

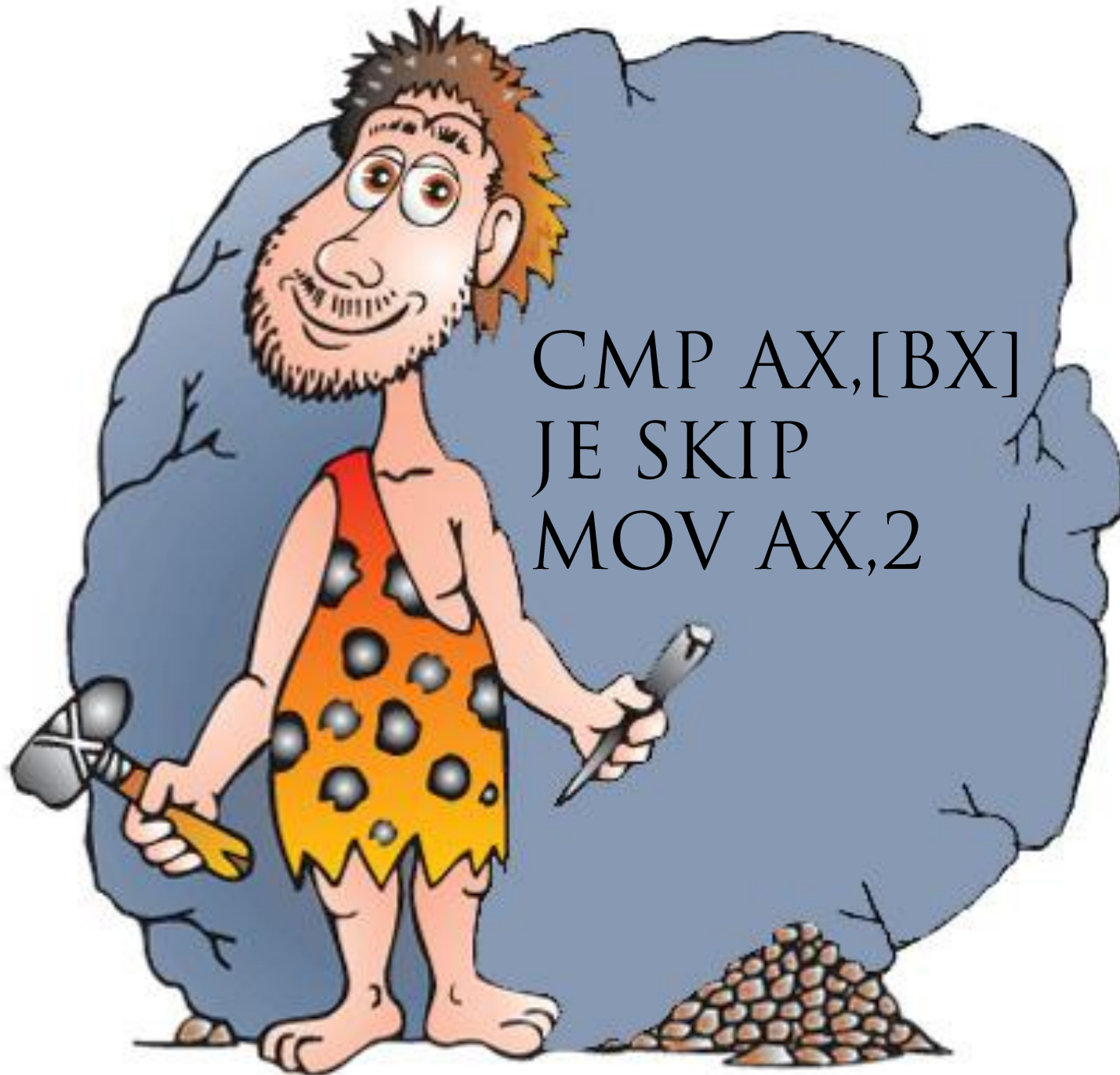
# Declarative Objects

7/22/10

Jonathan Edwards

sdg csail MIT





CMP AX,[BX]  
JE SKIP  
MOV AX,2

```
class Task {  
    int start;  
    int end;
```

```
class Task {  
    int start;  
    int end;  
    int length {  
        get {return end - start;}  
        set {end = start + value;}  
    }  
}
```

```
class Task {
    int start;
    int end;
    int length {
        get {return end - start;}
        set {end = start + value;}
    }
    // start after t, increment length
    void slipAfter(Task t) {
```

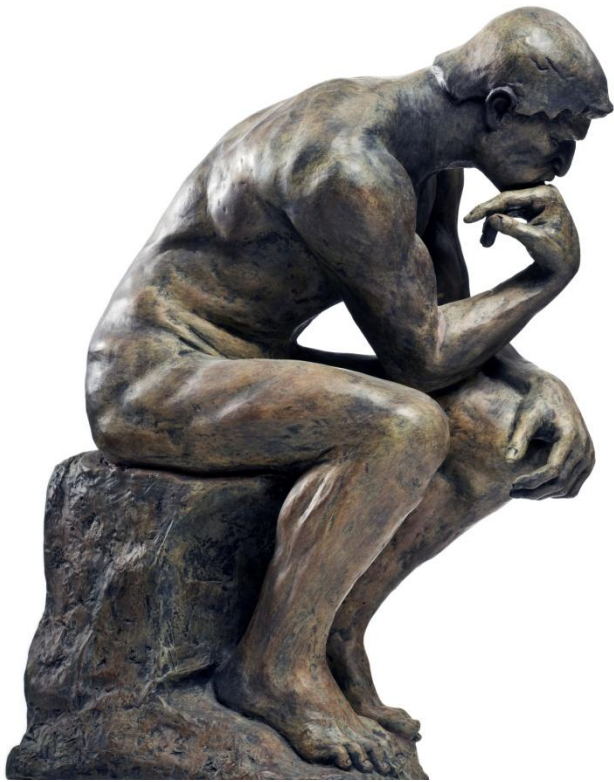
```
class Task {
    int start;
    int end;
    int length {
        get {return end - start;}
        set {end = start + value;}
    }
    // start after t, increment length
    void slipAfter(Task t) {
        start = t.end;
        length = length + 1;
    }
}
```

```
class Task {
    int start;
    int end;
    int length {
        get {return end - start;}
        set {end = start + value;}
    }
    // start after t, increment length
    void slipAfter(Task t) {
        start = t.end;
        length = length + 1; X
    }
}
```

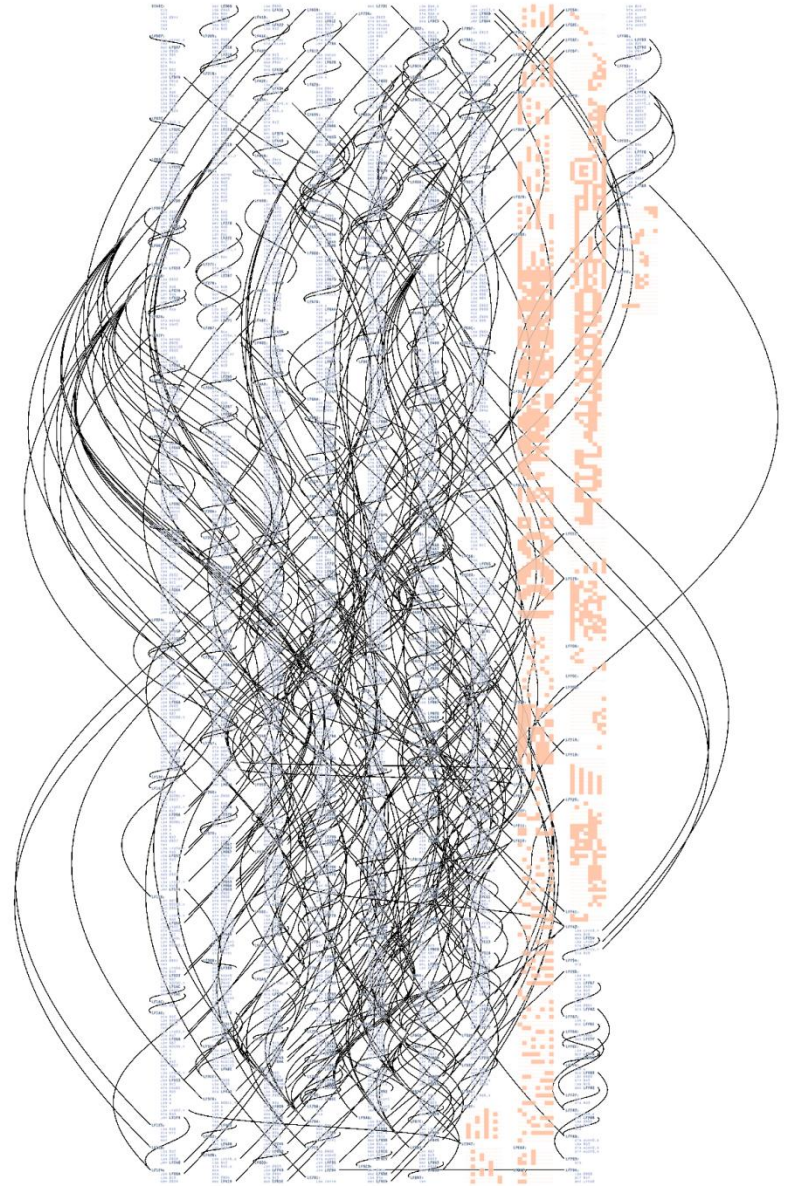
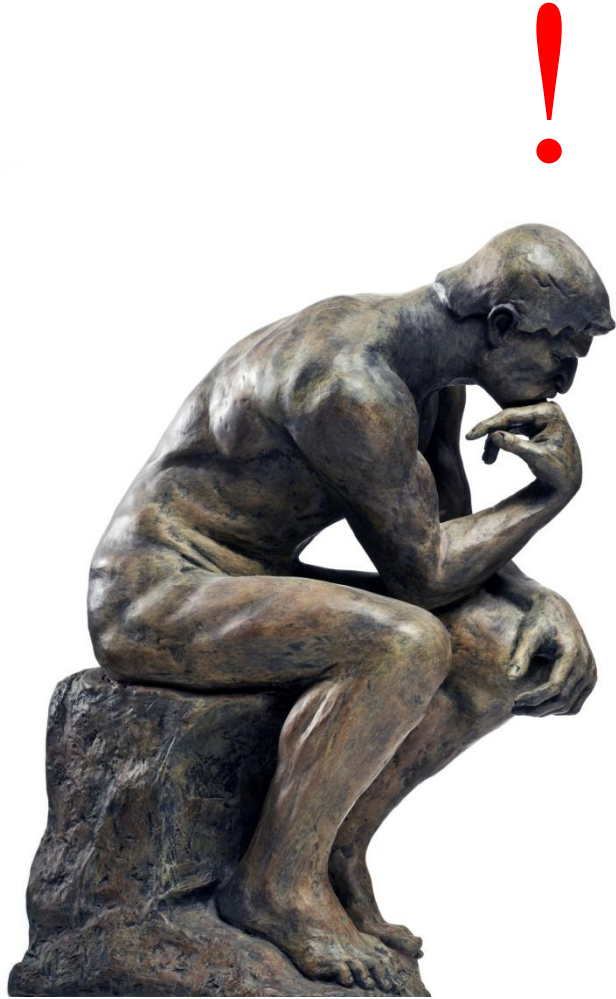
```
class Task {
    int start;
    int end;
    int length {
        get {return end - start;}
        set {end = start + value;}
    }
    // start after t, increment length
    void slipAfter(Task t) {
        length = length + 1; X
        start = t.end;
    }
}
```

```
class Task {
    int start;
    int end;
    int length {
        get {return end - start;}
        set {end = start + value;}
    }
    // start after t, increment length
    void slipAfter(Task t) {
        int oldLength = length;
        start = t.end;
        length = oldLength + 1;
    }}
```

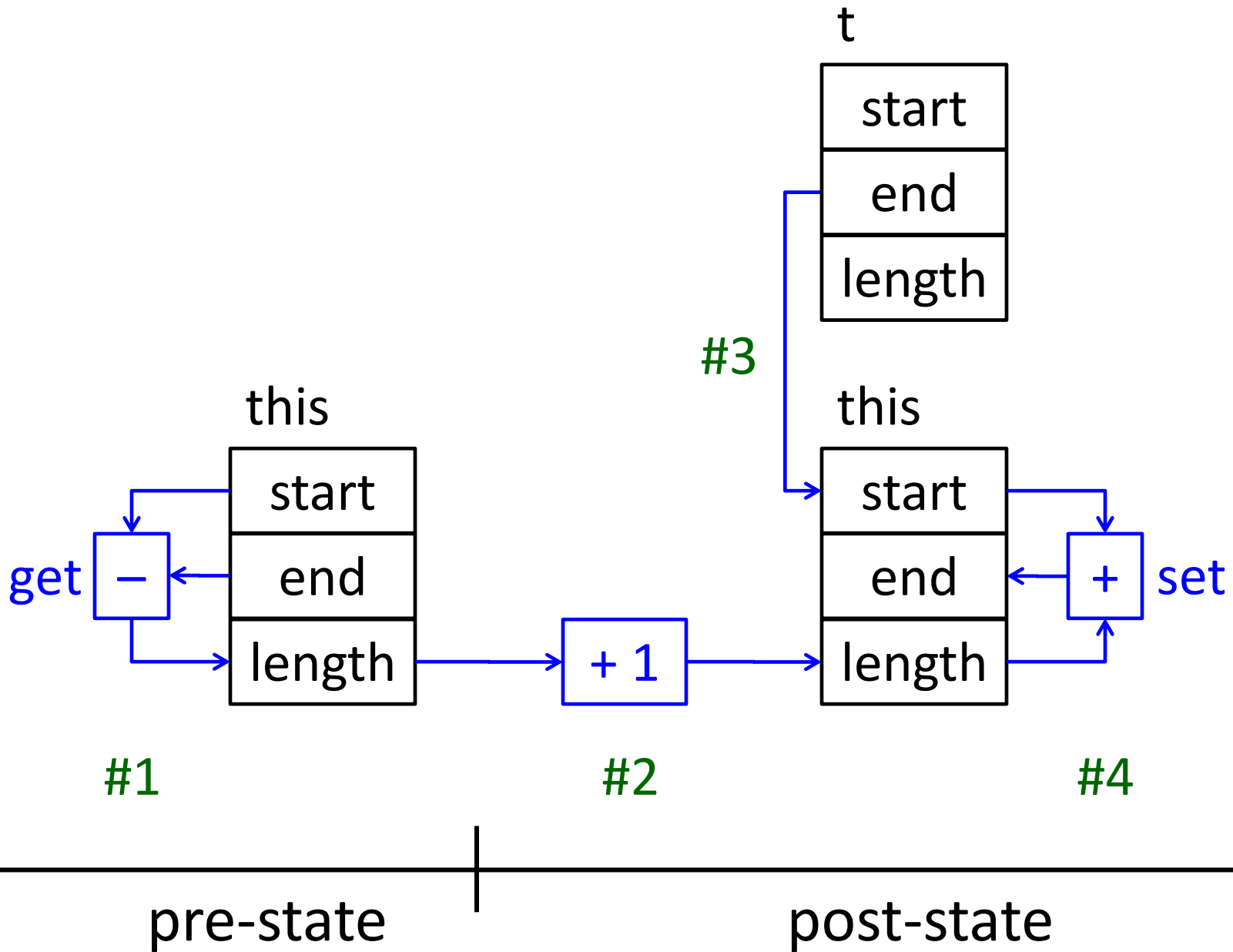
```
class Task {
    int start {
        get {return end - length;}
        set {end = value + length;}
    }
    int end;
    int length;
    // start after t, increment length
    void slipAfter(Task t) {
        length = length + 1;
        start = t.end;
    }
}
```



```
class Task {
    int start;
    int end;
    int length {
        get {return end - start;}
        set {end = start + value;}
    }
    void slipAfter(Task t) {
        int oldLength = length;
        start = t.end;
        length = oldLength + 1;
    }
}
```

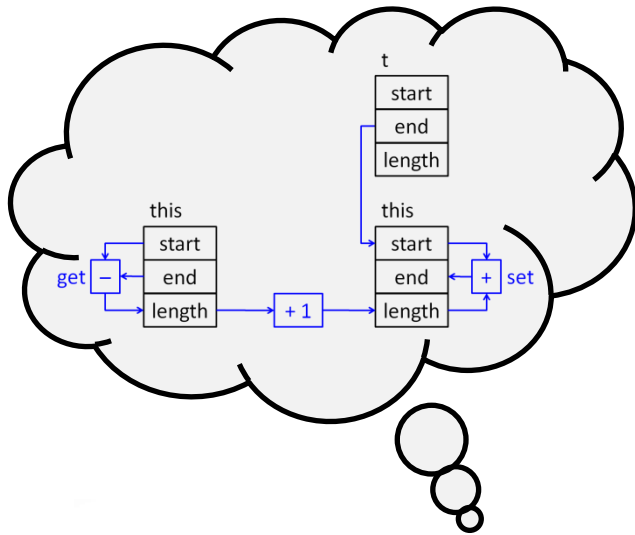


Pac-Man

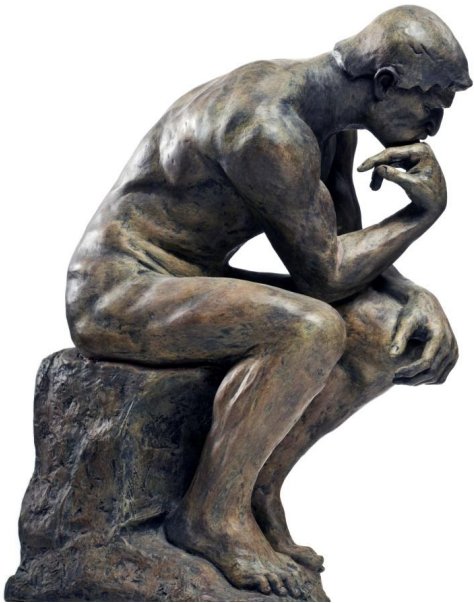


```
Task = obj {  
  start: int  
  end: int  
  length ::= end – start  
  length trig {  
    end <= start + length}  
  slipAfter = act {  
    t: Task  
    start <= t.end  
    length <= \length + 1}}
```

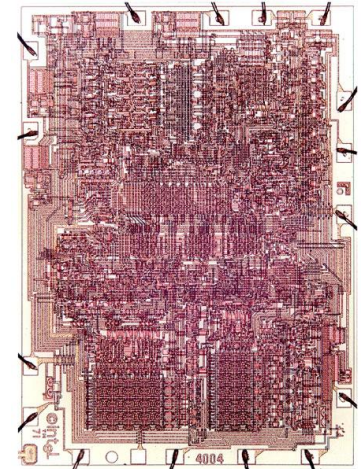
```
Task = obj {  
    start: int  
    end: int  
#1    length ::= end - start  
    length trig {  
#4        end <= start + length}  
    slipAfter = act {  
        t: Task  
#3        start <= t.end  
#2        length <= \length + 1}}
```



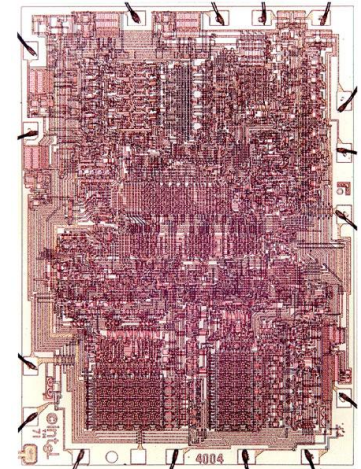
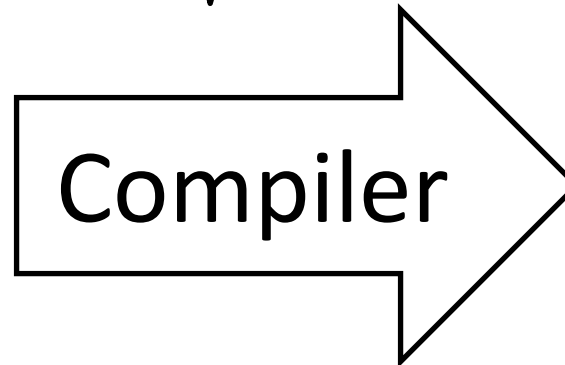
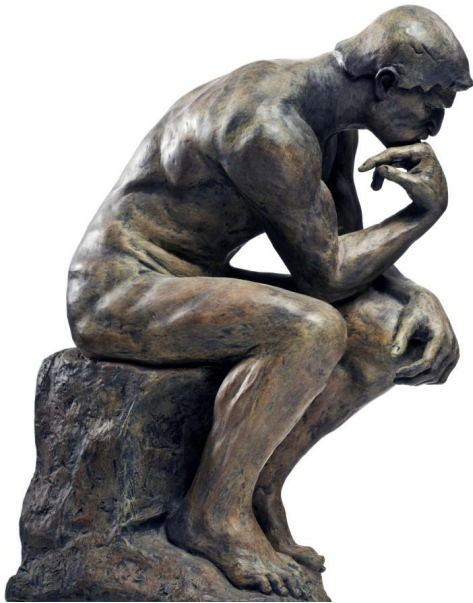
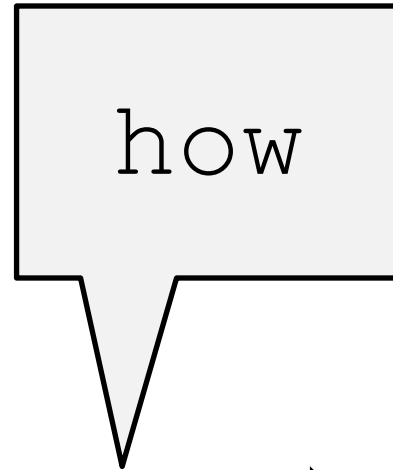
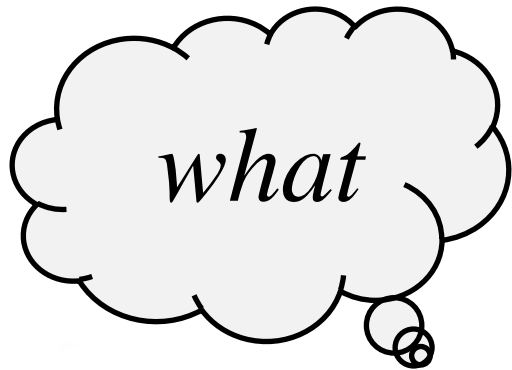
1.  $\backslash\text{length} := \text{end} - \text{start}$
2.  $\text{length} := \backslash\text{length} + 1$
3.  $\text{start} := \text{t.end}$
4.  $\text{end} := \text{start} + \text{length}$



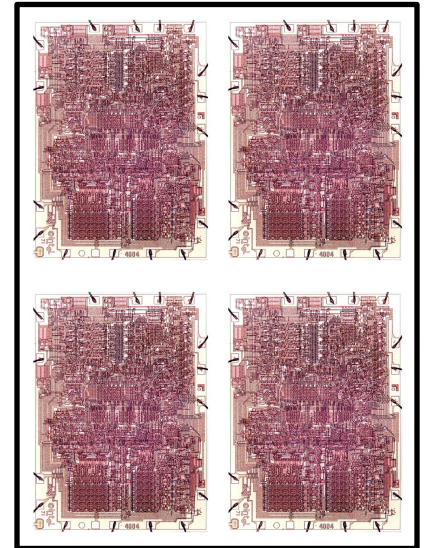
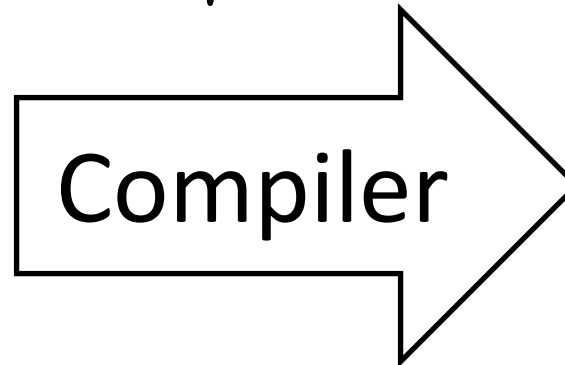
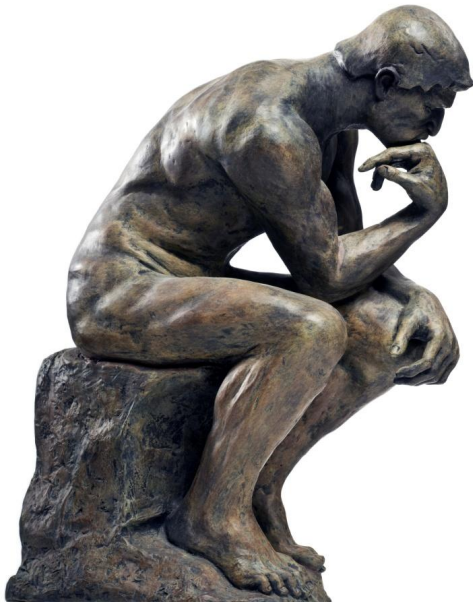
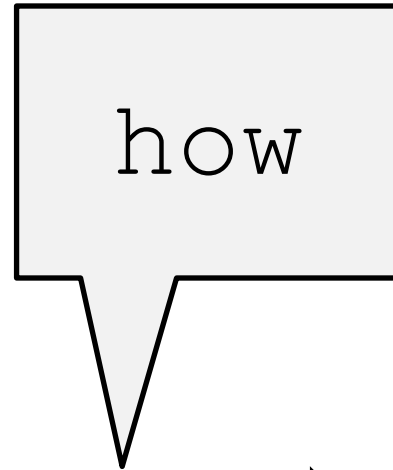
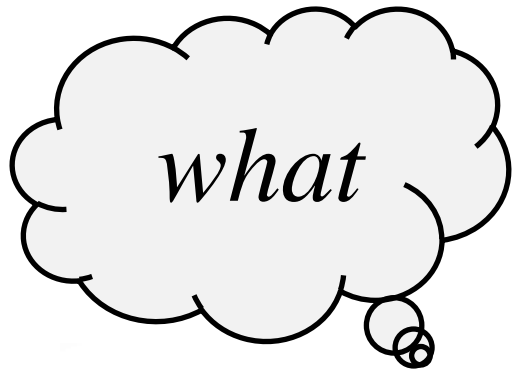
Compiler



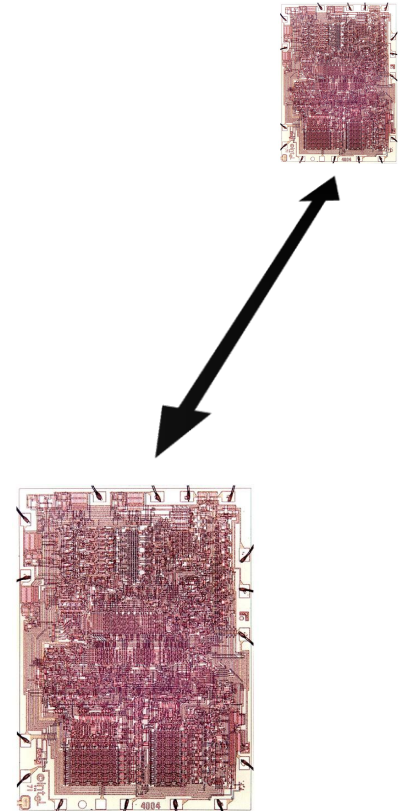
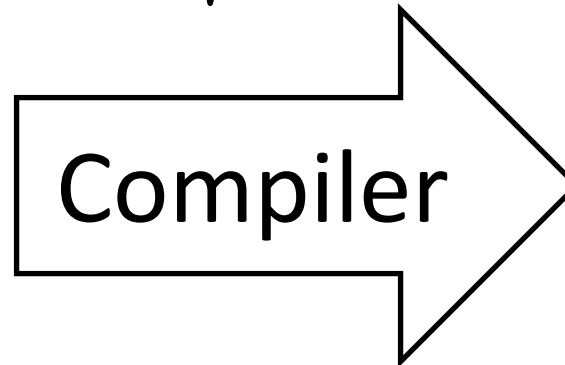
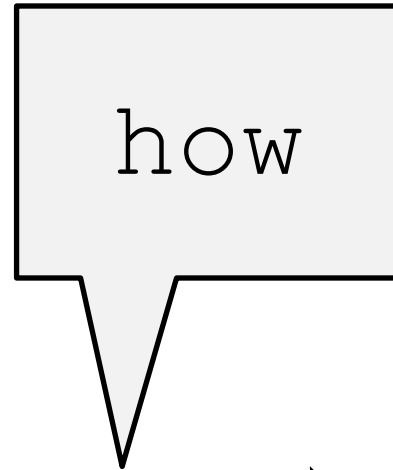
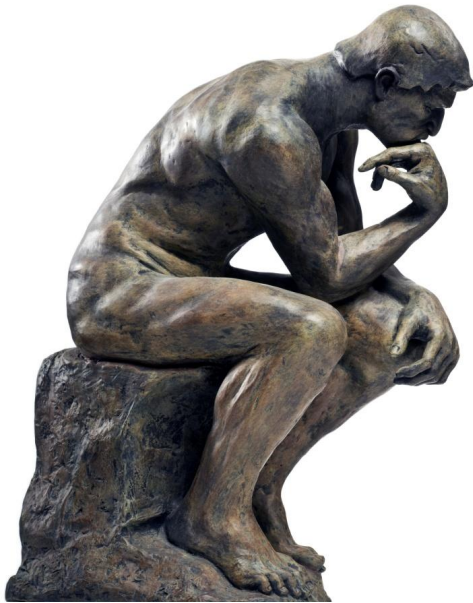
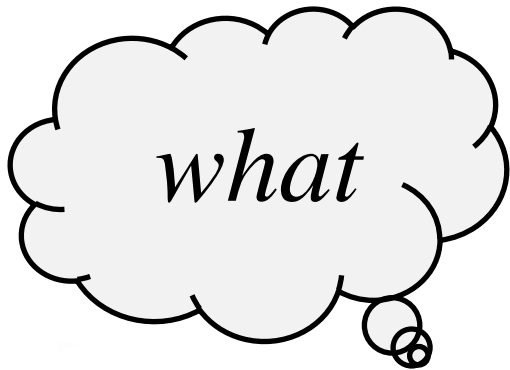
# Declarative Programming



# Parallel Processing

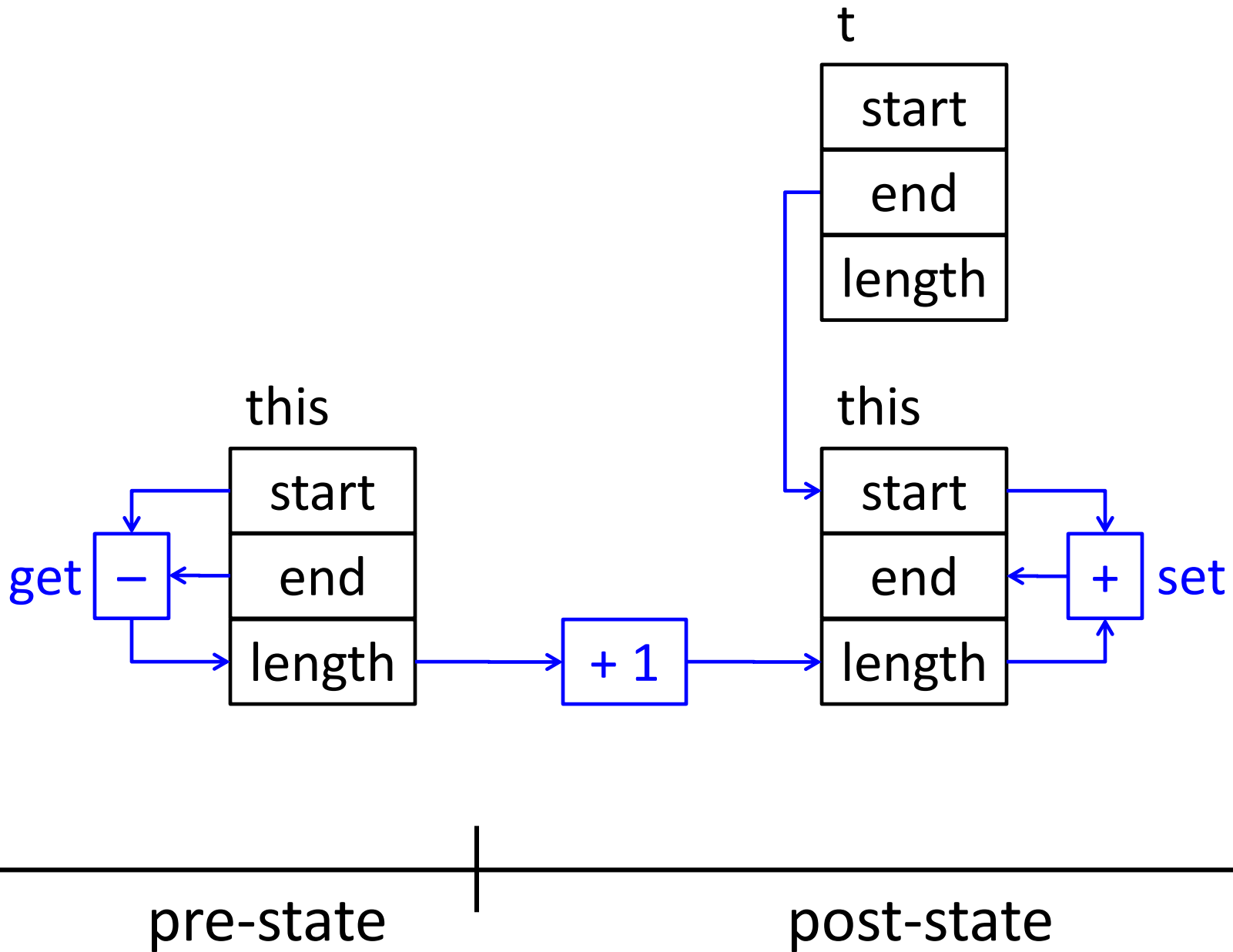


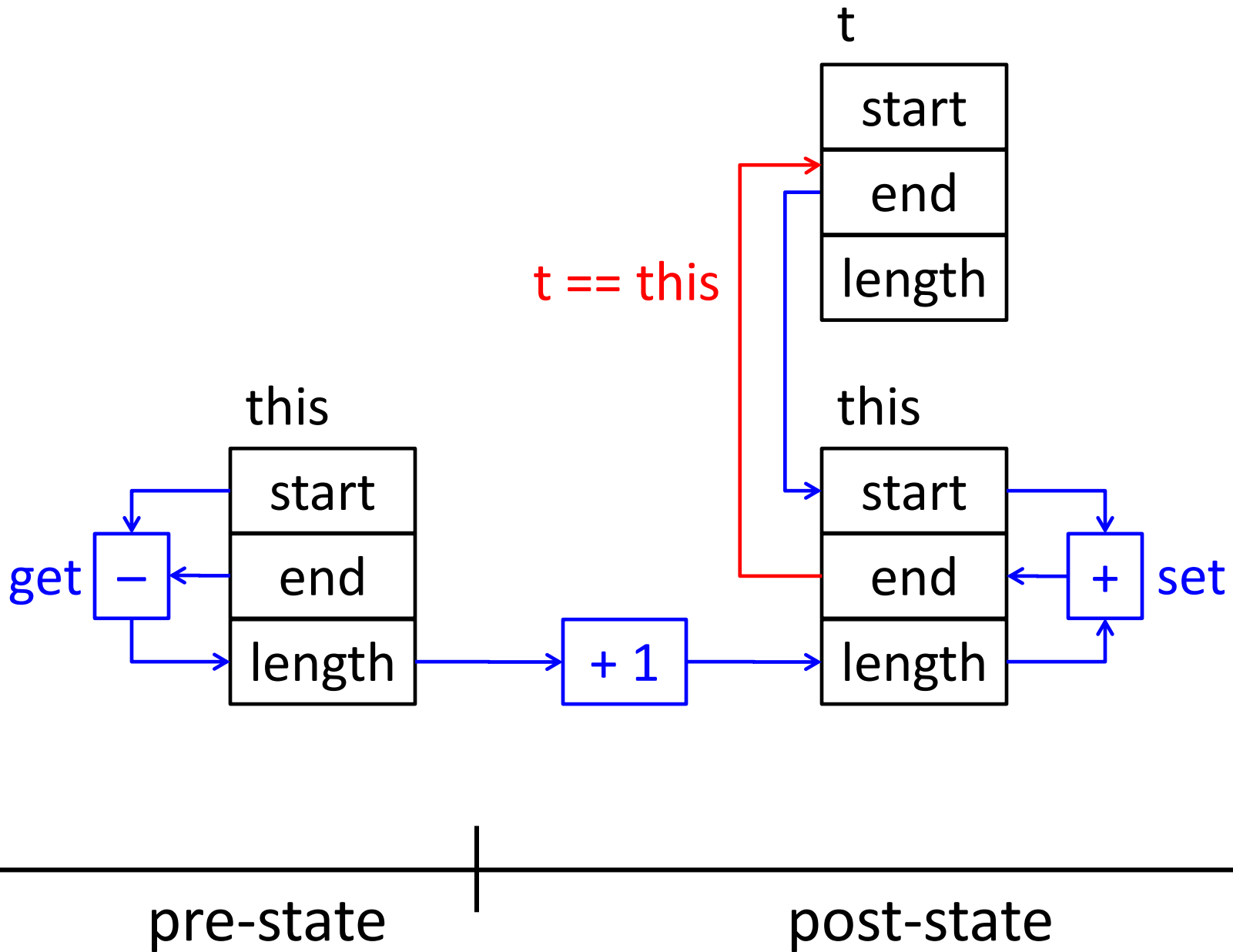
# Distributed Processing

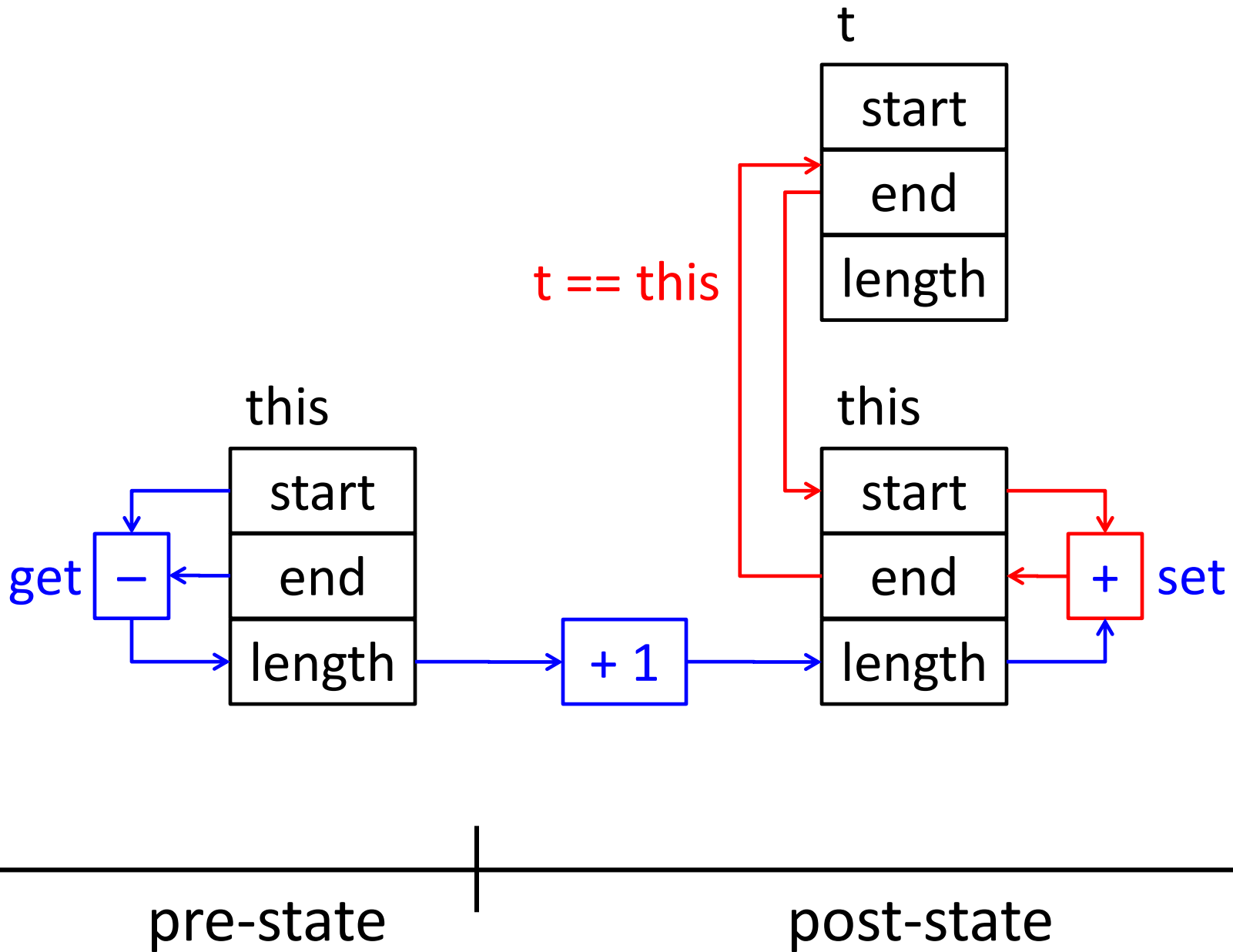




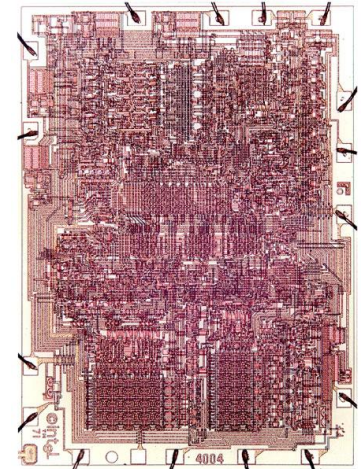
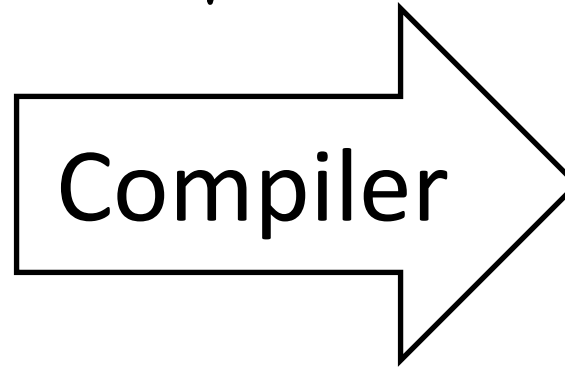
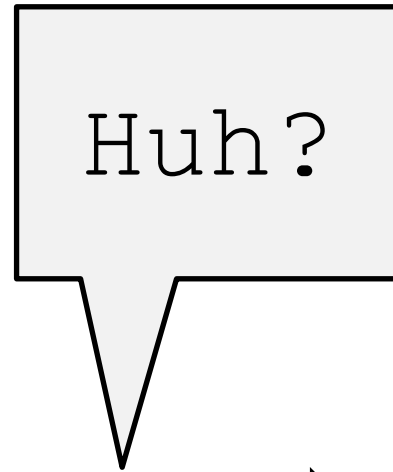
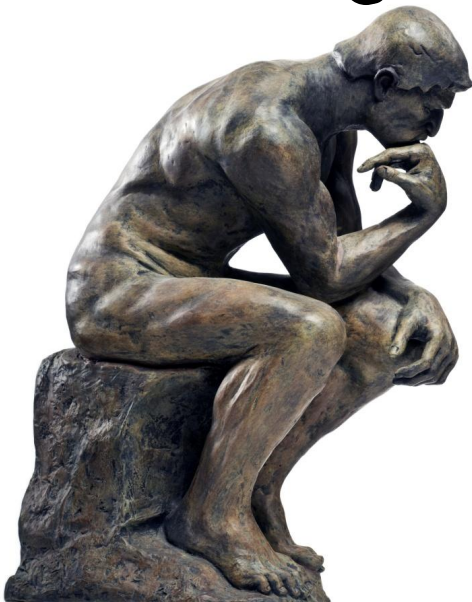
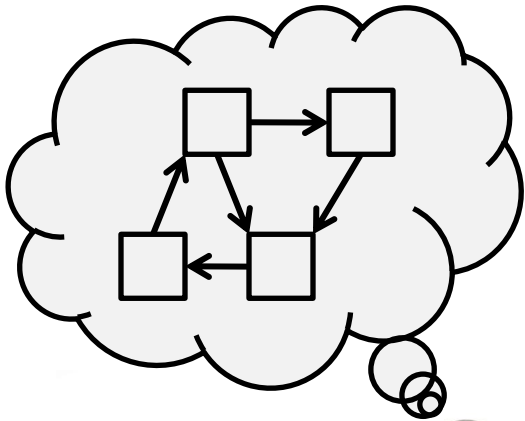
# Pointers







# pointers $\Rightarrow$ undecidable dataflow



# Pick two

~~Declarative  
programming~~

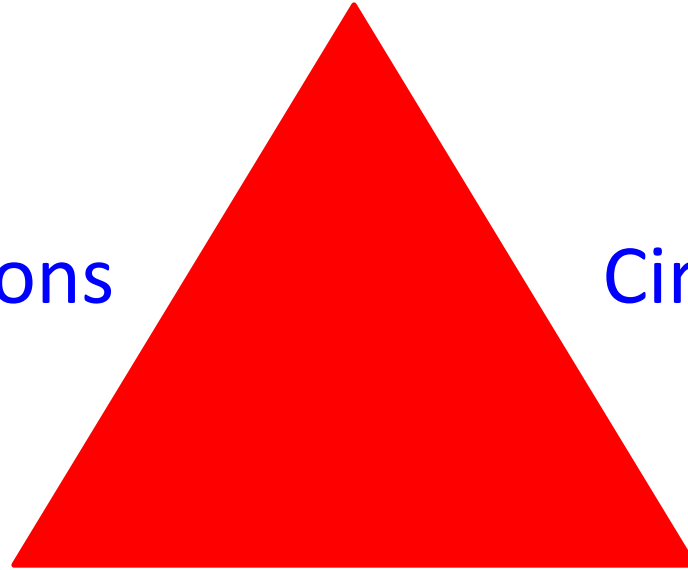
Functions

Circuits

~~Data  
structures~~

Imperative  
programming

~~Mutable  
state~~

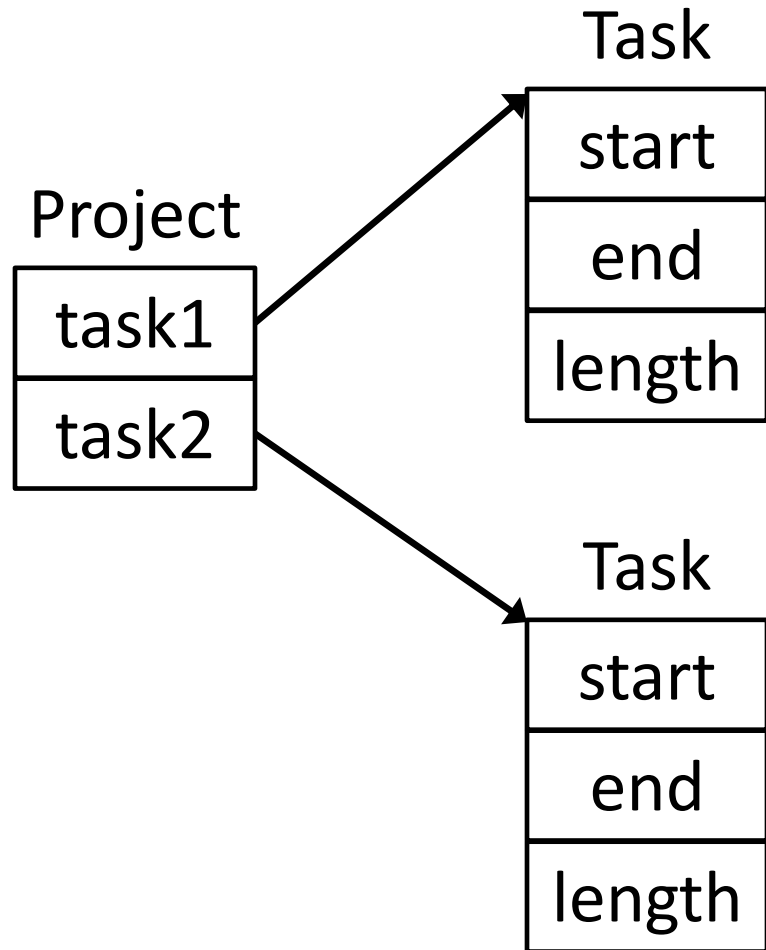




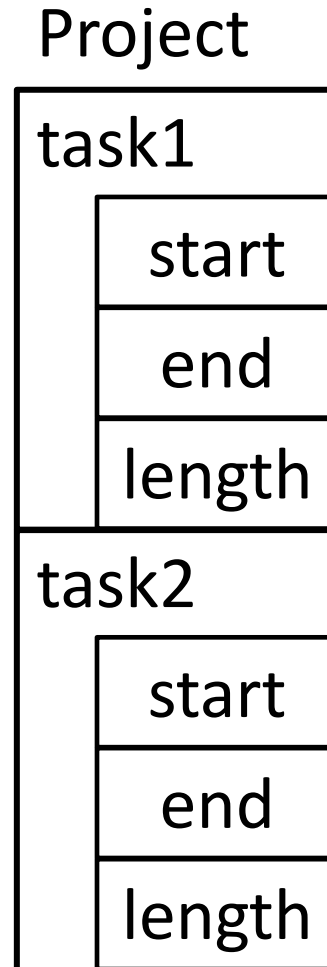
Nesting &  
Binding

Model-View  
dataflow

```
Project = obj {  
  task1: Task  
  task2: Task  
}
```



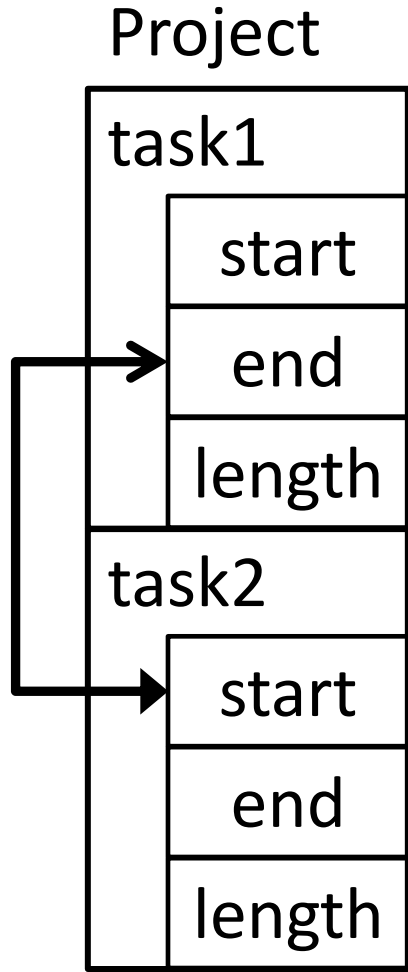
```
Project = obj {  
  task1: Task  
  task2: Task  
}
```



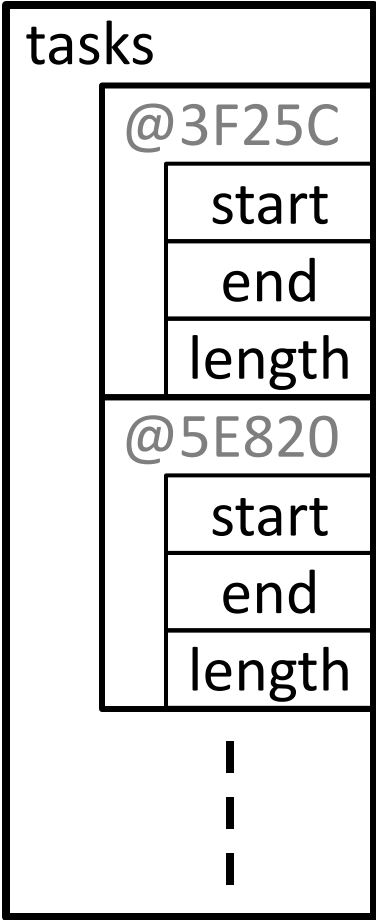
```
Project = obj {  
  task1: Task  
  task2: Task  
  task2.start ::=> task1.end  
}
```

task2.start ::=> task1.end

bidirectional binding



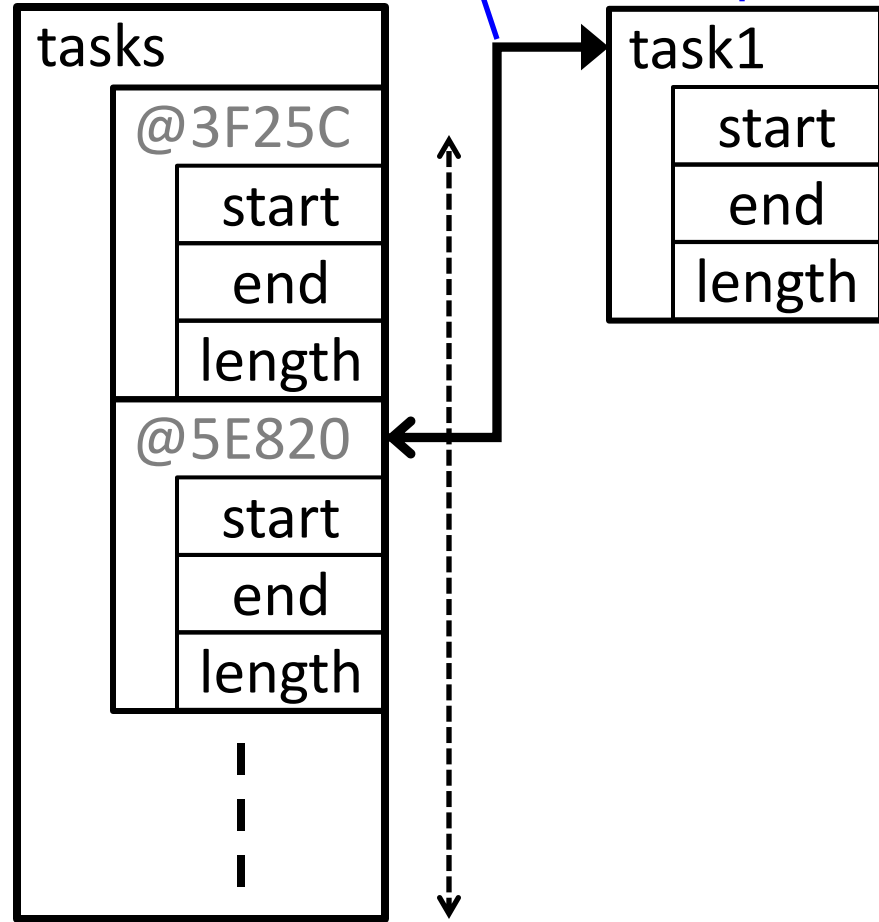
tasks: **dom** Task



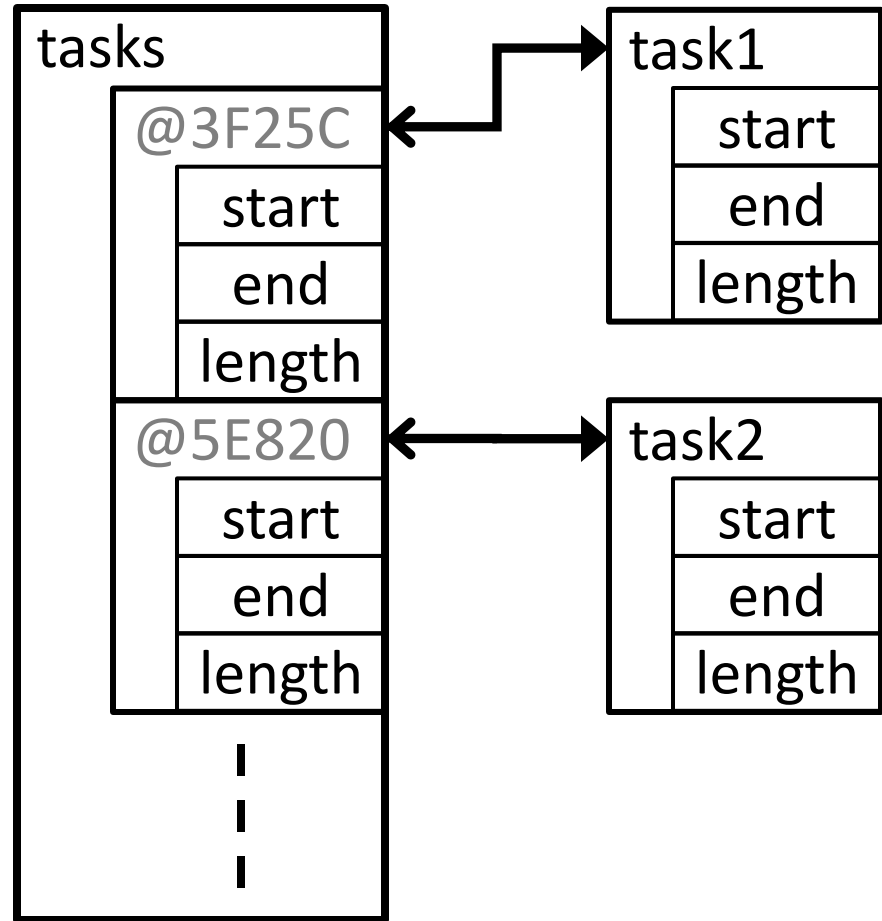
tasks: **dom** Task  
task1 \*=> tasks

quantified binding

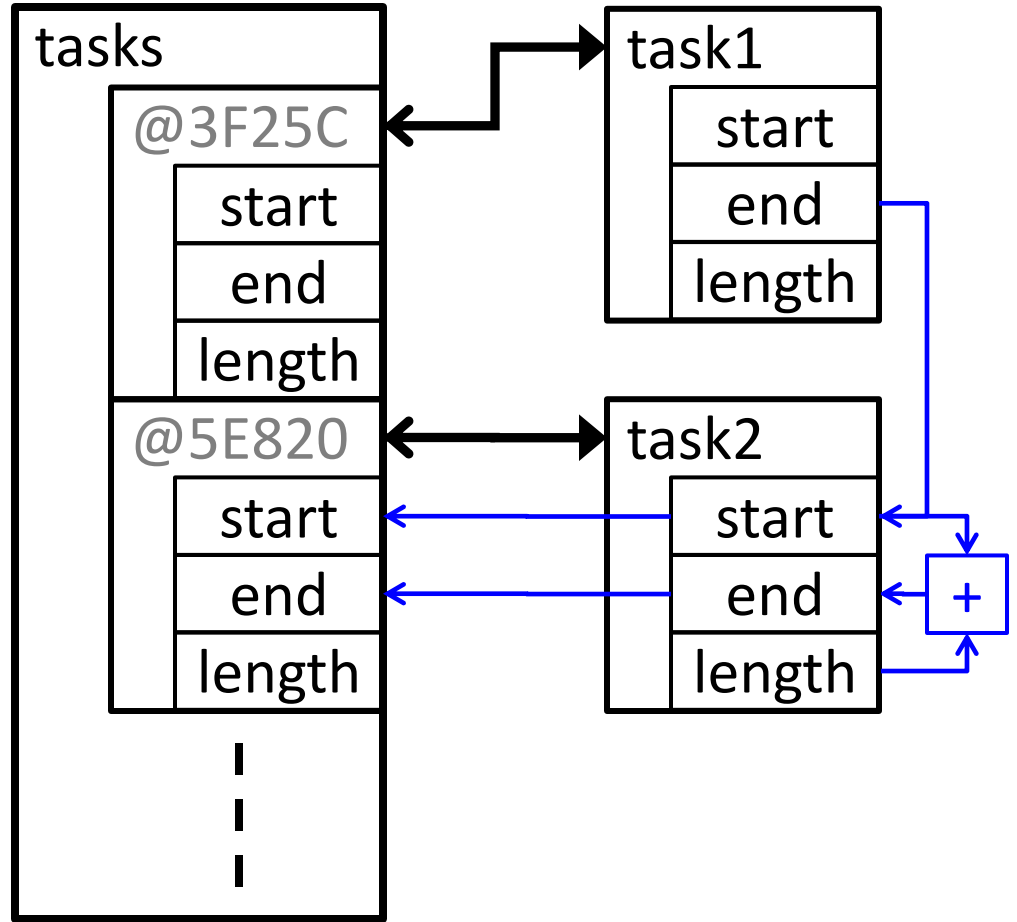
cursor



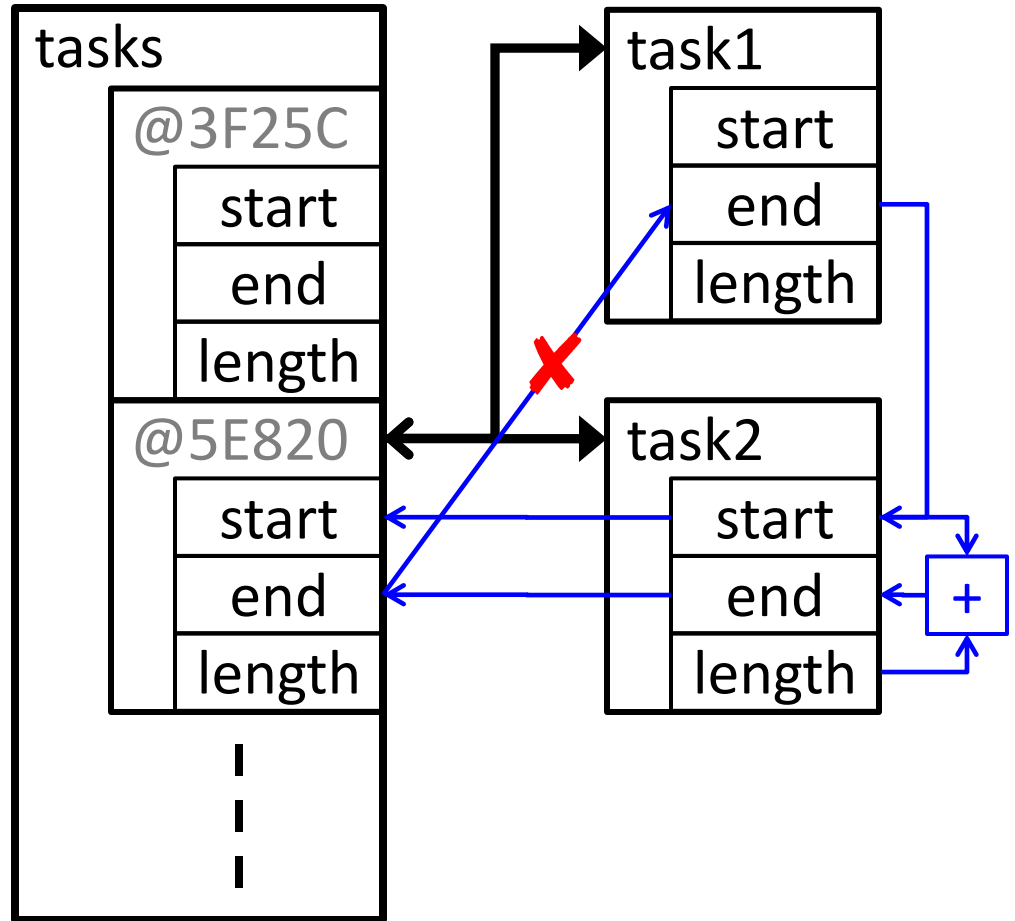
tasks: **dom** Task  
task1 **\*=>** tasks  
task2 **\*=>** tasks

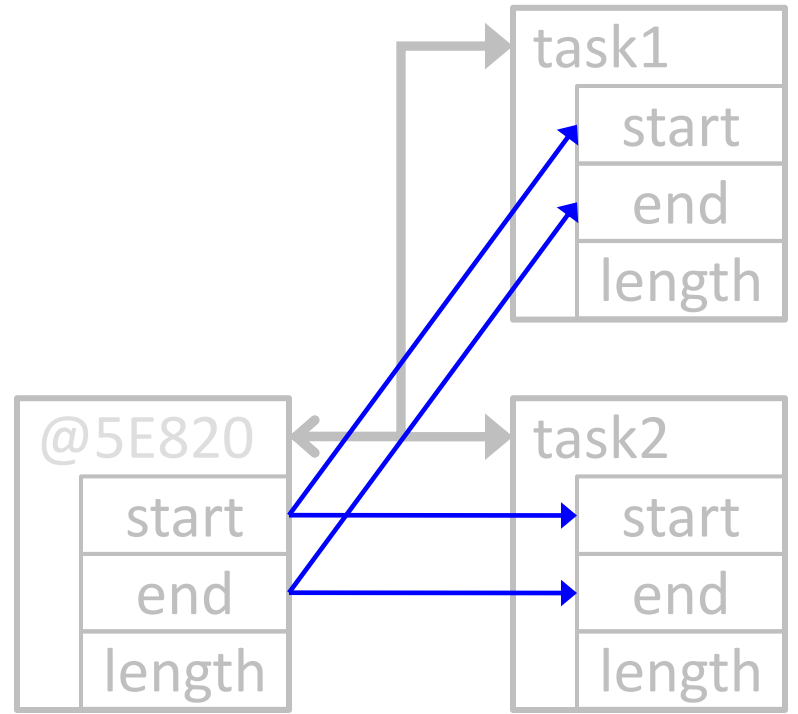
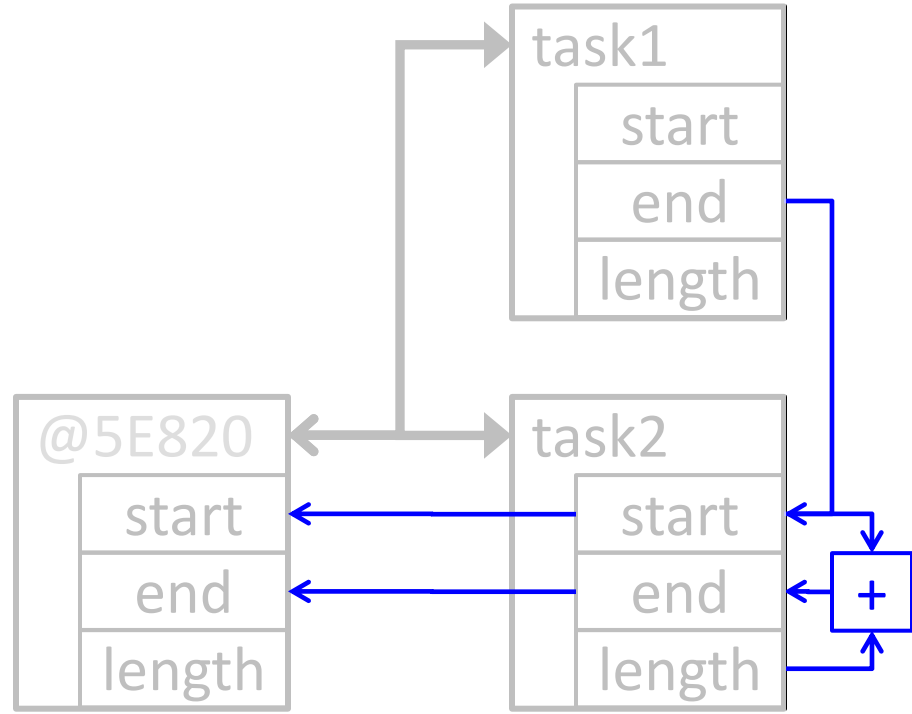


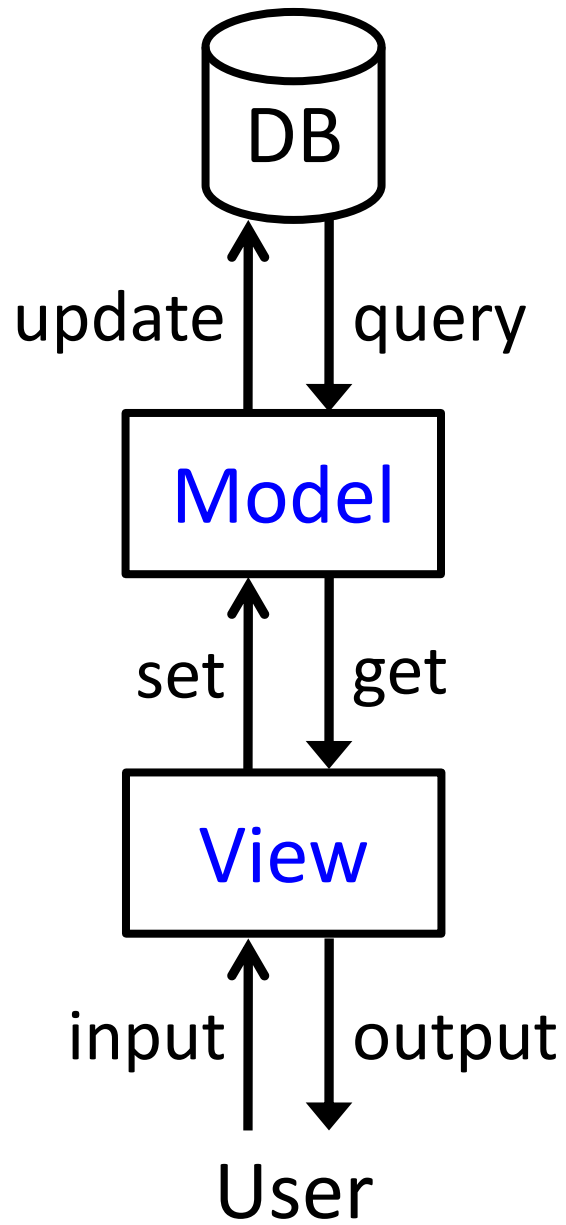
```
tasks: dom Task
task1 ==> tasks
task2 ==> tasks
task2.slipAfter(task1)
```



```
tasks: dom Task
task1 ==> tasks
task2 ==> tasks
task2.slipAfter(task1)
```



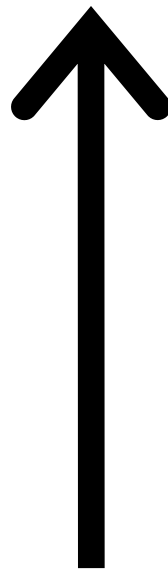




# Model-View dataflow

internal state

- inputs
- setters
- triggers



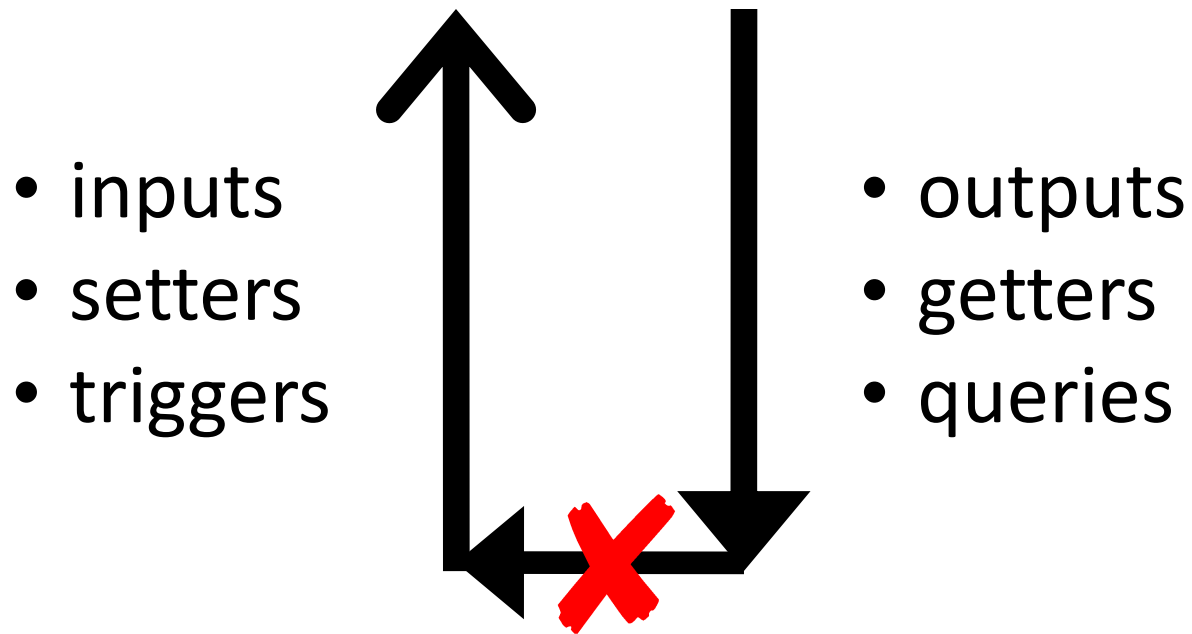
- outputs
- getters
- queries



external world

# Model-View dataflow

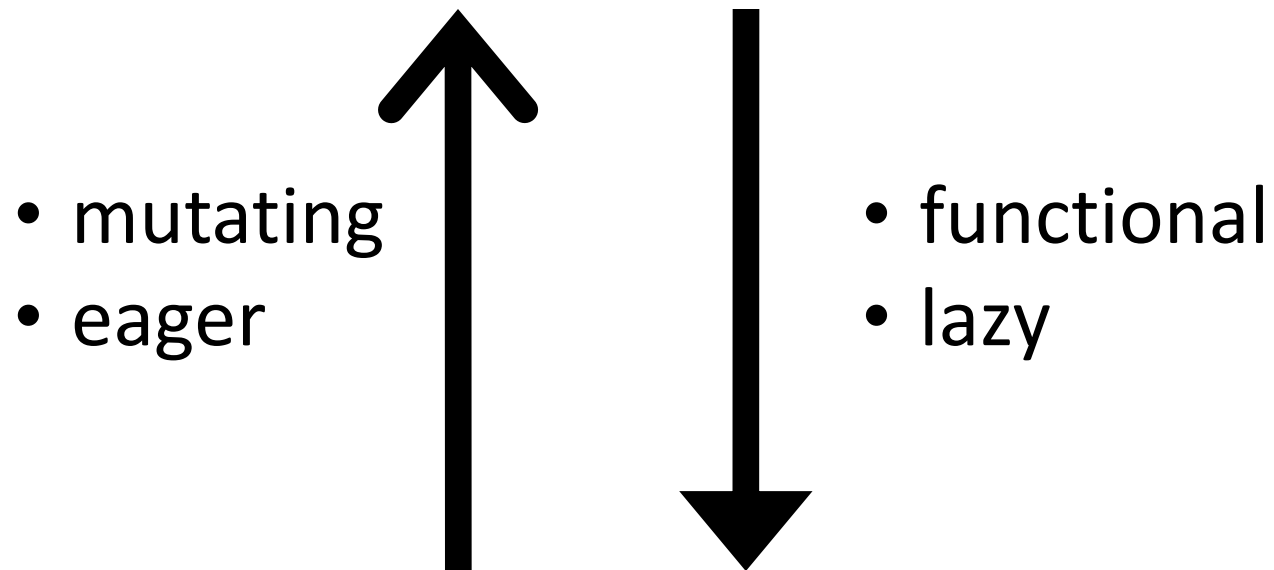
internal state



external world

# Model-View dataflow

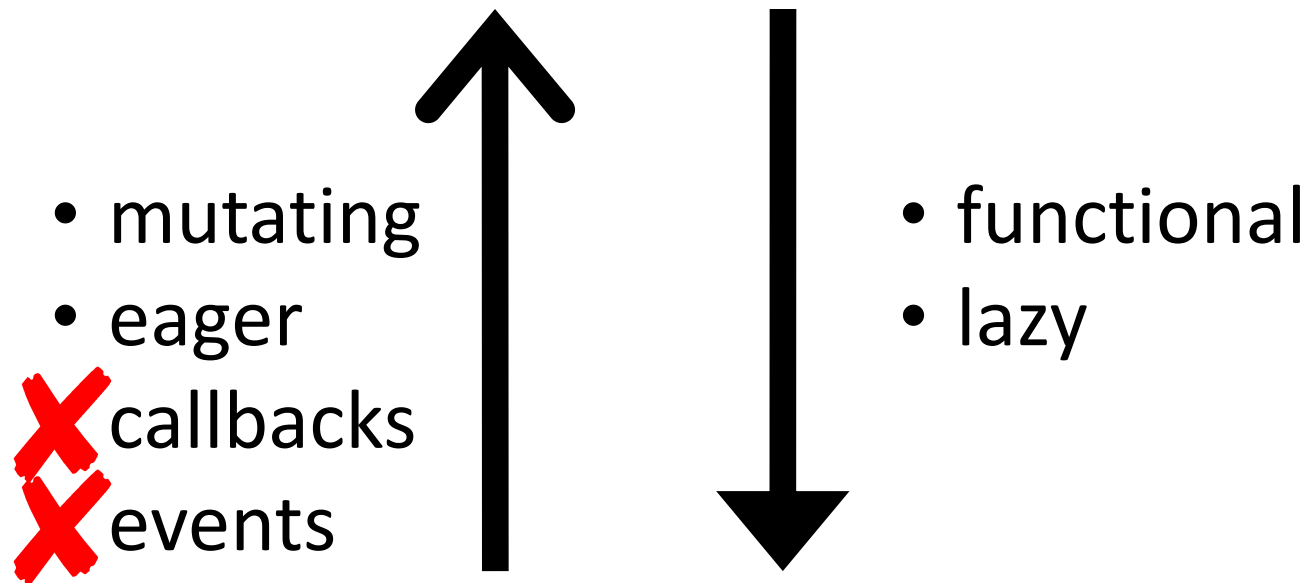
internal state



external world

# Model-View dataflow

internal state



external world

# New rules, new patterns

- Synchronous reactive programming
  - Input event triggers atomic state transition
  - Output is pure function of new state
  - Asynchronicity layered on top
- Syntax order irrelevant – no control flow
- Pre-state readable throughout transition
- Fields can change once per transition
- Actions can't see effects of own changes

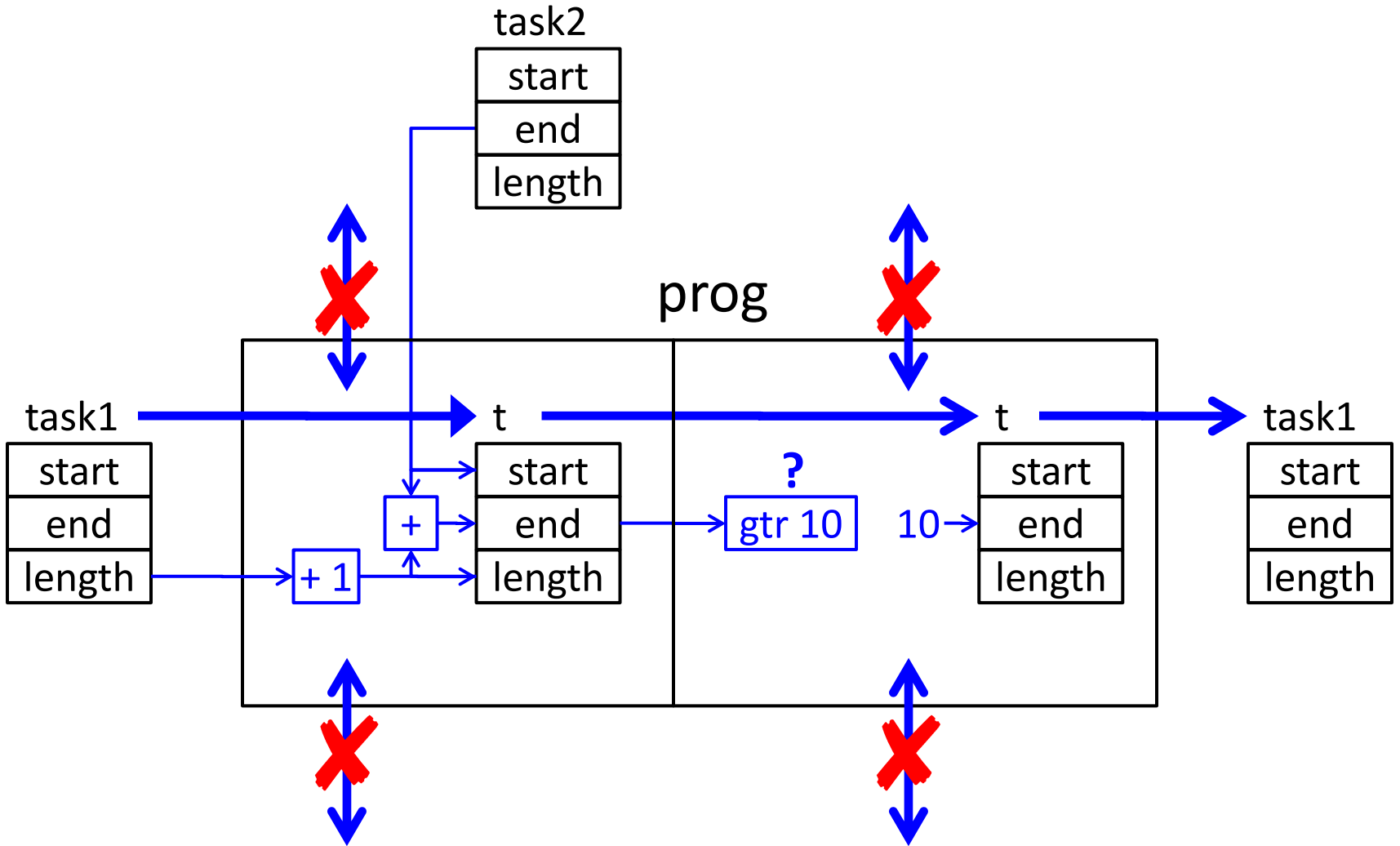
# Progression

```
prog {  
  t +=> task1  
  step {  
    t.slipAfter(task2)  
    step  
    \t.end ?gtr 10  
    t.end <= 10  
  }  
}
```

progressive binding

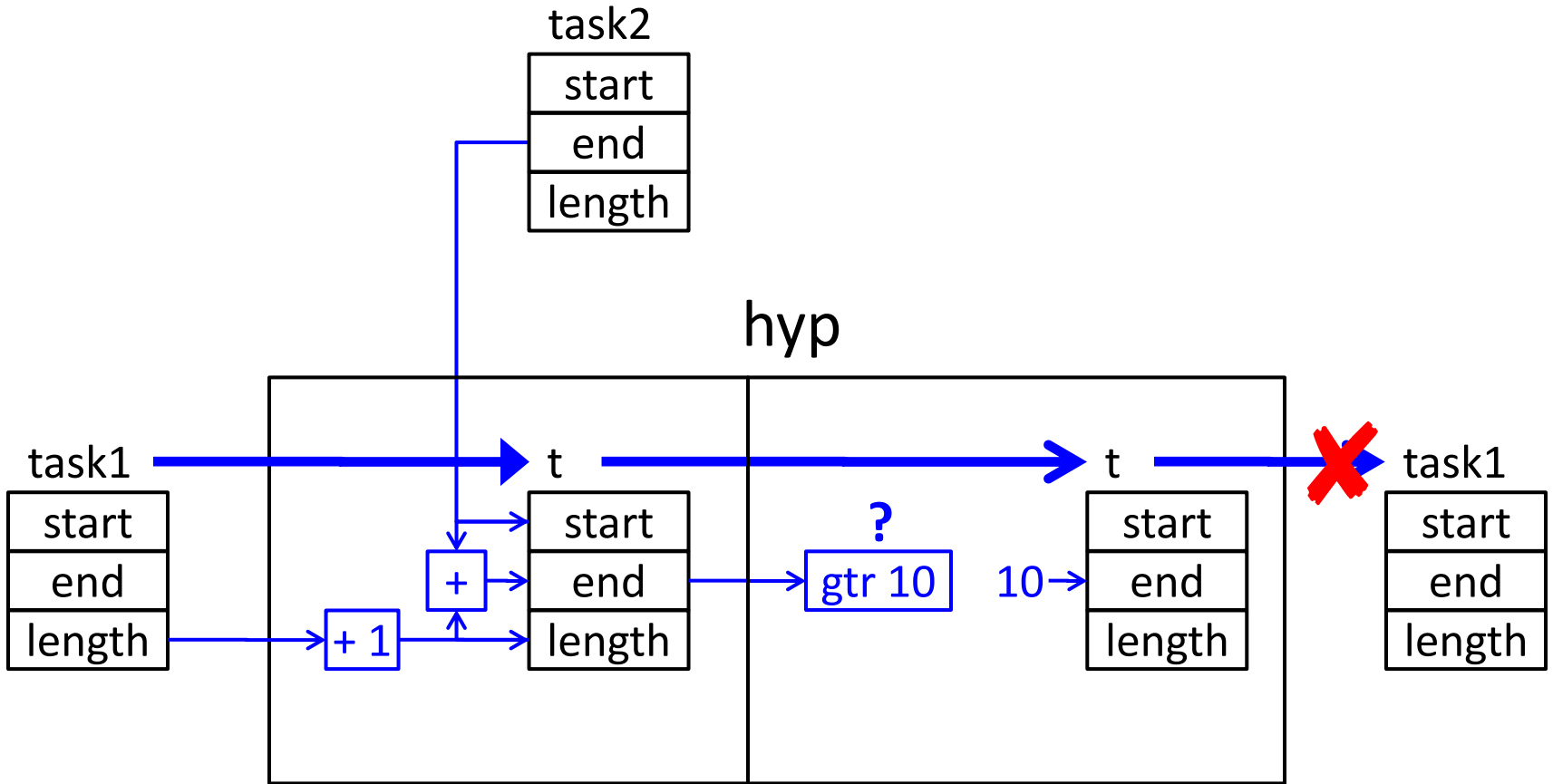
result of prev step

guard



pre-state

post-state



pre-state

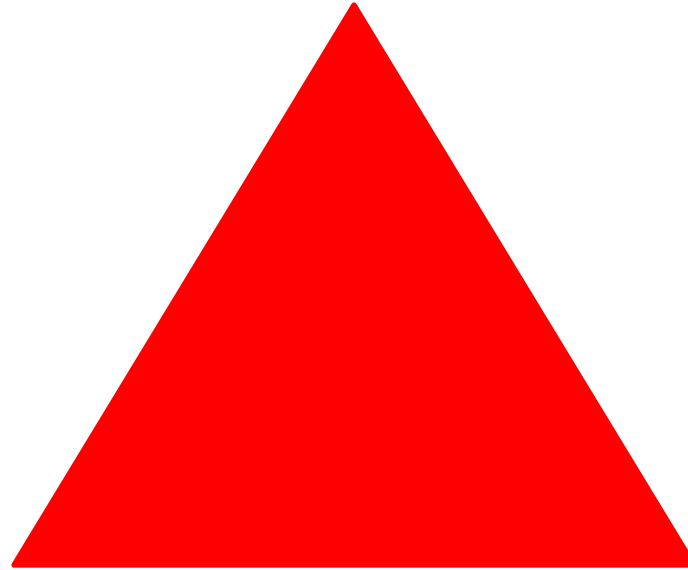
post-state

# Technical Summary

- Nesting gives objects a location in global tree
- Bindings propagate changes through relative paths in tree (precise static effects)
- Bindings are directed: input, output, or both
  - Input is change-driven, cascades eagerly
  - Output is lazy pure functional
  - Each is statically acyclic based on tree path effects
  - Outputs do not feedback to inputs
- Imperative islands in a declarative sea

Pick two

Declarative  
programming

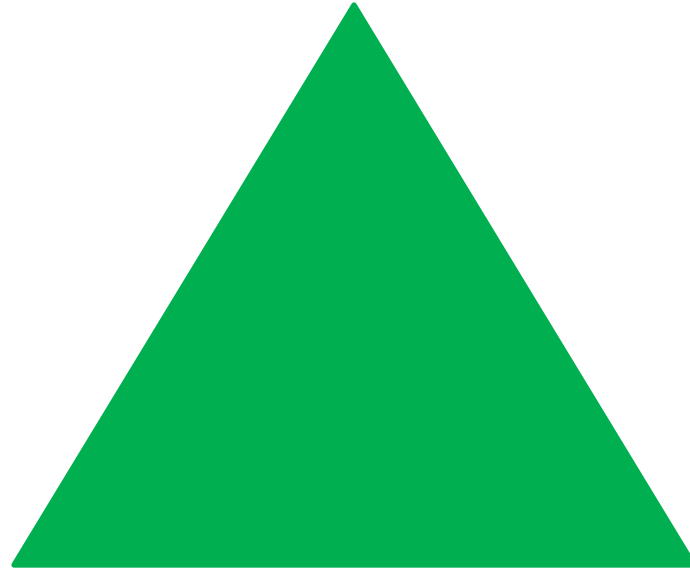


Data  
structures

Mutable  
state

# Declarative Objects

Declarative  
programming



Nesting &  
Binding

Model-View  
dataflow

# Imperative Programming



# Declarative Programming

