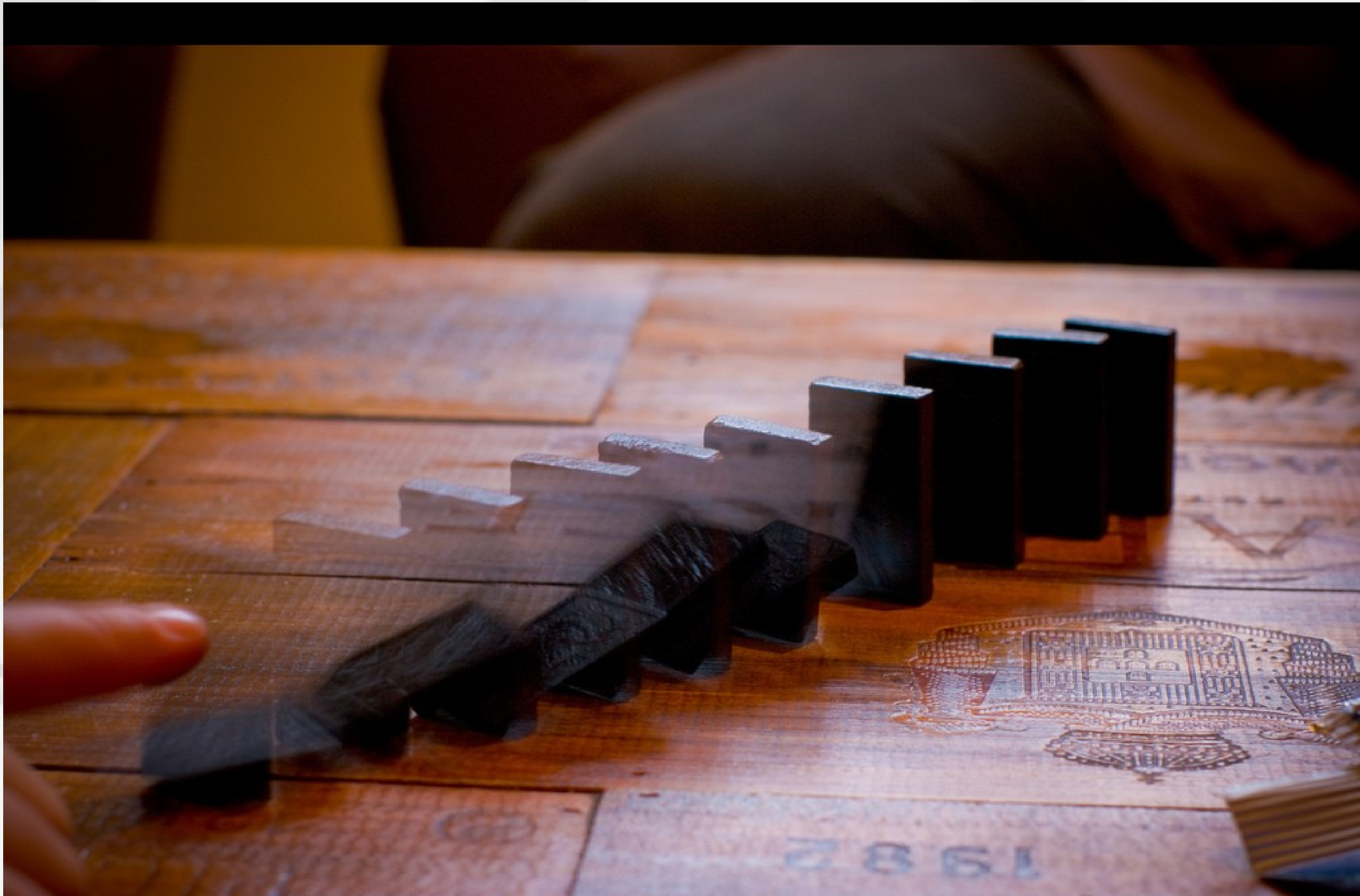
# Objects (Kay) & Actors (Hewitt)

- Everything is an *object*
- Objects communicate by sending and receiving *messages*
- Objects have their *own memory*
- Inheritance? Polymorphism?

- Configuration = *actors* + *messages*
- Actors *respond* to messages by:
  - Sending messages
  - Creating actors
  - Changing behavior
- Everything is *concurrent*

# It's all about the *messages*

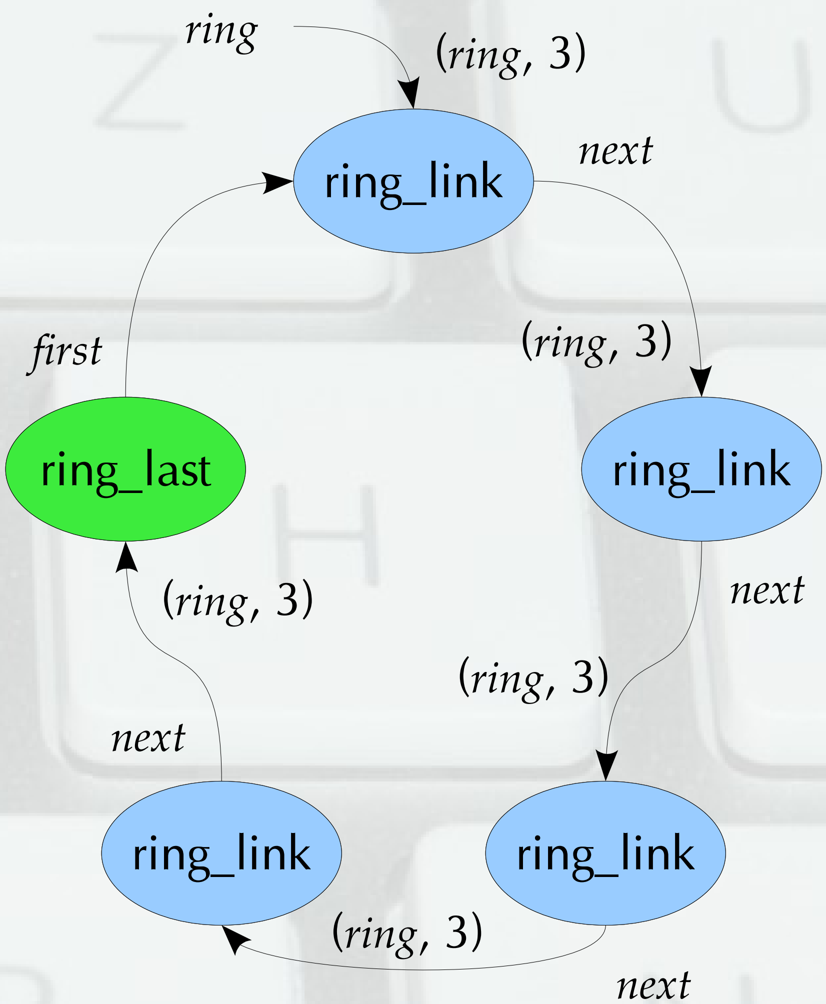




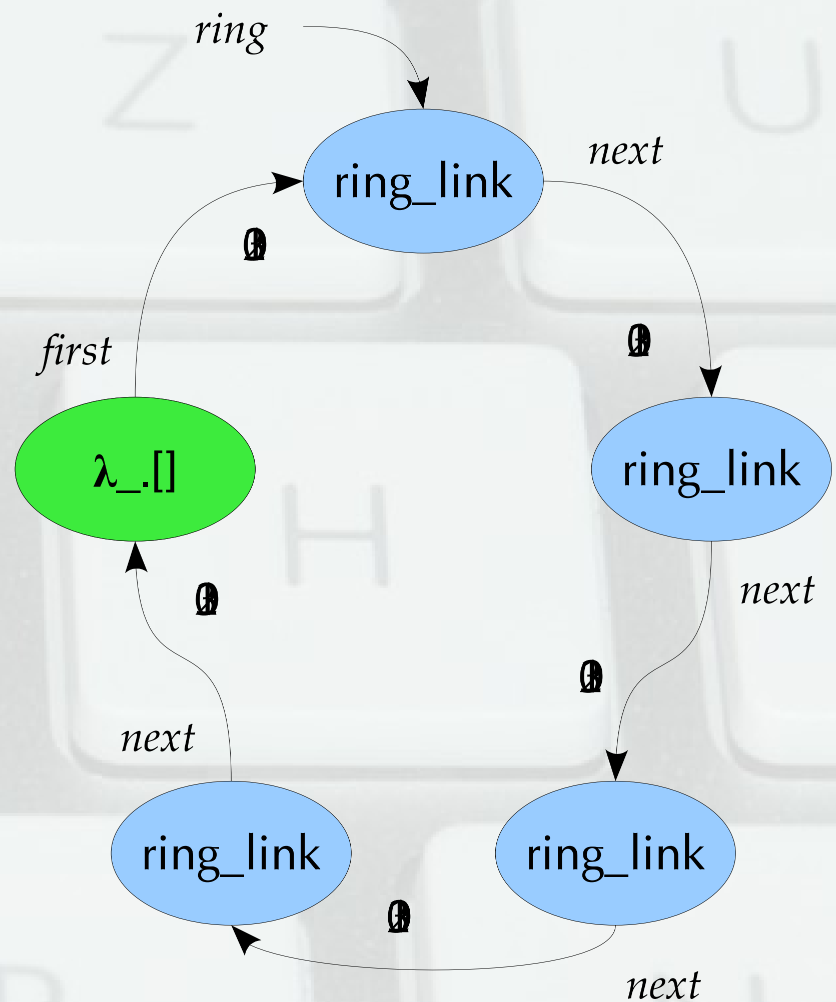
http://www.flickr.com/photos/sunface13/4815937062/


F
J

```
LET ring_builder(n) = λ(first, m).[
    CASE n OF
    0 : [
        BECOME λm.[ # ring_last(first)
            CASE m OF
            0 : [ BECOME λ_.[] ]
            _ : [ SEND dec(m) TO first ]
            END
        ]
        SEND m TO first
    ]
    _ : [
        CREATE next WITH ring_builder(dec(n))
        SEND (first, m) TO next
        BECOME λm.[ # ring_link(next)
            SEND m TO next
        ]
    ]
    END
]
CREATE ring WITH ring_builder(4)
SEND (ring, 3) TO ring
```

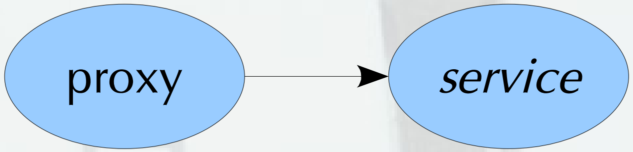


```
LET ring_builder(n) = λ(first, m).[
    CASE n OF
    0 : [
        BECOME λm.[  # ring_last(first)
            CASE m OF
            0 : [ BECOME λ_.[] ]
            _ : [ SEND dec(m) TO first ]
            END
        ]
        SEND m TO first
    ]
    _ : [
        CREATE next WITH ring_builder(dec(n))
        SEND (first, m) TO next
        BECOME λm.[  # ring_link(next)
            SEND m TO next
        ]
    ]
    END
]
CREATE ring WITH ring_builder(4)
SEND (ring, 3) TO ring
```

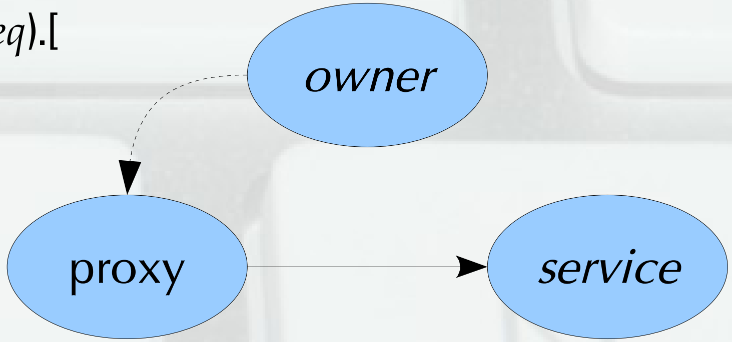


# Object-Capability Security

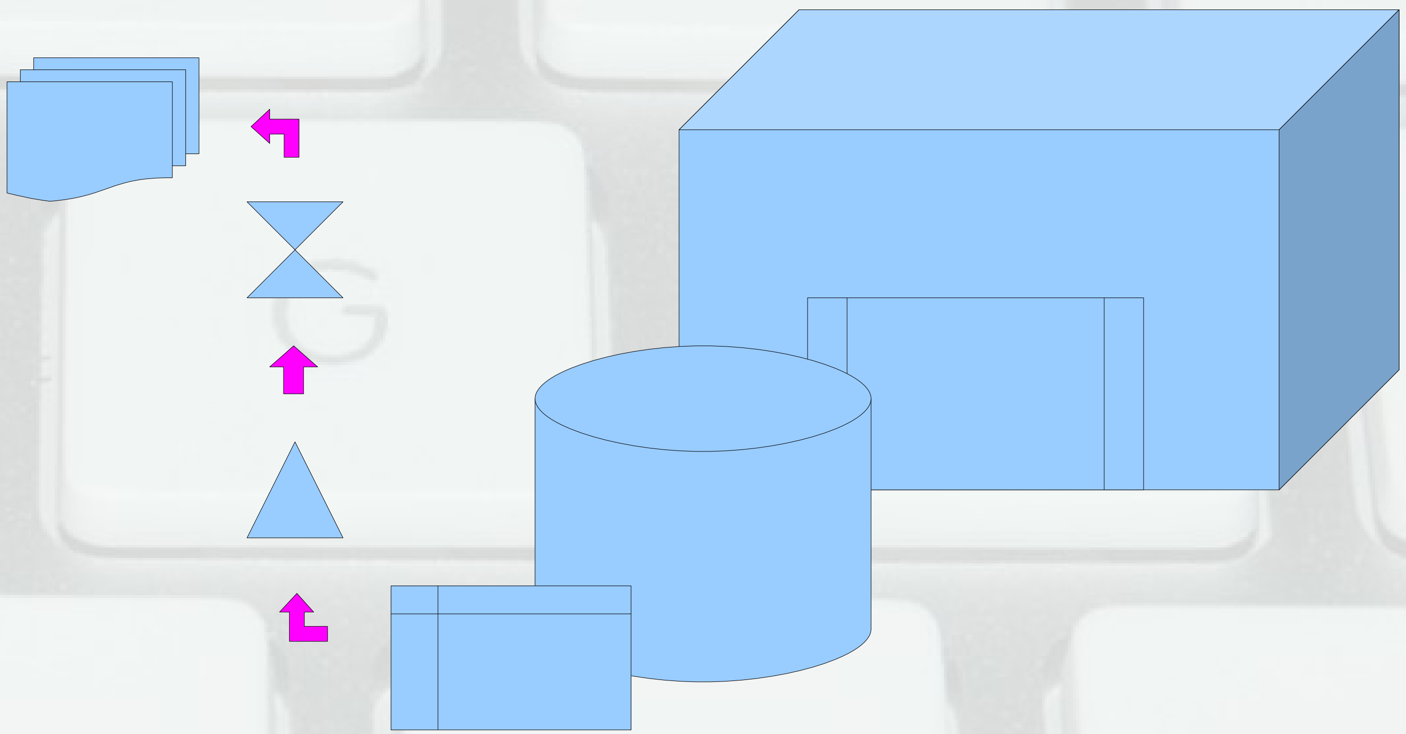
```
# service protocol: (cust, {#create, #read, #update, #delete}, key[, value])

LET read_only_proxy_beh(service) = λ(cust, req).[
	CASE req OF
	(#read, key) : [ SEND (cust, req) TO service ]
	_ : [ SEND ? TO cust ]
	END
]
```



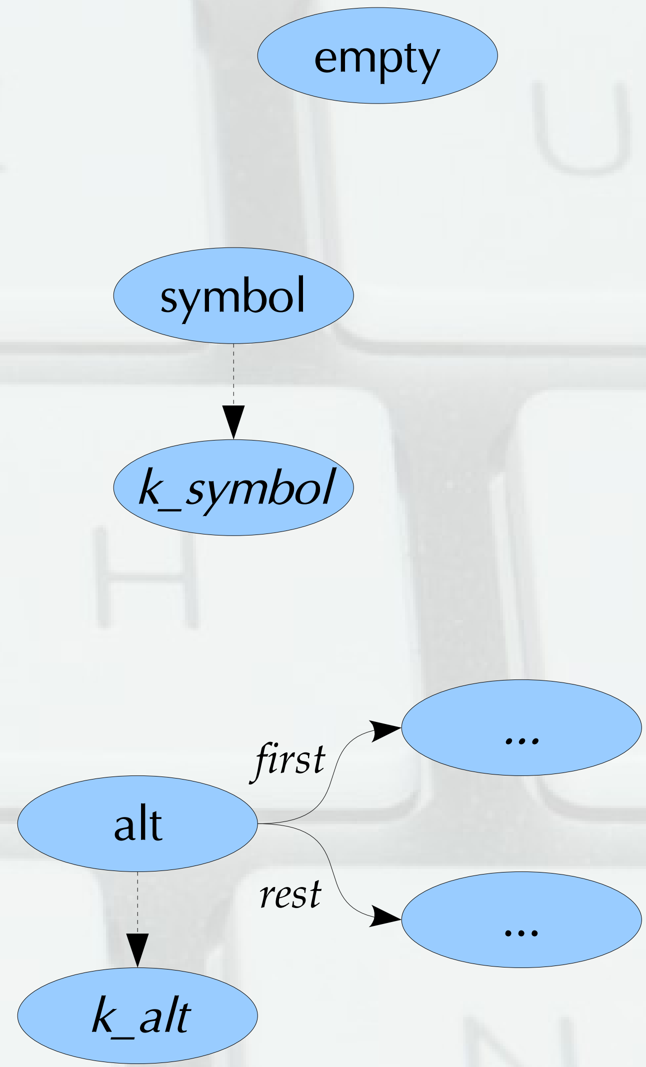

```
LET revocable_delete_proxy_beh(service, owner) = λ(cust, req).[
	CASE req OF
	(#delete, key) : [ SEND (cust, req) TO service ]
	(#revoke, $owner) : [ SEND #revoked TO cust ]
	_ : [ SEND ? TO cust ]
	END
	BECOME λ(cust, _).[ SEND ? TO cust ]
]
```



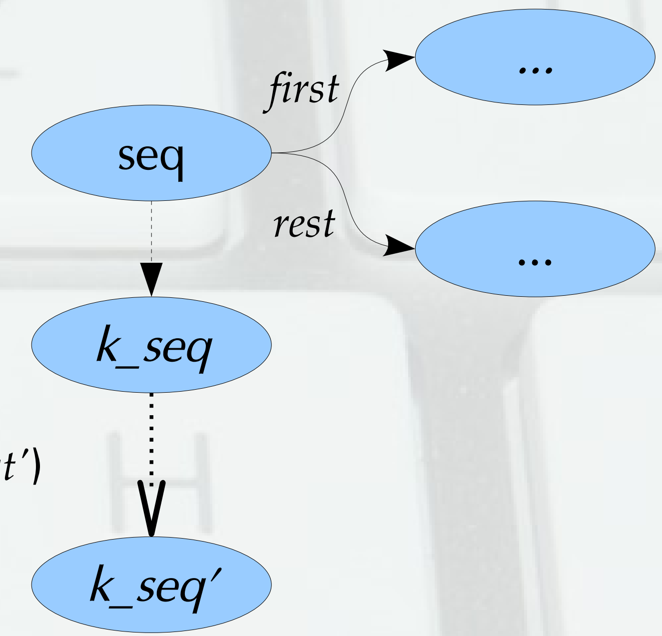
# Lifetimes vary dramatically



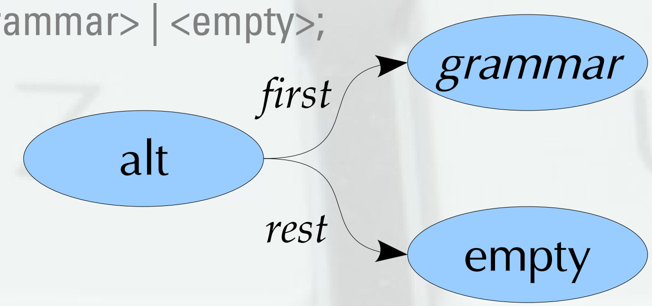
```
CREATE empty_grammar WITH λ(cust, #match, src).[
	SEND (TRUE, NIL, src) TO cust
]
LET symbol_grammar_beh(symbol) = λ(cust, #match, src).[
	SEND (k_symbol, #read) TO src
	CREATE k_symbol WITH λ(token, next).[
		CASE token OF
		$symbol : [ SEND (TRUE, token, next) TO cust ]
		_ : [ SEND (FALSE, src) TO cust ]
		END
	]
]
LET alt_grammar_beh(first, rest) = λ(cust, #match, src).[
	SEND (k_alt, #match, src) TO first
	CREATE k_alt WITH λmatch.[
		CASE match OF
		(TRUE, _) : [ SEND match TO cust ]
		_ : [ SEND (cust, #match, src) TO rest ]
		END
	]
]
```

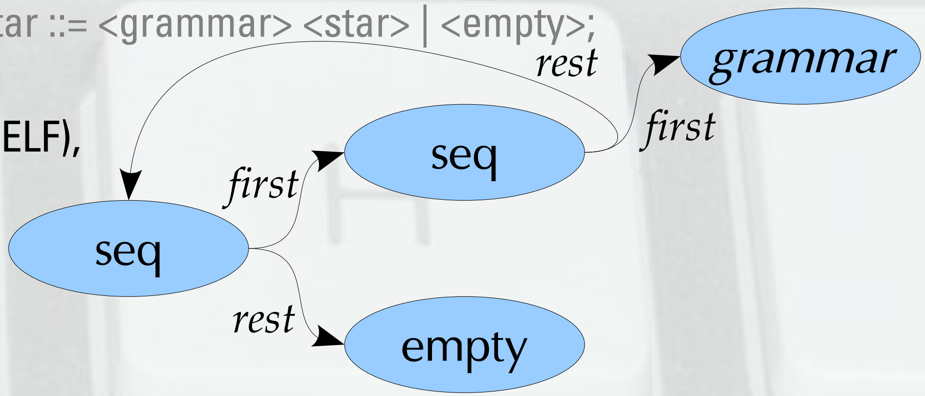


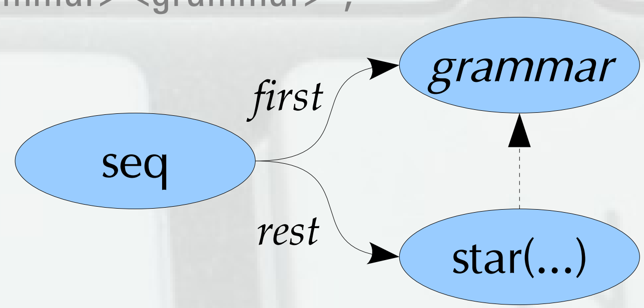
```
LET seq_grammar_beh(first, rest) = λ(cust, #match, src).[
    SEND (k_seq, #match, src) TO first
    CREATE k_seq WITH λmatch.[
        CASE match OF
        (TRUE, token, next) : [
            SEND (SELF, #match, next) TO rest
            BECOME λmatch′.[
                CASE match′ OF
                (TRUE, token′, next′) : [
                    SEND (TRUE, (token, token′), next′)
                    TO cust
                ]
                _ : [ SEND (FALSE, src) TO cust ]
                END
            ]
        ]
        _ : [ SEND (FALSE, src) TO cust ]
        END
    ]
]
```



```
LET opt_grammar_beh(grammar) = λmsg.[  # opt ::= <grammar> | <empty>;
        BECOME alt_grammar_beh(
                grammar,
                empty_grammar
        )
        SEND msg TO SELF
]
LET star_grammar_beh(grammar) = λmsg.[  # star ::= <grammar> <star> | <empty>;
        BECOME seq_grammar_beh(
                NEW seq_grammar_beh(grammar, SELF),
                empty_grammar
        )
        SEND msg TO SELF
]
LET plus_grammar_beh(grammar) = λmsg.[  # plus ::= <grammar> <grammar>*;
        BECOME seq_grammar_beh(
                grammar,
                NEW star_grammar_beh(grammar)
        )
        SEND msg TO SELF
]
```



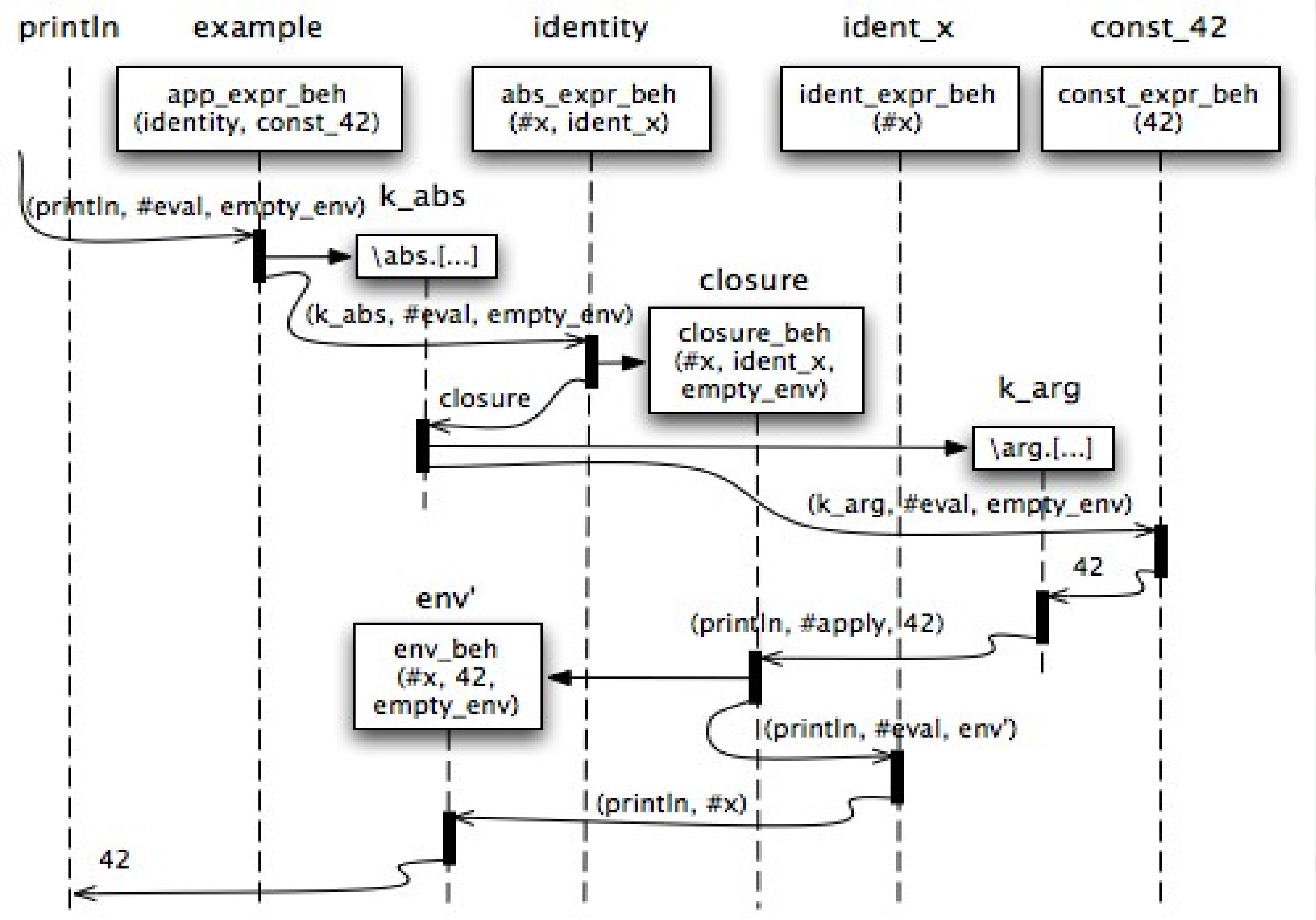



# Un-typed Lambda Calculus

```
expr  ::= <const>
        | <ident>
        | 'λ' <ident> '.' <expr>
        | <expr> '(' <expr> ')';
```

```
CREATE empty_env WITH λ(cust, _).[ SEND ? TO cust ]
LET env_beh(ident, value, next) = λ(cust, ident′).[
        IF $ident′ = $ident [ SEND value TO cust ] ELSE [ SEND (cust, ident′) TO next ]
]
LET const_expr_beh(value) = λ(cust, #eval, _).[ SEND value TO cust ]
LET ident_expr_beh(ident) = λ(cust, #eval, env).[ SEND (cust, ident) TO env ]
LET abs_expr_beh(ident, body_expr) = λ(cust, #eval, env).[
        CREATE closure WITH λ(cust, #apply, arg).[
                CREATE env′ WITH env_beh(ident, arg, env)
                SEND (cust, #eval, env′) TO body_expr
        ]
        SEND closure TO cust
]
LET app_expr_beh(abs_expr, arg_expr) = λ(cust, #eval, env).[
        SEND (k_abs, #eval, env) TO abs_expr
        CREATE k_abs WITH λabs.[
                SEND (k_arg, #eval, env) TO arg_expr
                CREATE k_arg WITH λarg.[
                        SEND (cust, #apply, arg) TO abs
                ]
        ]
]
```

# Evaluating $(\lambda x.x)(42)$



# Open Systems

- Continuous Change and Evolution
- Decentralized Decision-Making
  - Absence of Bottlenecks
  - Arms-length Relationships
- Perpetual Inconsistency
- Negotiation Among Components

–C. Hewitt and P. de Jong (1983)


Keep on Truckin'...

# References


- *It's Actors All The Way Down* <http://dalnefre.com/>
- C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intelligence*, 8(3), 1977.
- G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.
- C. Hewitt, H. Lieberman. Design Issues in Parallel Architectures for Artificial Intelligence. AI Memo 750, MIT AI Lab, 1983.
- G. Agha, I. Mason, S. Smith, and C. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, Vol. 7, No. 1, January 1997.