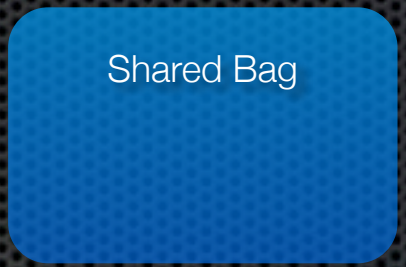
Paradigms for Process Interaction

- Three organizations
  - coordinator
    - an example of manager/workers style
  - symmetric
    - an example of heartbeat style
  - ring
    - an example of pipeline style

Manager / Workers

Manager / Workers

- Also known as distributed bag of tasks or work farm model



Manager / Workers

- Also known as distributed bag of tasks or work farm model

Manager / Workers

- Also known as distributed bag of tasks or work farm model

- Each worker:

Manager / Workers

- Also known as distributed bag of tasks or work farm model

- Each worker:
  - gets a task

Manager / Workers

- Also known as distributed bag of tasks or work farm model

- Each worker:
  - gets a task
  - does it, and

Manager / Workers

- Also known as distributed bag of tasks or work farm model

- Each worker:
  - gets a task
  - does it, and
  - optionally produces new tasks

Manager / Workers

- Also known as distributed bag of tasks
  or work farm model

- Each worker:
  - gets a task
  - does it, and
  - optionally produces new tasks

- Requirement for using the bag of tasks -- independent tasks

Manager / Workers

- Also known as distributed bag of tasks
  or work farm model

- Each worker:
  - gets a task
  - does it, and
  - optionally produces new tasks

- Requirement for using the bag of tasks -- independent tasks
  - Static number -- primes, matrix mult., words program

Manager / Workers

- Also known as distributed bag of tasks
  or work farm model

- Each worker:
  - gets a task
  - does it, and
  - optionally produces new tasks

- Requirement for using the bag of tasks -- independent tasks
  - Static number -- primes, matrix mult., words program
  - Dynamic number -- recursive parallelism

Manager / Workers

- Also known as distributed bag of tasks
  or work farm model

- Each worker:
  - gets a task
  - does it, and
  - optionally produces new tasks

- Requirement for using the bag of tasks -- independent tasks
  - Static number -- primes, matrix mult., words program
  - Dynamic number -- recursive parallelism
- Advantages: scalability and load balancing

Distributed Implementation

Distributed Implementation

- `Bag -- manager process`

Distributed Implementation

- Bag -- manager process
- Workers send messages to the manager and receive new tasks from the manager

Distributed Implementation

- Bag -- manager process
- Workers send messages to the manager and receive new tasks from the manager
- Manager is a server process with two kinds of operations:

Distributed Implementation

- Bag -- manager process
- Workers send messages to the manager and receive new tasks from the manager
- Manager is a server process with two kinds of operations:
  - get a task

Distributed Implementation

- Bag -- manager process
- Workers send messages to the manager and receive new tasks from the manager
- Manager is a server process with two kinds of operations:
  - get a task
  - produce a result

Distributed Implementation

- Bag -- manager process
- Workers send messages to the manager and receive new tasks from the manager
- Manager is a server process with two kinds of operations:
  - get a task
  - produce a result
- How can the manager detect termination?

Distributed Implementation

- Bag -- manager process
- Workers send messages to the manager and receive new tasks from the manager
- Manager is a server process with two kinds of operations:
  - get a task
  - produce a result
- How can the manager detect termination?
  - every worker is waiting to get a new task and the bag is empty

Linda

Linda

Linda

- Linda is called a coordination language
  - It is both a language and a kind of library
  - It allows one to easily implement replicated workers with a "shared" bag

Linda

- Linda is called a coordination language
  - It is both a language and a kind of library
  - It allows one to easily implement replicated workers with a "shared" bag
- Basic idea: processes share a *tuple space*

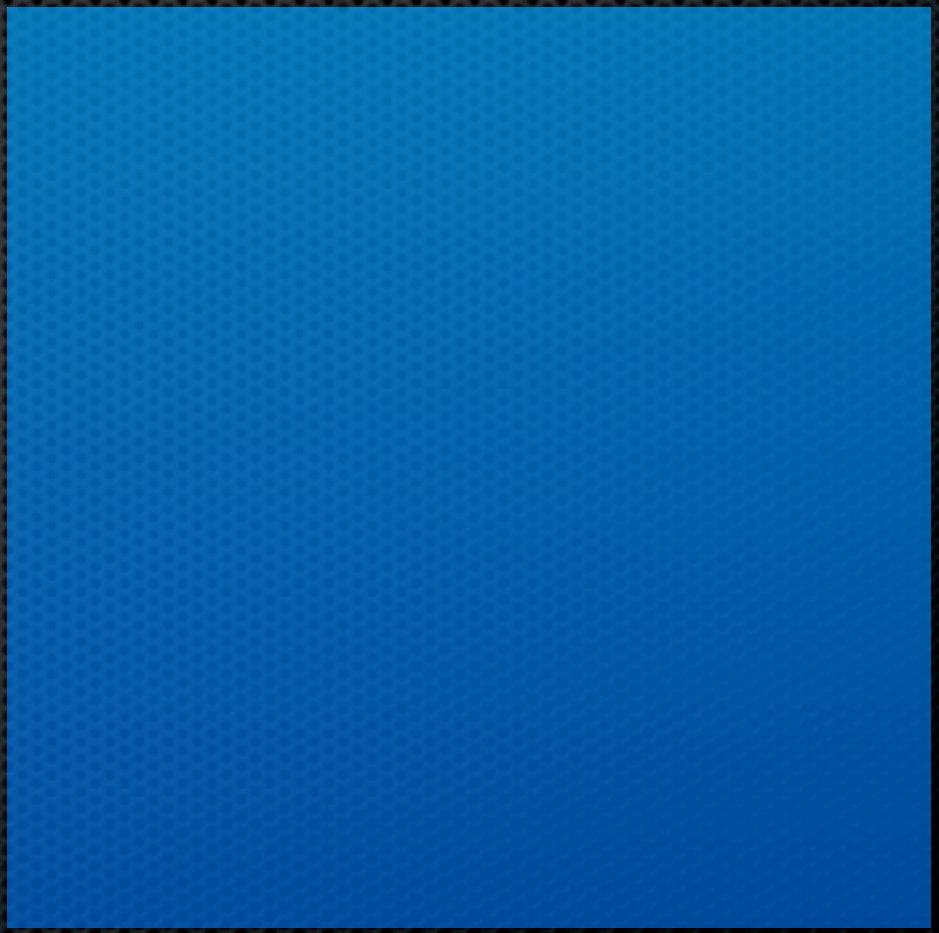
Linda

- Linda is called a coordination language
  - It is both a language and a kind of library
  - It allows one to easily implement replicated workers with a "shared" bag
- Basic idea: processes share a *tuple space*
- Tuple: ("tag", values)

Linda Primitives

- `out ("tag", values)`
- `in ("tag", vars/values)`
- `rd ("tag", vars/values)`
- `eval ("tag", f(...))`

Tuple Space



out (PB, Mark, 223-4879)

out (PB, Kyle, 654-3923)

rd (PB, Mark, u)

rd (PB, v, 654-3923)

rd (PB, x, y)

in (PB, x, z)

eval (“worker”, worker())

u=

v=

x=

y=

z=

Linda Primitives

- `out ("tag", values)`
- `in ("tag", vars/values)`
- `rd ("tag", vars/values)`
- `eval ("tag", f(...))`

out (PB, Mark, 223-4879)

out (PB, Kyle, 654-3923)

rd (PB, Mark, u)

rd (PB, v, 654-3923)

rd (PB, x, y)

in (PB, x, z)

eval (“worker”, worker())

u=

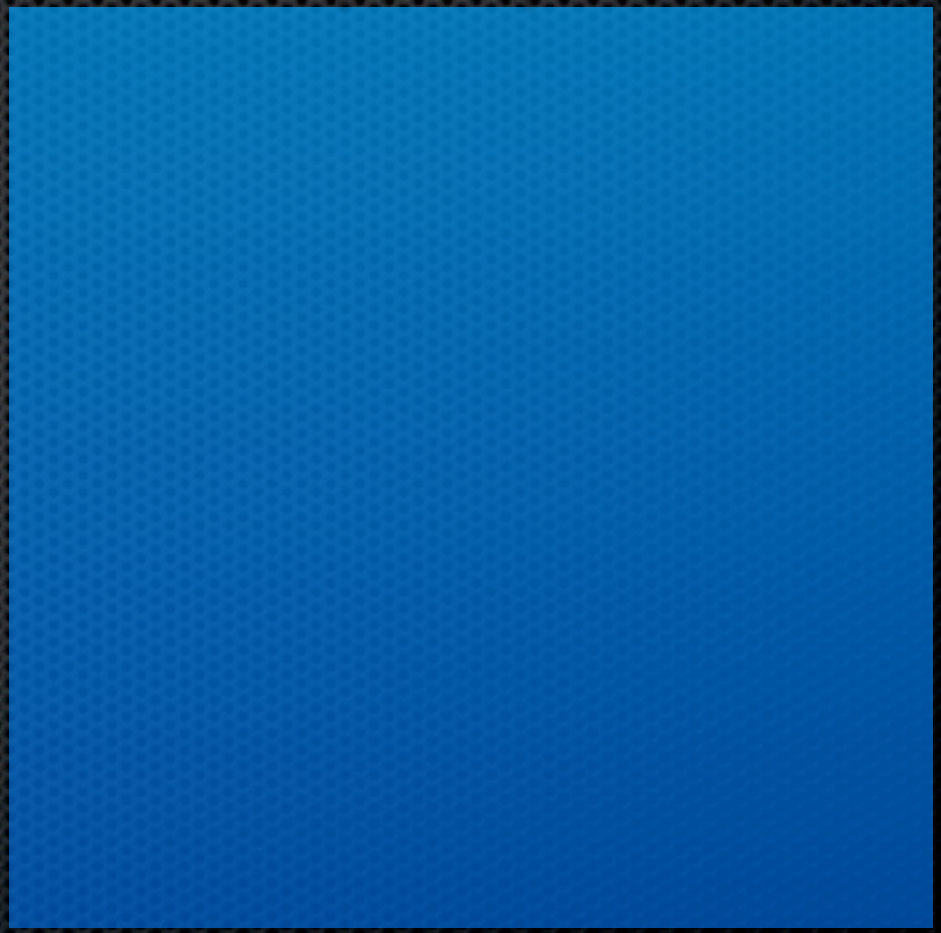
v=

x=

y=

z=

Tuple Space



Linda Primitives

- `out ("tag", values)`
- `in ("tag", vars/values)`
- `rd ("tag", vars/values)`
- `eval ("tag", f(...))`

out (PB, Mark, 223-4879)

out (PB, Kyle, 654-3923)

rd (PB, Mark, u)

rd (PB, v, 654-3923)

rd (PB, x, y)

in (PB, x, z)

eval (“worker”, worker())

u=

v=

x=

y=

z=

Tuple Space

(PB, Mark,223-4879)

Linda Primitives

- `out ("tag", values)`
- `in ("tag", vars/values)`
- `rd ("tag", vars/values)`
- `eval ("tag", f(...))`

out (PB, Mark, 223-4879)

out (PB, Kyle, 654-3923)

rd (PB, Mark, u)

rd (PB, v, 654-3923)

rd (PB, x, y)

in (PB, x, z)

eval (“worker”, worker())

u=

v=

x=

y=

z=

Tuple Space

(PB, Mark,223-4879)

Linda Primitives

- `out ("tag", values)`
- `in ("tag", vars/values)`
- `rd ("tag", vars/values)`
- `eval ("tag", f(...))`

out (PB, Mark, 223-4879)

out (PB, Kyle, 654-3923)

rd (PB, Mark, u)

rd (PB, v, 654-3923)

rd (PB, x, y)

in (PB, x, z)

eval (“worker”, worker())

u=

v=

x=

y=

z=

Tuple Space

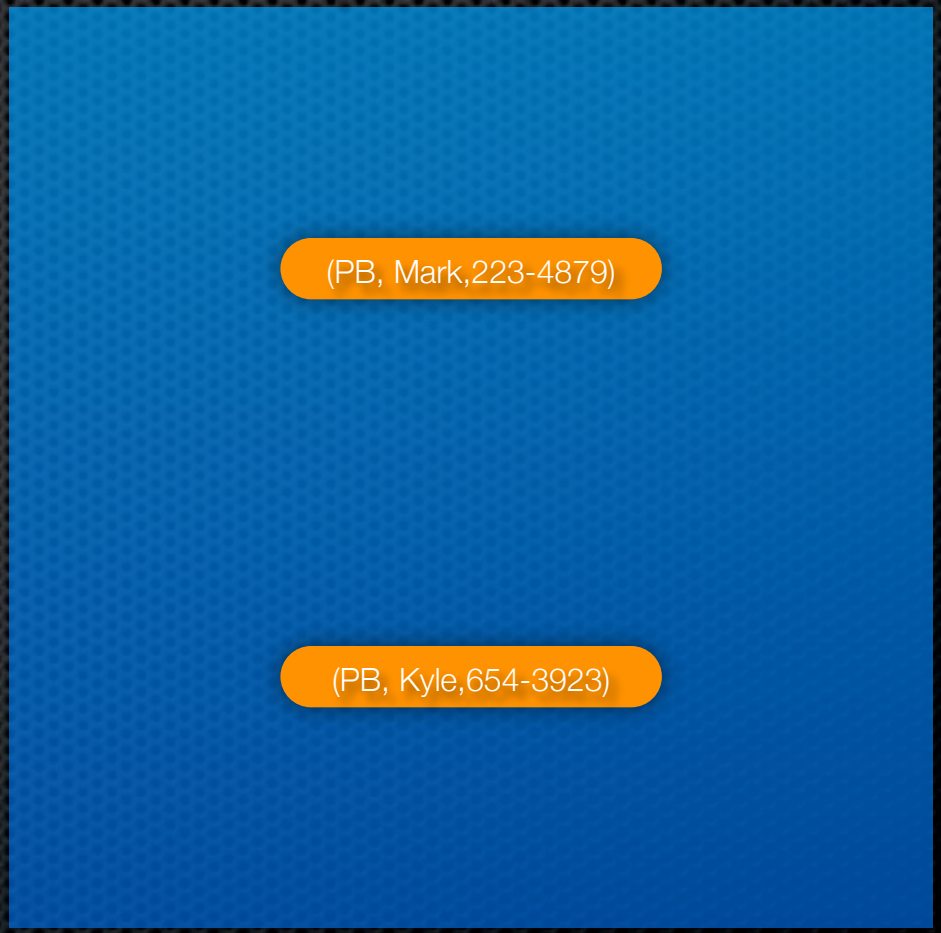
(PB, Mark,223-4879)

(PB, Kyle,654-3923)

Linda Primitives

- `out ("tag", values)`
- `in ("tag", vars/values)`
- `rd ("tag", vars/values)`
- `eval ("tag", f(...))`

Tuple Space



out (PB, Mark, 223-4879)

out (PB, Kyle, 654-3923)

rd (PB, Mark, u)

rd (PB, v, 654-3923)

rd (PB, x, y)

in (PB, x, z)

eval (“worker”, worker())

u=

v=

x=

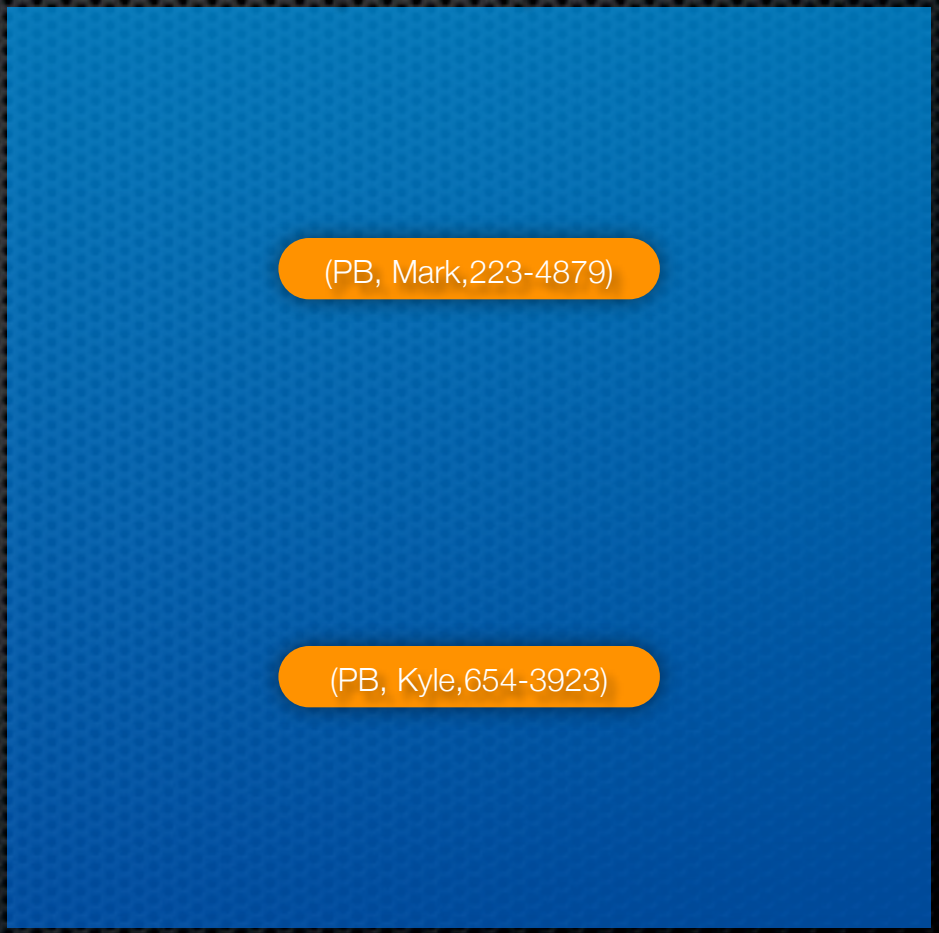
y=

z=

Linda Primitives

- `out ("tag", values)`
- `in ("tag", vars/values)`
- `rd ("tag", vars/values)`
- `eval ("tag", f(...))`

Tuple Space



out (PB, Mark, 223-4879)

out (PB, Kyle, 654-3923)

rd (PB, Mark, u)

rd (PB, v, 654-3923)

rd (PB, x, y)

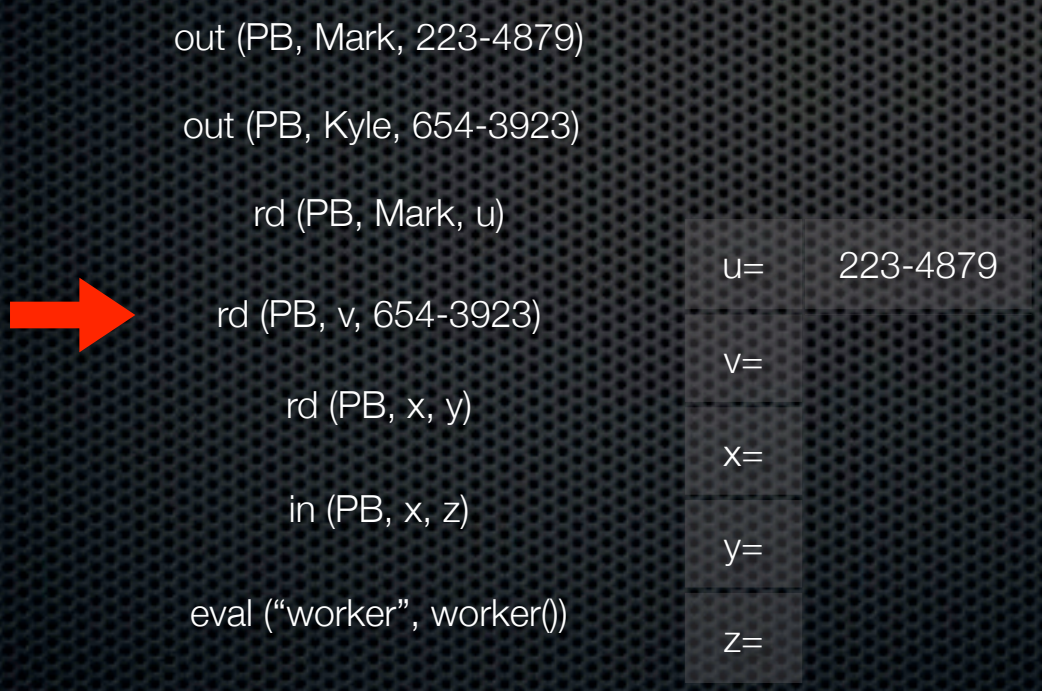
in (PB, x, z)

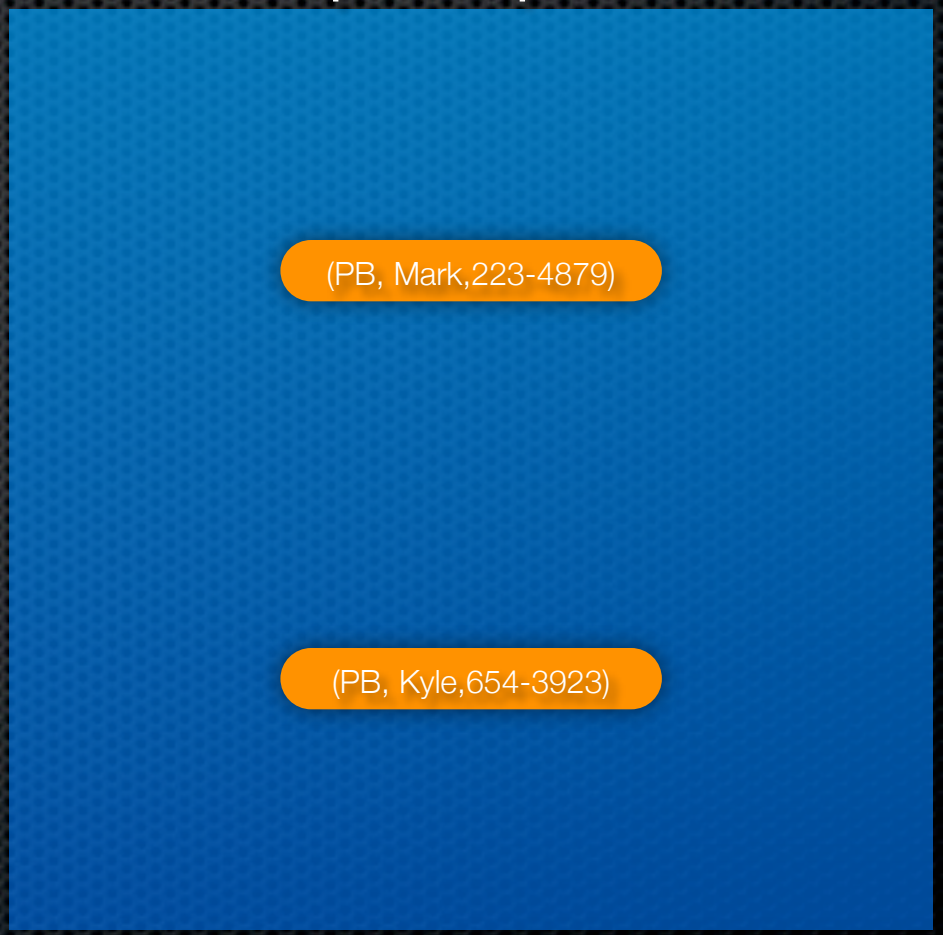
eval (“worker”, worker())

u=	223-4879
v=	
x=	
y=	
z=	

Linda Primitives

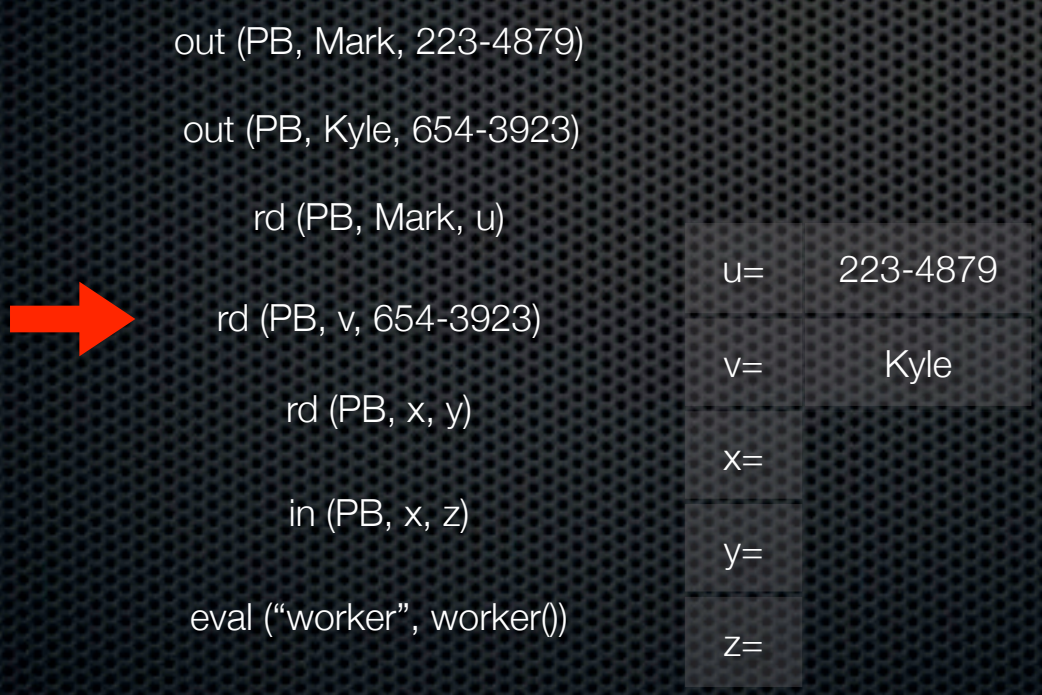
- `out ("tag", values)`
- `in ("tag", vars/values)`
- `rd ("tag", vars/values)`
- `eval ("tag", f(...))`

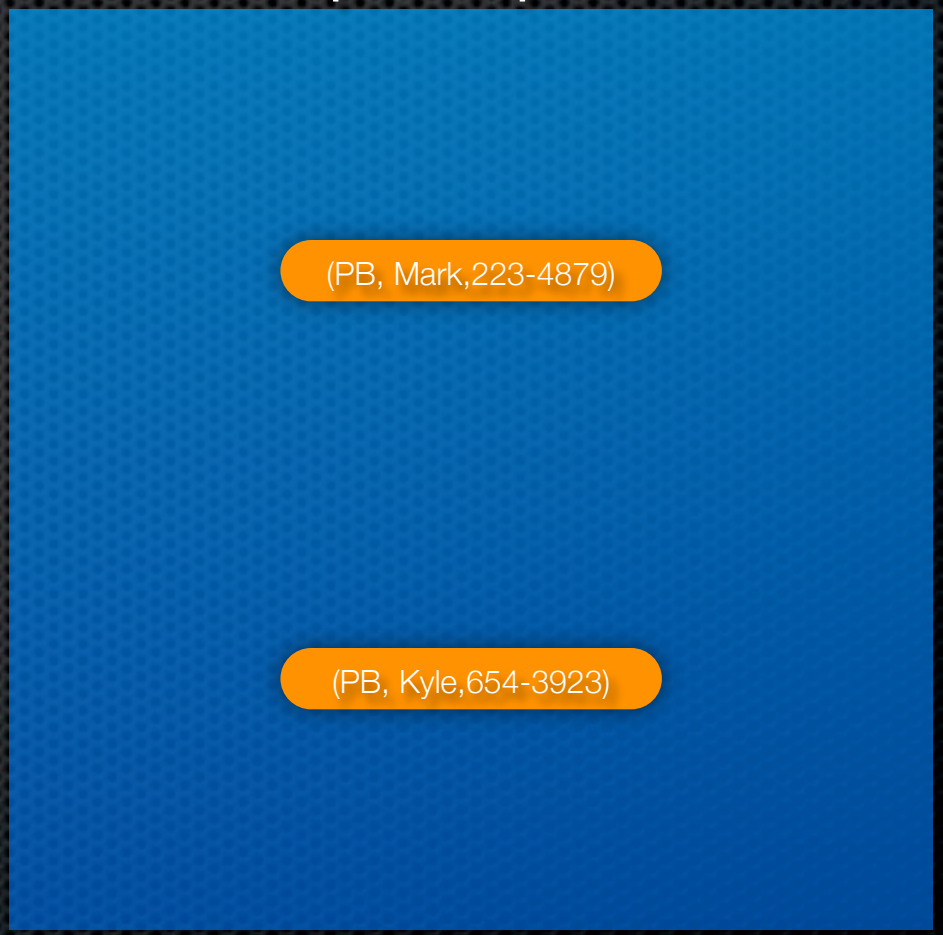




Linda Primitives

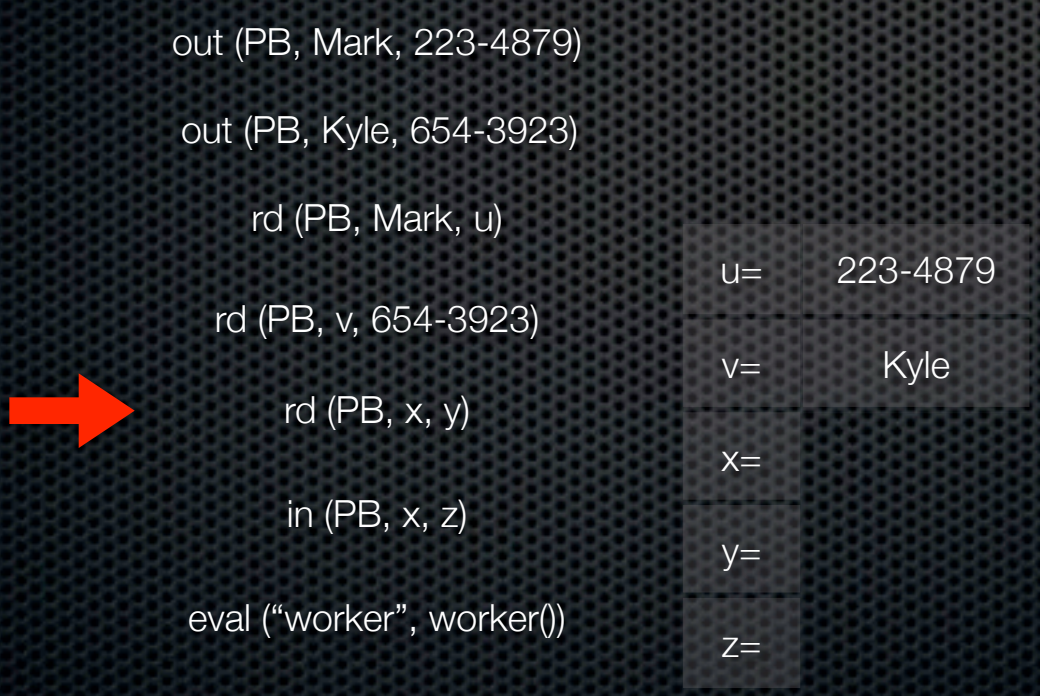
- `out ("tag", values)`
- `in ("tag", vars/values)`
- `rd ("tag", vars/values)`
- `eval ("tag", f(...))`

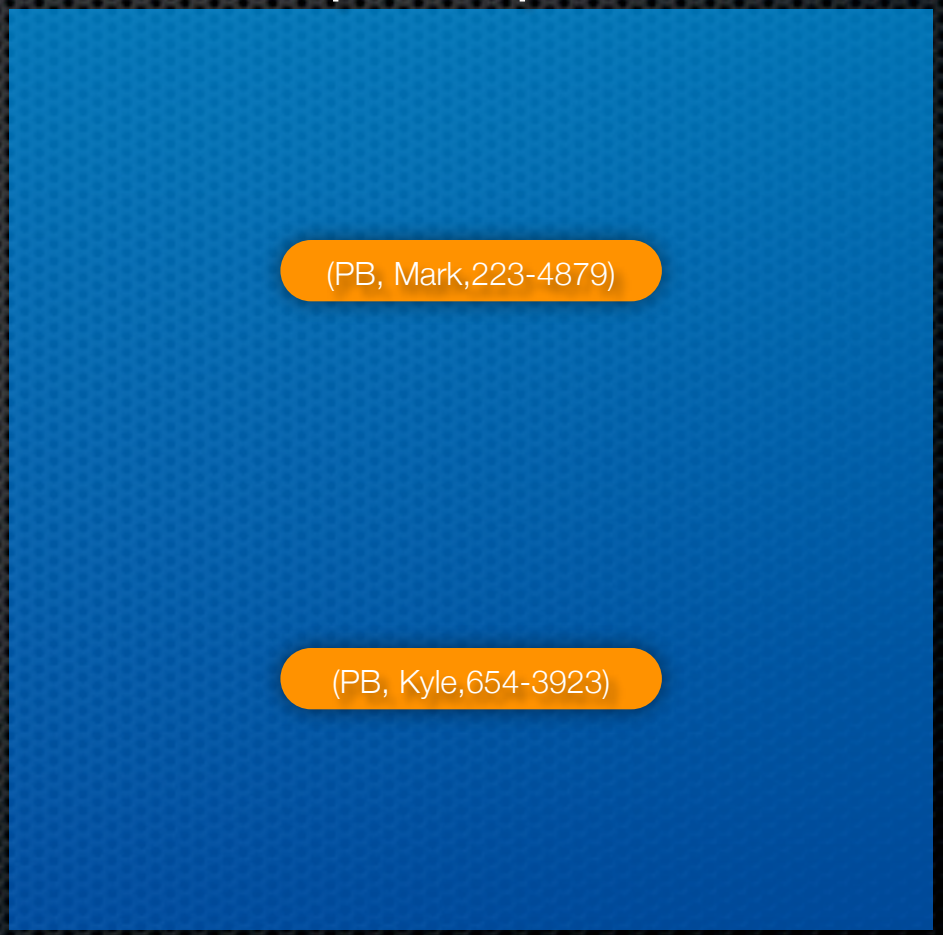




Linda Primitives

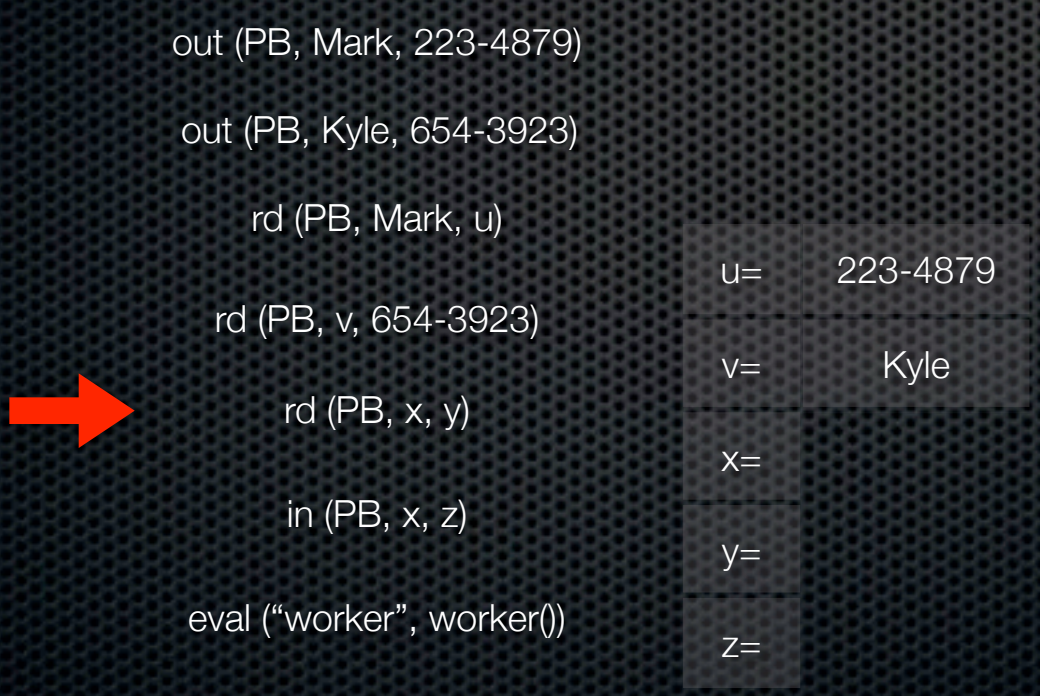
- `out ("tag", values)`
- `in ("tag", vars/values)`
- `rd ("tag", vars/values)`
- `eval ("tag", f(...))`

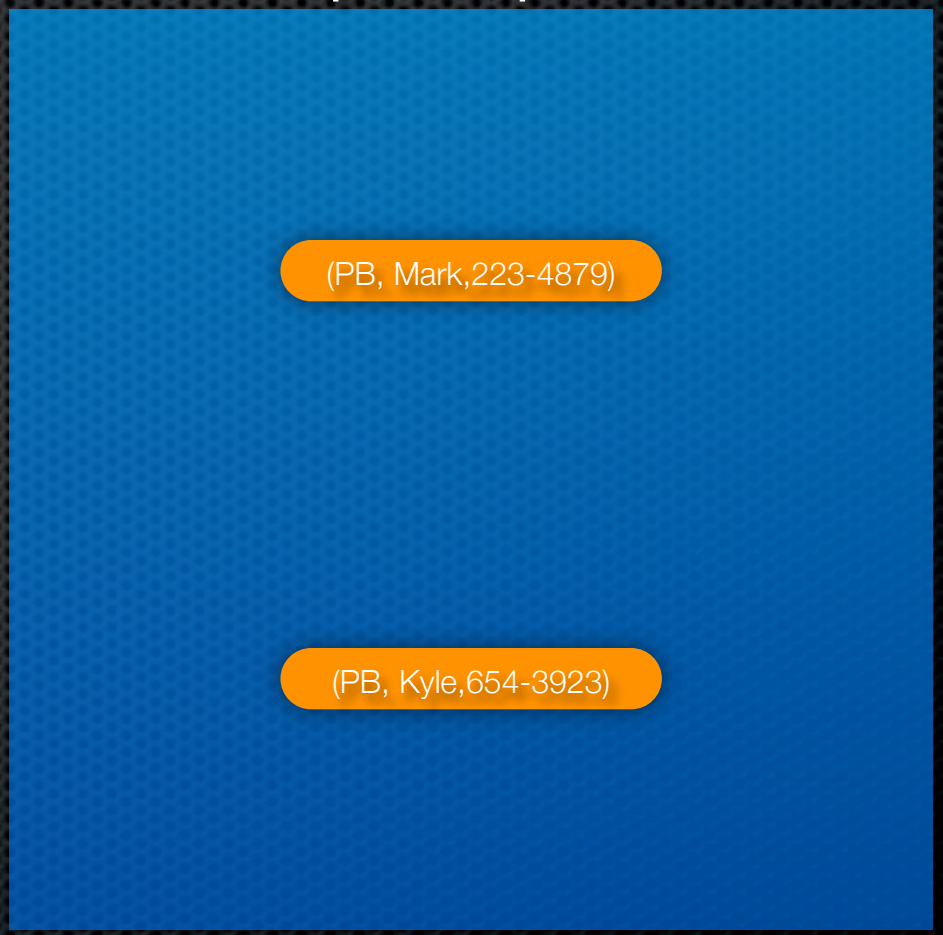




Linda Primitives

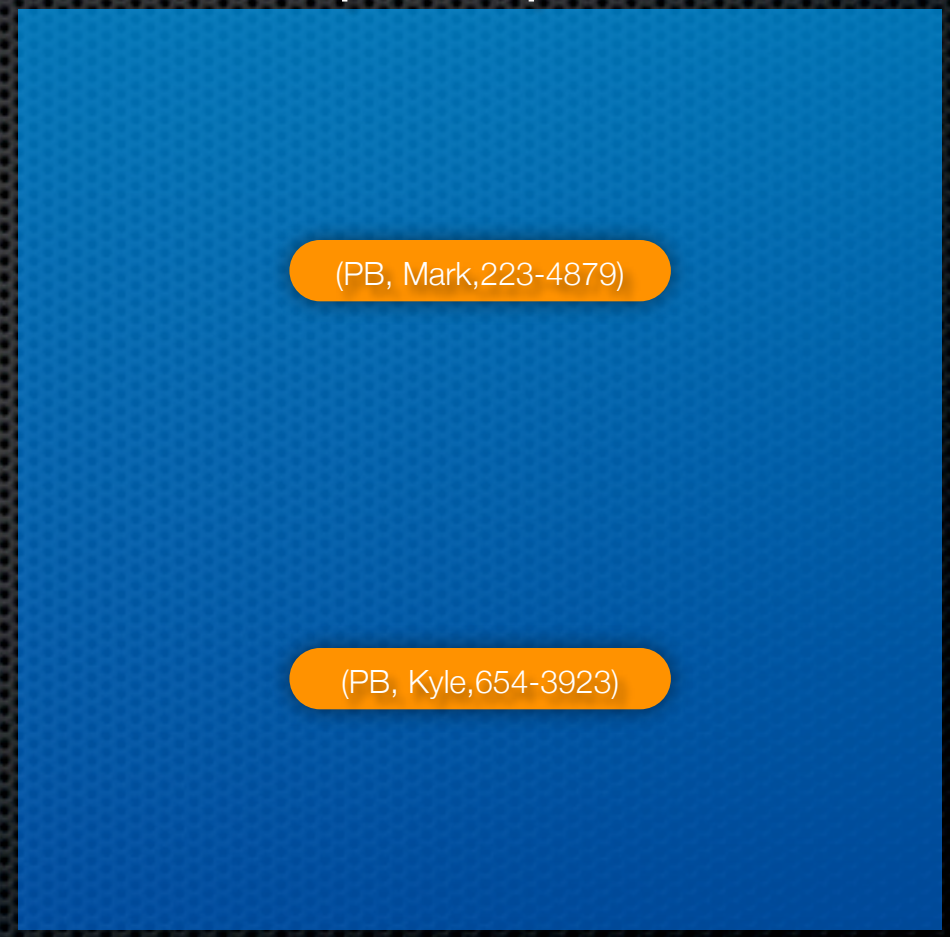
- `out ("tag", values)`
- `in ("tag", vars/values)`
- `rd ("tag", vars/values)`
- `eval ("tag", f(...))`





Linda Primitives

- `out ("tag", values)`
- `in ("tag", vars/values)`
- `rd ("tag", vars/values)`
- `eval ("tag", f(...))`



out (PB, Mark, 223-4879)

out (PB, Kyle, 654-3923)

rd (PB, Mark, u)

rd (PB, v, 654-3923)

rd (PB, x, y)

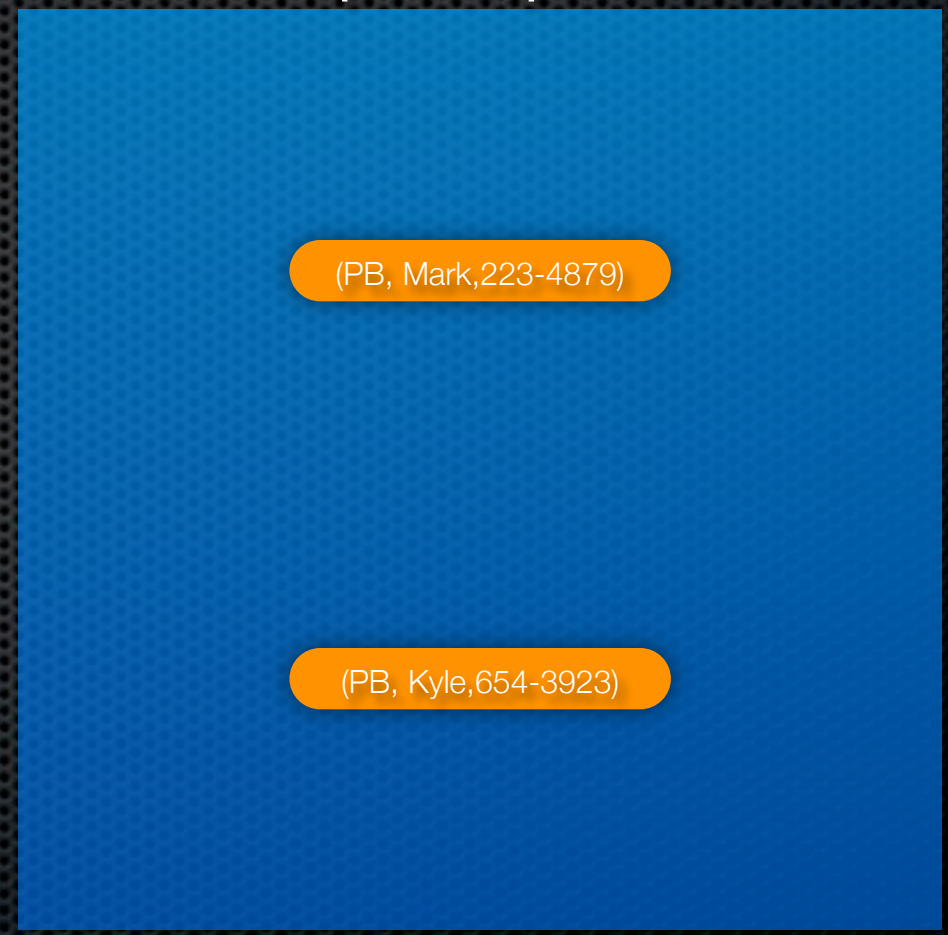
in (PB, x, z)

eval (“worker”, worker())

u=	223-4879
v=	Kyle
x=	Mark
y=	223-4879
z=	

Linda Primitives

- `out ("tag", values)`
- `in ("tag", vars/values)`
- `rd ("tag", vars/values)`
- `eval ("tag", f(...))`



out (PB, Mark, 223-4879)

out (PB, Kyle, 654-3923)

rd (PB, Mark, u)

rd (PB, v, 654-3923)

rd (PB, x, y)

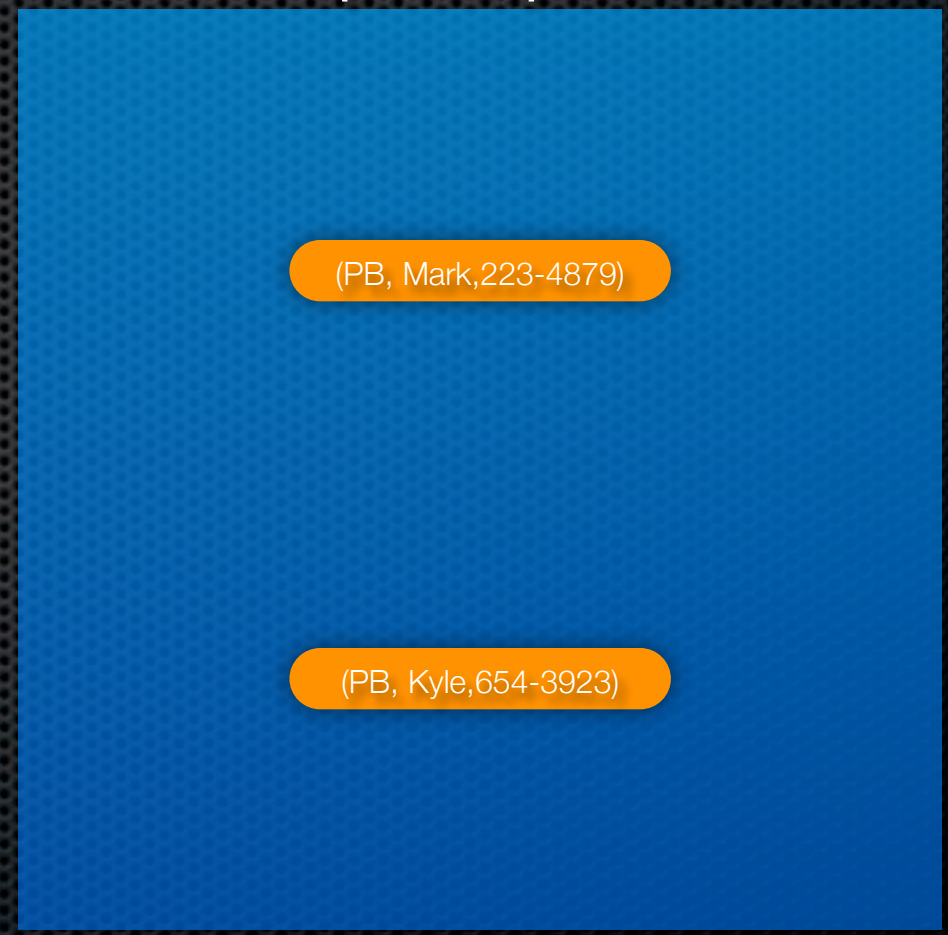
in (PB, x, z)

eval (“worker”, worker())

u=	223-4879
v=	Kyle
x=	Mark
y=	223-4879
z=	

Linda Primitives

- `out ("tag", values)`
- `in ("tag", vars/values)`
- `rd ("tag", vars/values)`
- `eval ("tag", f(...))`



out (PB, Mark, 223-4879)

out (PB, Kyle, 654-3923)

rd (PB, Mark, u)

rd (PB, v, 654-3923)

rd (PB, x, y)

in (PB, x, z)

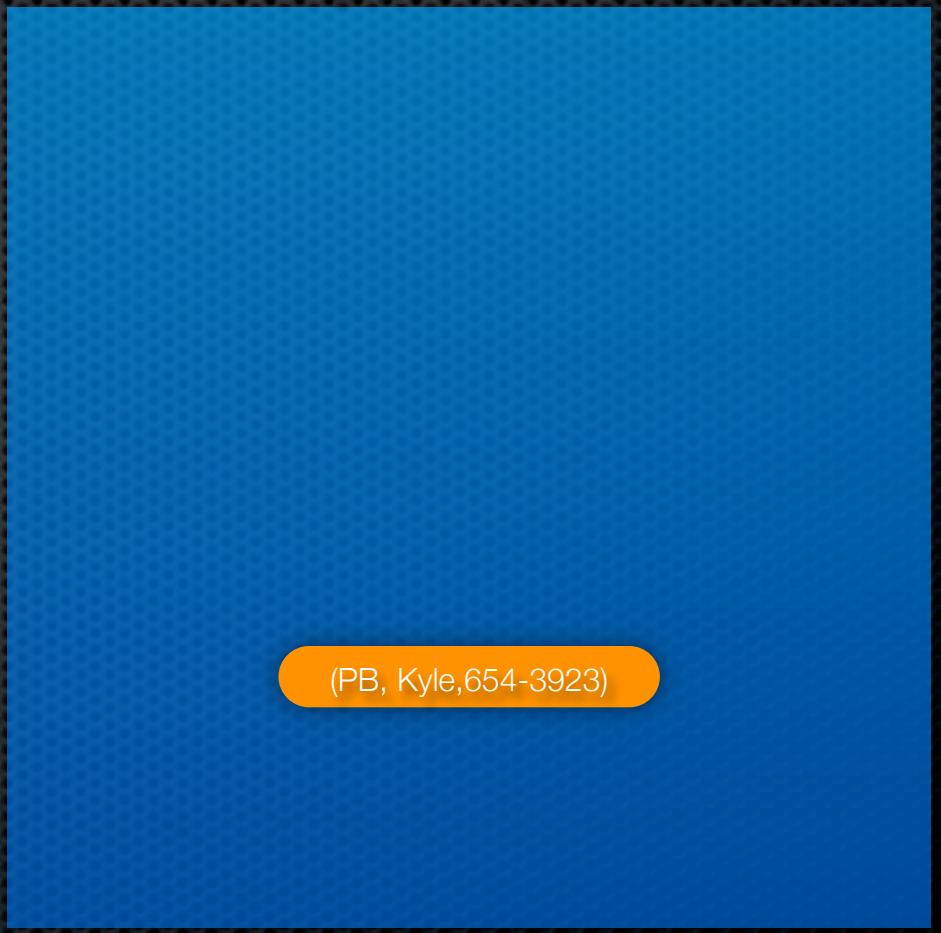
eval (“worker”, worker())

u=	223-4879
v=	Kyle
x=	Mark
y=	223-4879
z=	

Linda Primitives

- `out ("tag", values)`
- `in ("tag", vars/values)`
- `rd ("tag", vars/values)`
- `eval ("tag", f(...))`

Tuple Space



out (PB, Mark, 223-4879)

out (PB, Kyle, 654-3923)

rd (PB, Mark, u)

rd (PB, v, 654-3923)

rd (PB, x, y)

in (PB, x, z)

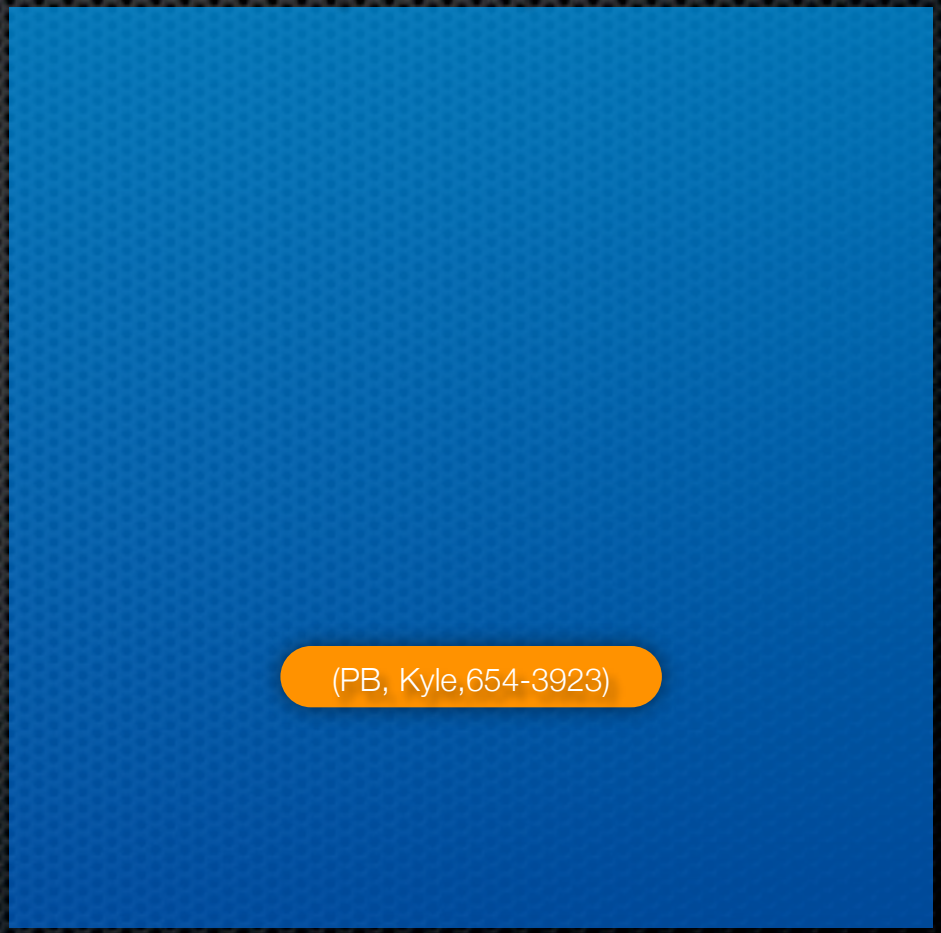
eval (“worker”, worker())

u=	223-4879
v=	Kyle
x=	Mark
y=	223-4879
z=	223-4879

Linda Primitives

- `out ("tag", values)`
- `in ("tag", vars/values)`
- `rd ("tag", vars/values)`
- `eval ("tag", f(...))`

Tuple Space



out (PB, Mark, 223-4879)

out (PB, Kyle, 654-3923)

rd (PB, Mark, u)

rd (PB, v, 654-3923)

rd (PB, x, y)

in (PB, x, z)

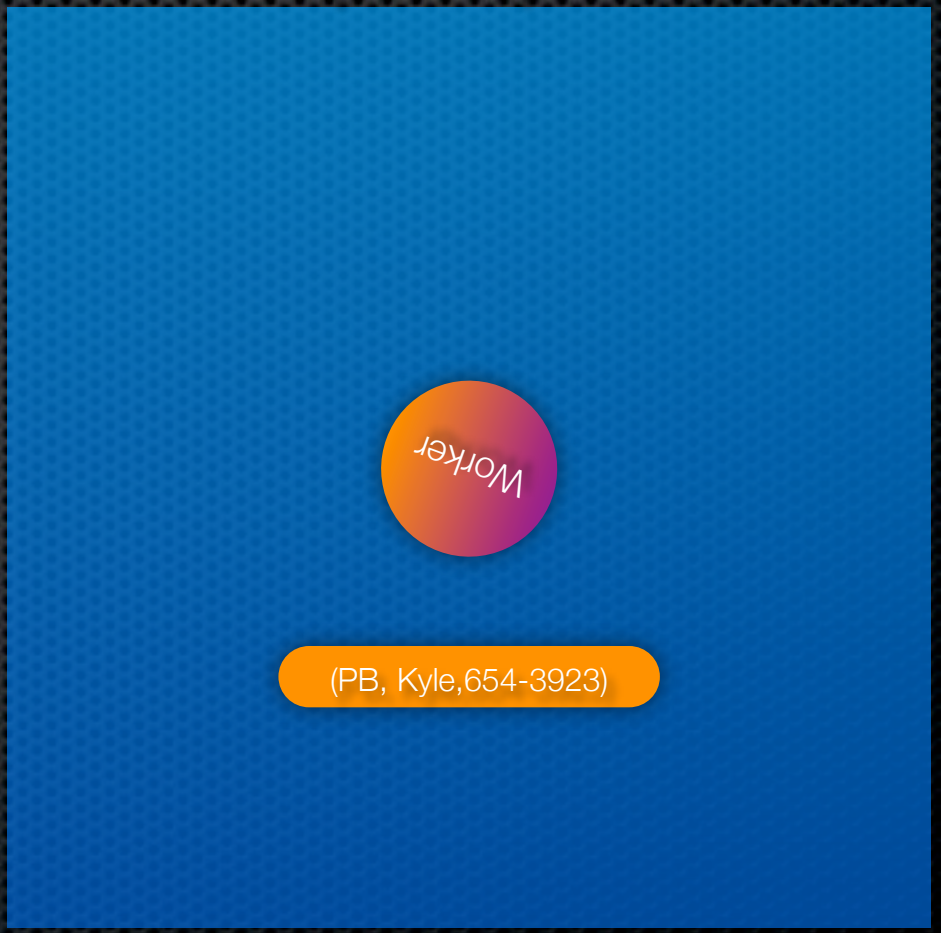
eval (“worker”, worker())

u=	223-4879
v=	Kyle
x=	Mark
y=	223-4879
z=	223-4879

Linda Primitives

- `out ("tag", values)`
- `in ("tag", vars/values)`
- `rd ("tag", vars/values)`
- `eval ("tag", f(...))`

Tuple Space



out (PB, Mark, 223-4879)

out (PB, Kyle, 654-3923)

rd (PB, Mark, u)

rd (PB, v, 654-3923)

rd (PB, x, y)

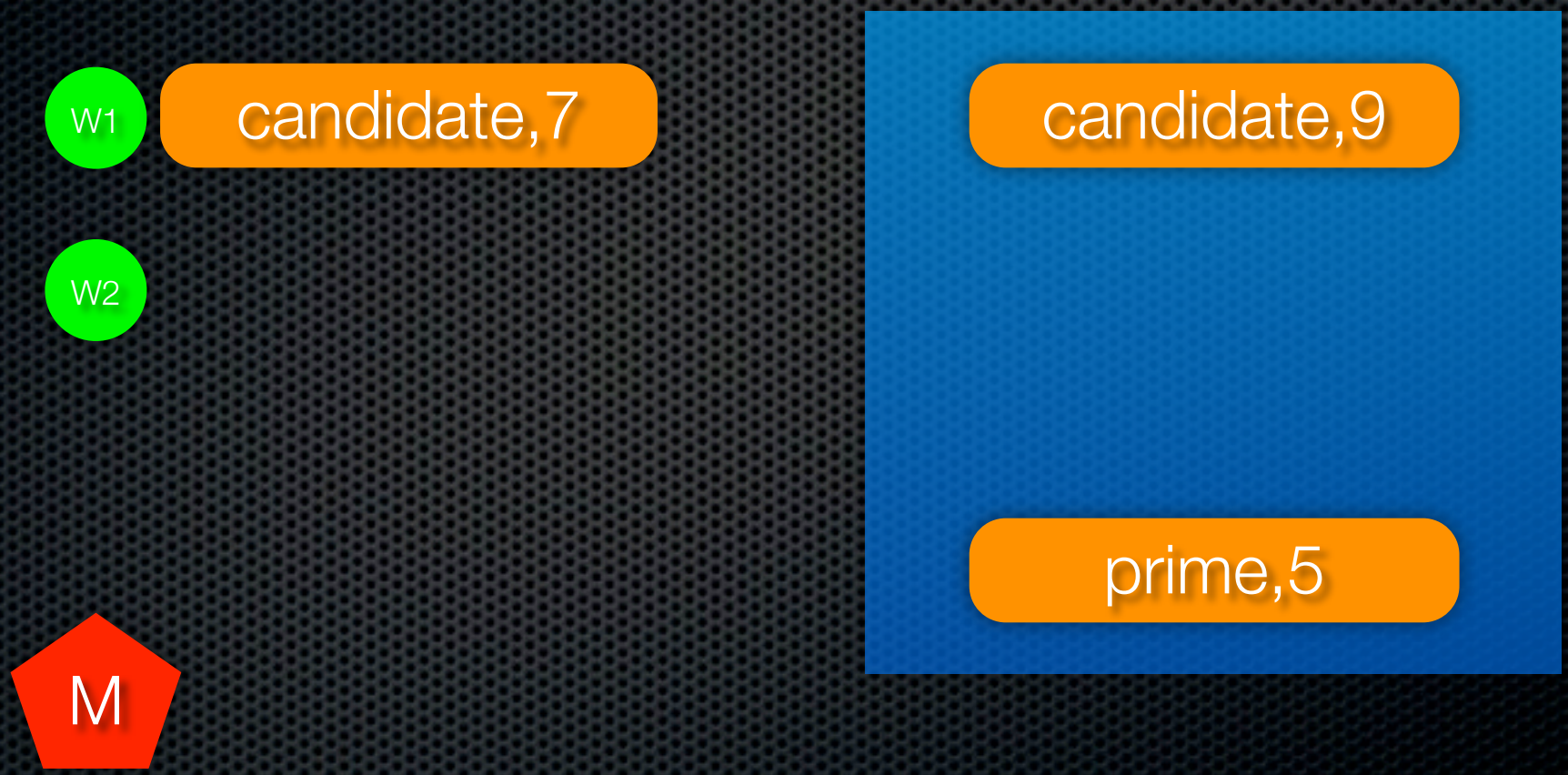
in (PB, x, z)

eval (“worker”, worker())

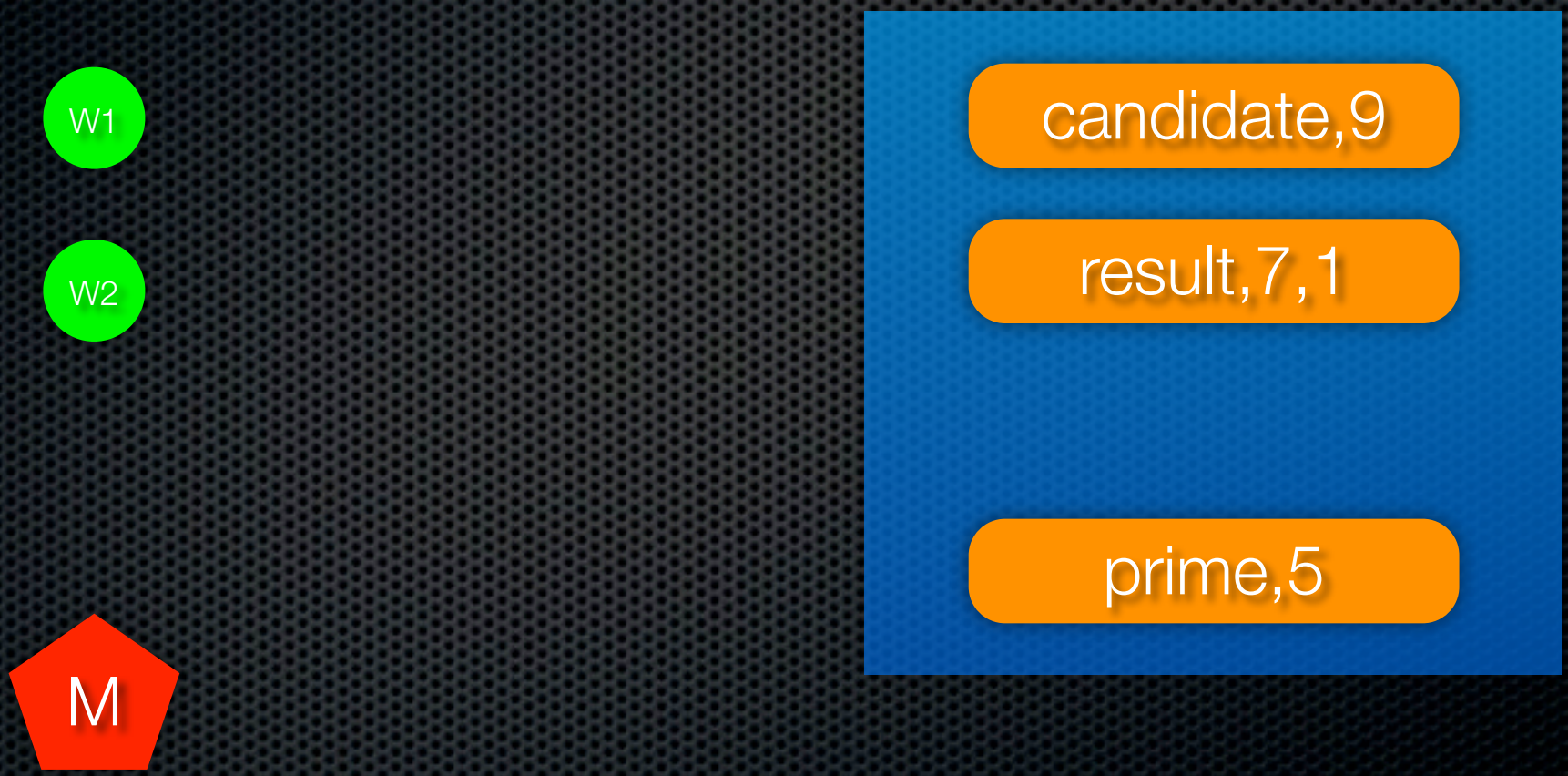
u=	223-4879
v=	Kyle
x=	Mark
y=	223-4879
z=	223-4879

Example: Prime Generation

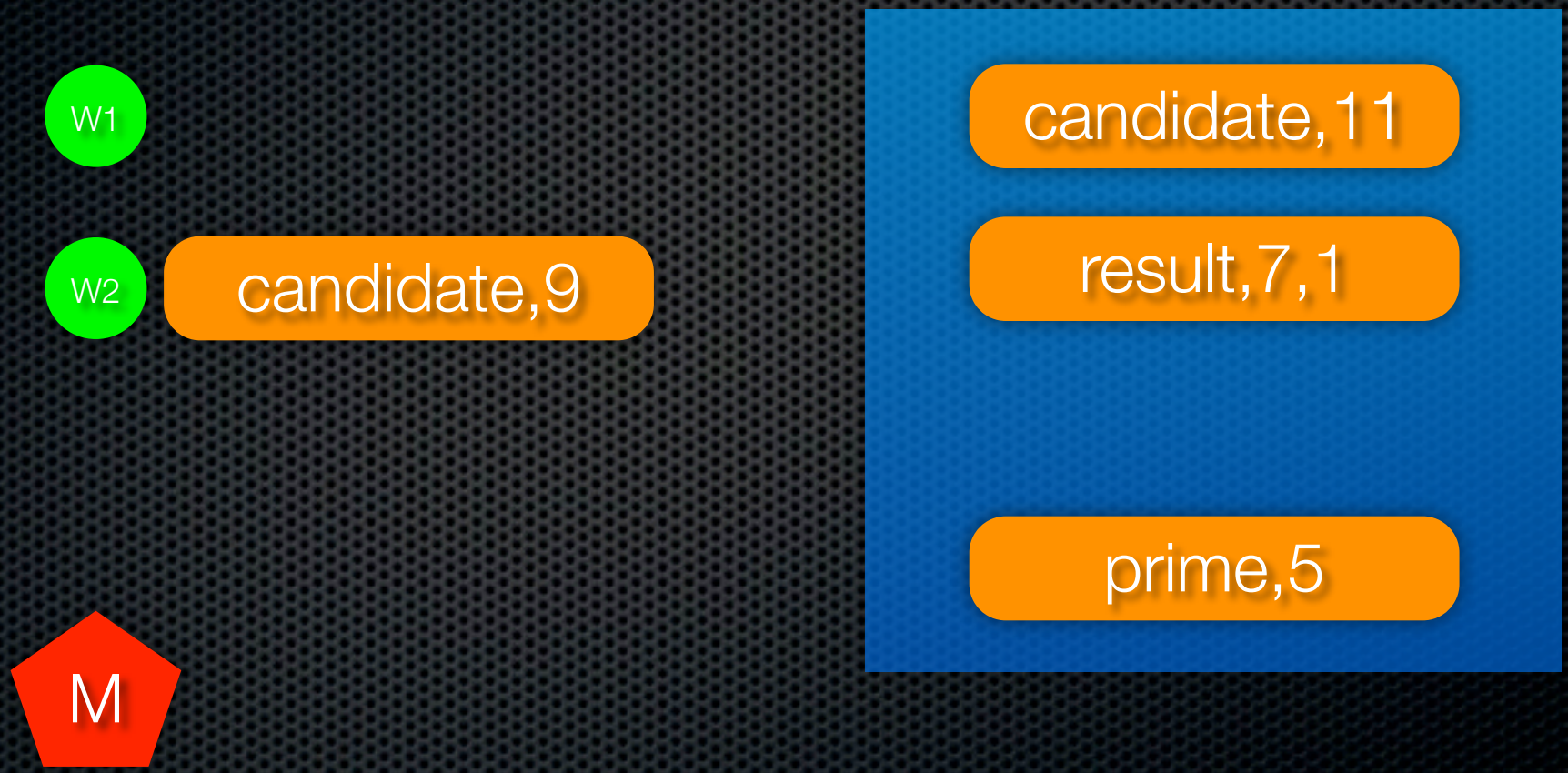
Example: Prime Generation



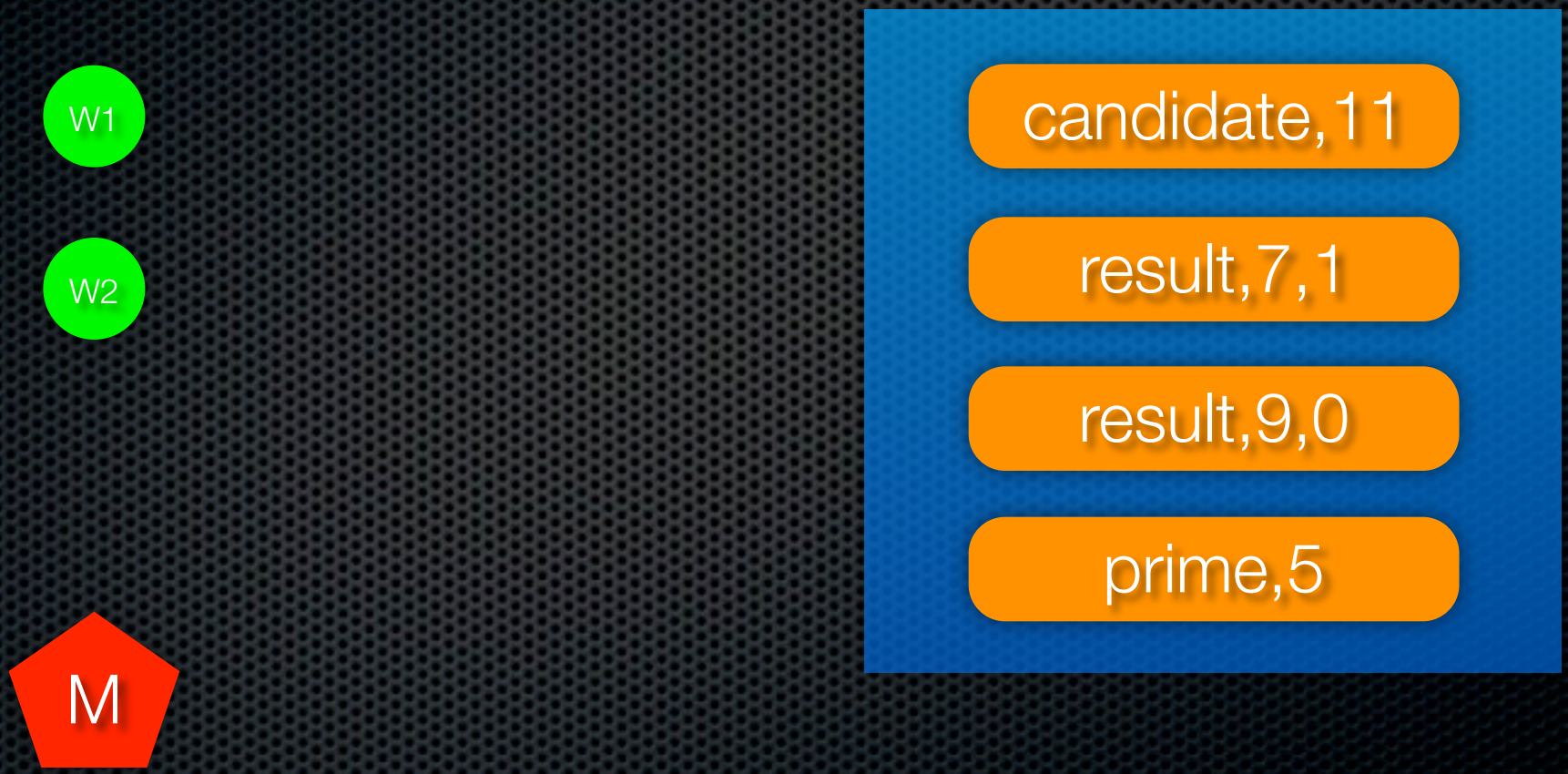
Example: Prime Generation



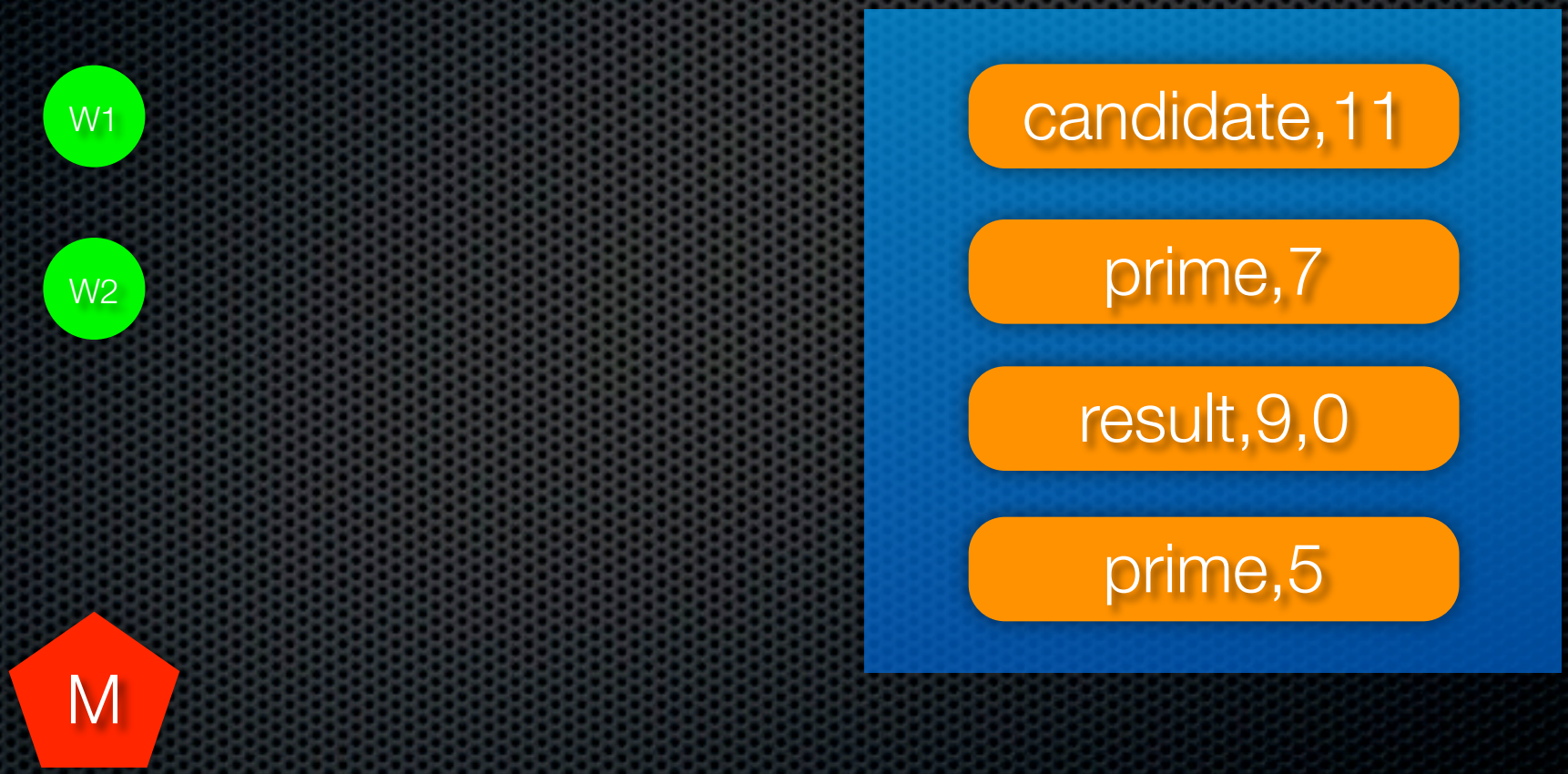
Example: Prime Generation



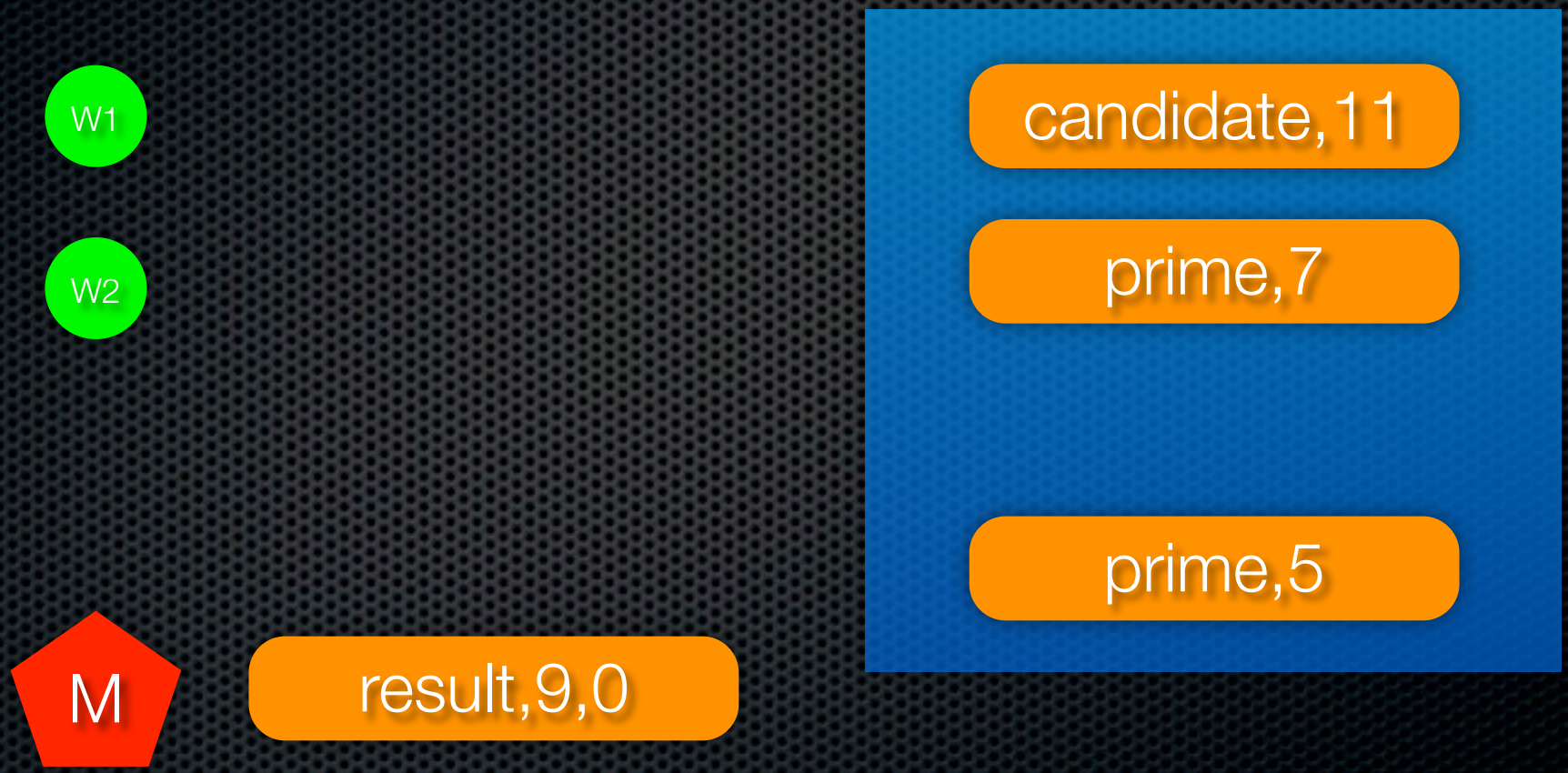
Example: Prime Generation



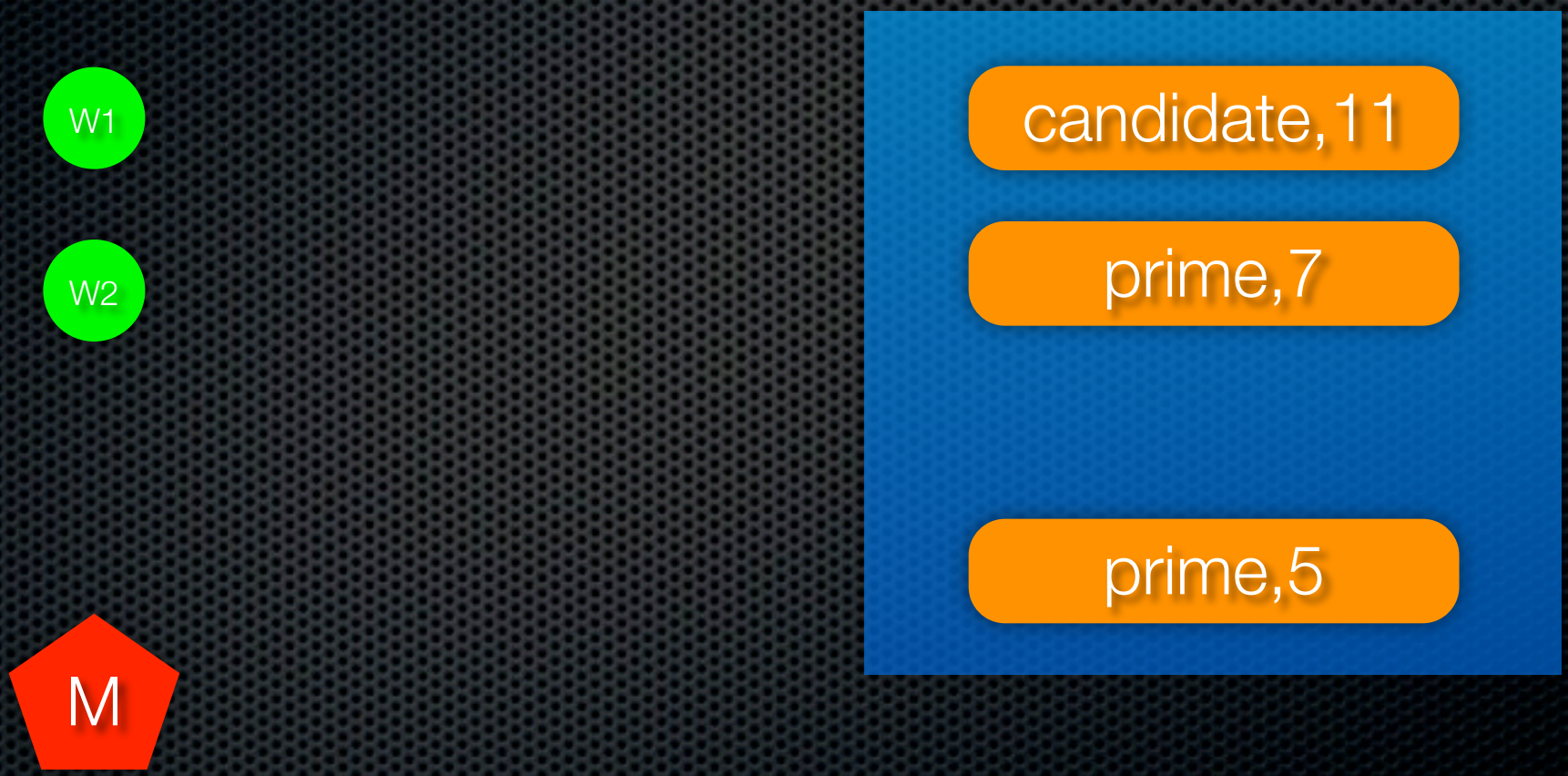
Example: Prime Generation



Example: Prime Generation



Example: Prime Generation



Prime Number Generation (Cont.)

```
real_main(int argc, char *argv[]) {
  int primes[LIMIT] = {2,3};  /* my table of primes */
  int limit, numWorkers, i, isprime;
  int numPrimes = 2, value = 5;
  limit = atoi(argv[1]);  /* read command line */
  numWorkers = atoi(argv[2]);

  /* create workers and put first candidate in bag */
  for (i = 1; i <= numWorkers; i++)
    EVAL("worker", worker());
  OUT("candidate", value);

  /* get results from workers in increasing order */
  while (numPrimes < limit) {
    IN("result", value, ?isprime);
    if (isprime) {   /* put value in table and TS */
      primes[numPrimes] = value;
      OUT("prime", numPrimes, value);
      numPrimes++;
    }
    value = value + 2;
  }
  /* tell workers to quit, then print the primes */
  OUT("stop");
  for (i = 0; i < limit; i++)
    printf("%d\n", primes[i]);
```

Example: Prime Generation (Cont.)

```
#include "linda.h"
#define LIMIT 1000     /* upper bound for limit */

void worker() {
  int primes[LIMIT] = {2,3};  /* table of primes */
  int numPrimes = 1, i, candidate, isprime;

  /* repeatedly get candidates and check them */
  while(true) {
    if (RDP("stop"))  /* check for termination */
      return;
    IN("candidate", ?candidate);    /* get candidate */
    OUT("candidate", candidate+2);  /* output next one */
    i = 0; isprime = 1;
    while (primes[i]*primes[i] <= candidate) {
      if (candidate%primes[i] == 0) { /* not prime */
        isprime = 0; break;
      }
      i++;
      if (i > numPrimes) {  /* need another prime */
        numPrimes++;
        RD("prime", numPrimes, ?primes[numPrimes]);
      }
    }
    /* tell manager the result */
    OUT("result", candidate, isprime);
  }
}
```