12. Architectural Styles for Concurrency

Overview

- What is Software Architecture?
- Three-layered application architecture
- Flow architectures
  - ☞ Active Prime Sieve
- Blackboard architectures
  - ☞ Fibonacci with Linda

Sources

- M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.
- F. Buschmann, et al., *Pattern-Oriented Software Architecture — A System of Patterns*, John Wiley, 1996.
- D. Lea, *Concurrent Programming in Java — Design principles and Patterns*, The Java Series, Addison-Wesley, 1996.
- N. Carriero and D. Gelernter, *How to Write Parallel Programs: a First Course*, MIT Press, Cambridge, 1990.

Software Architecture

A Software Architecture defines a system in terms of computational components and interactions amongst those components.

An Architectural Style defines a family of systems in terms of a pattern of structural organization.

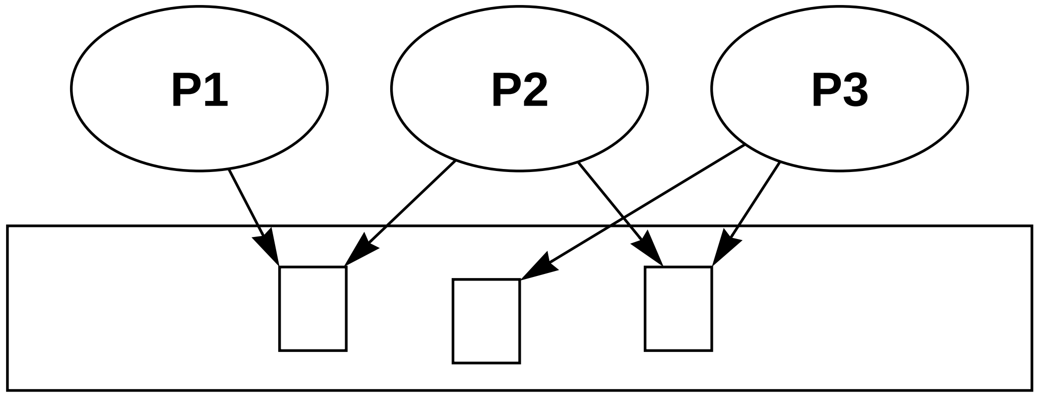
— cf. Shaw & Garlan, Software Architecture, pp. 3, 19

Architectural style

Architectural styles typically entail four kinds of properties:

- A *vocabulary* of design elements
  - ☞ e.g., “pipes”, “filters”, “sources”, and “sinks”
- A set of *configuration rules* that *constrain* compositions
  - ☞ e.g., pipes and filters must alternate in a linear sequence
- A *semantic interpretation*
  - ☞ e.g., each filter reads bytes from its input stream and writes bytes to its output stream
- A set of *analyses* that can be performed
  - ☞ e.g., if filters are “well-behaved”, no deadlock can occur, and all filters can progress in tandem

Communication Styles



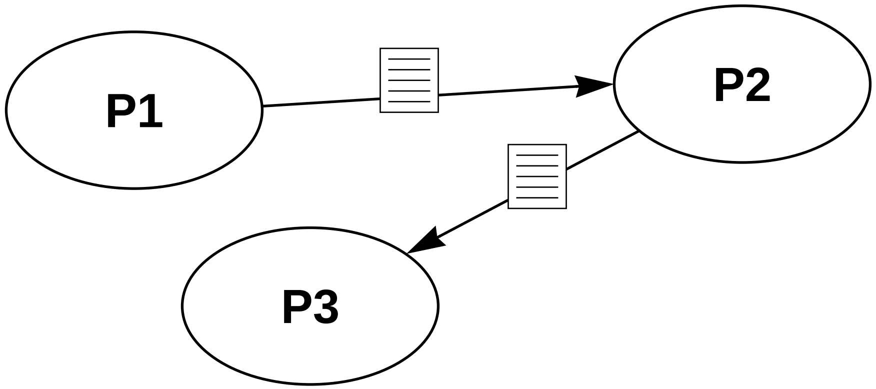
Shared Variables

Processes communicate *indirectly*.

Explicit synchronization mechanisms are needed.

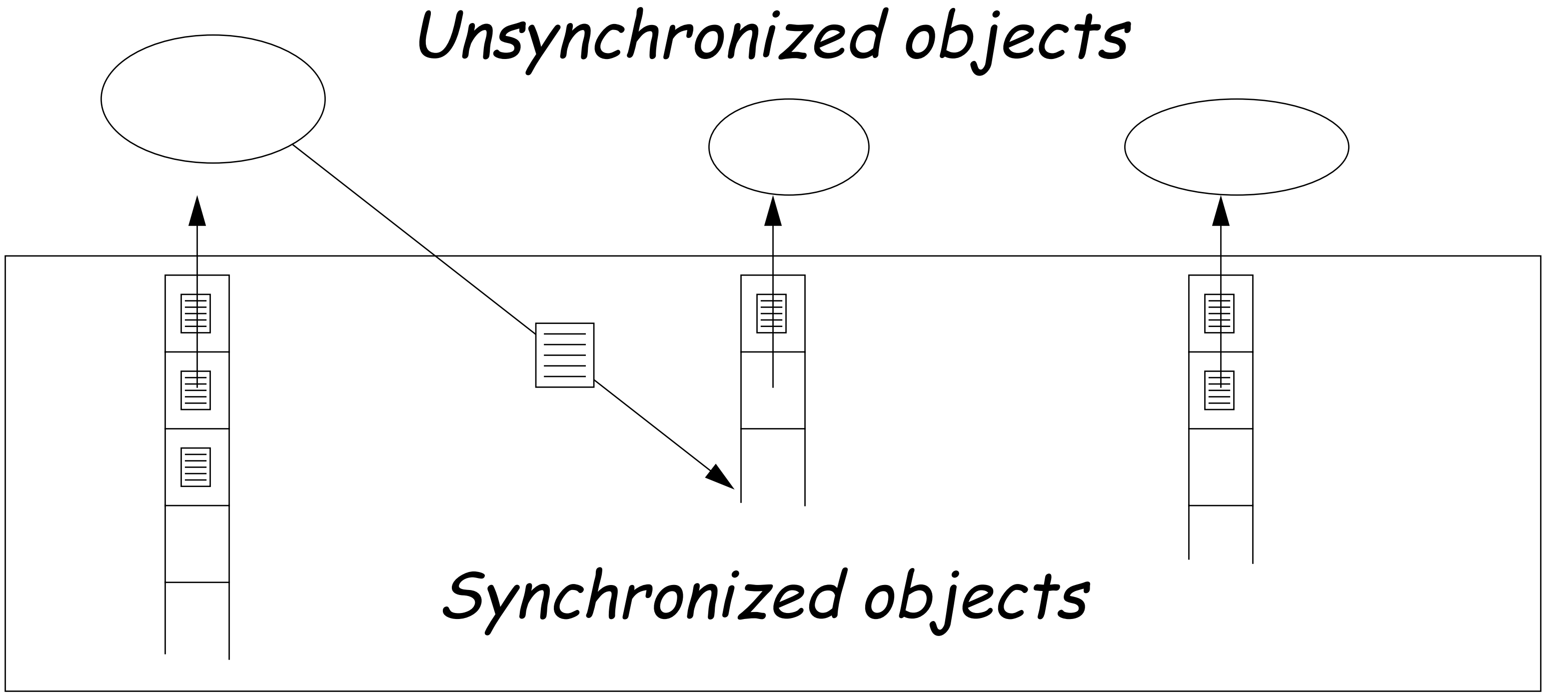
Message-Passing

Communication and synchronization are *combined*.



Simulated Message-Passing

Most concurrency and communication styles can be simulated by one another:



Message-passing can be modelled by associating message queues to each process.

Three-layered Application Architectures

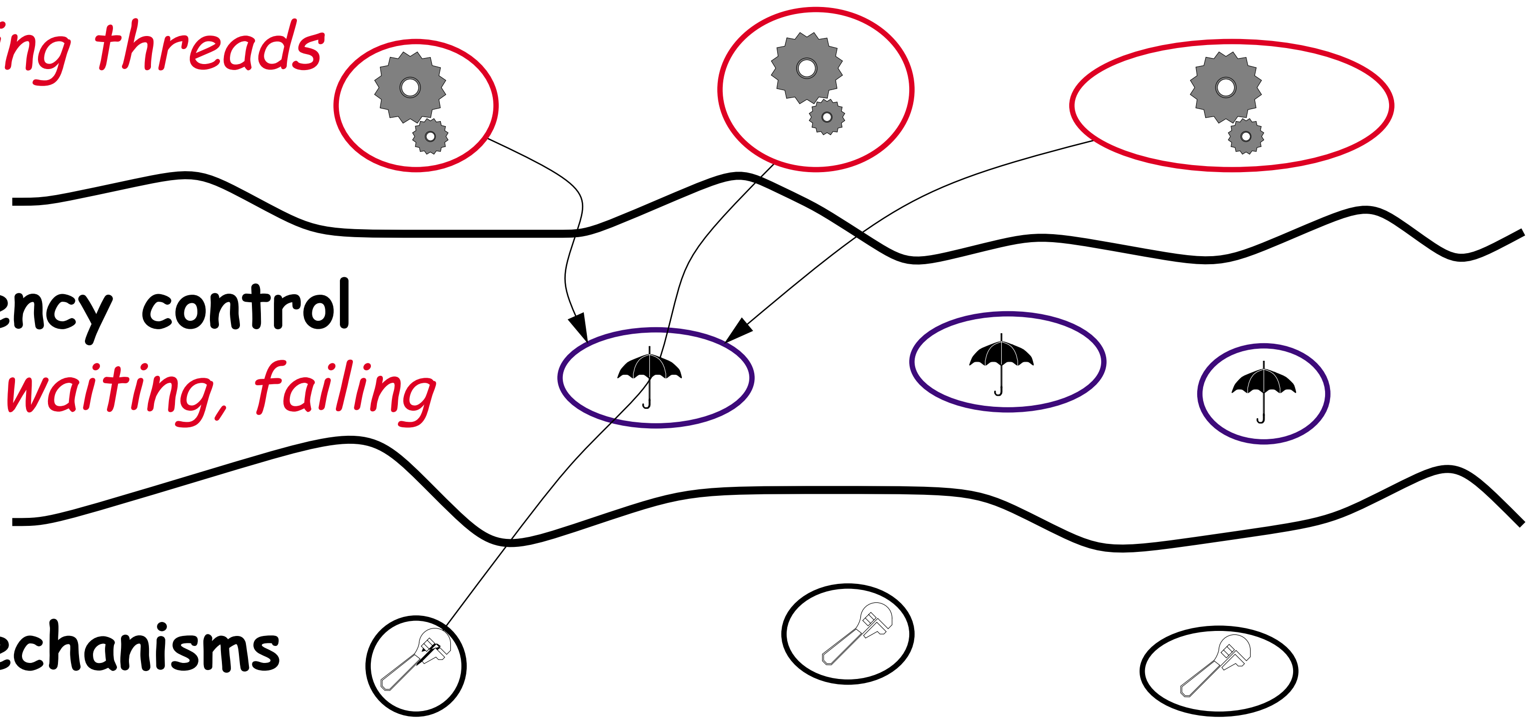
Interaction with external world

Generating threads

Concurrency control

Locking, waiting, failing

Basic mechanisms



This kind of architecture avoids nested monitor problems by restricting concurrency control to a single layer.

Problems with Layered Designs

Hard to extend beyond three layers because:

- Control may depend on *unavailable information*
  - Because it is not safely accessible
  - Because it is not represented (e.g., message history)
- Synchronization *policies* of different layers *may conflict*
  - E.g., nested monitor lockouts
- *Ground* actions may *need to know current policy*
  - E.g., blocking vs. failing

Flow Architectures

Many synchronization problems can be avoided by arranging things so that information only flows in one direction from sources to filters to sinks.

Unix "pipes and filters": Processes are connected in a linear sequence.

Control systems: events are picked up by sensors, processed, and generate new events.

Workflow systems: Electronic documents flow through workflow procedures.

Unix Pipes

Unix pipes are *bounded buffers* that *connect producer* and *consumer* processes (*sources*, *sinks* and *filters*):

```
cat file      # send file contents to output stream
| tr -c 'a-zA-Z' '\012' # put each word on one line
| sort        # sort the words
| uniq -c     # count occurrences of each word
| sort -rn    # sort in reverse numerical order
| more        # and display the result
```

Unix Pipes

Processes should *read* from standard input and *write* to standard output streams:

- Misbehaving processes give rise to *"broken pipes"!*

Process creation and *scheduling* are handled by the O/S.

Synchronization is handled implicitly by the I/O system (through buffering).

Flow Stages

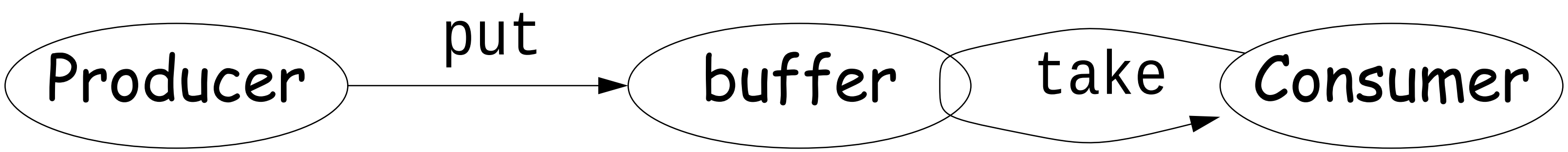
Every flow stage is a *producer* or *consumer* or both:

- *Splitters* (Multiplexers) have multiple successors
  - *Multicasters* clone results to multiple consumers
  - *Routers* distribute results amongst consumers
- *Mergers* (Demultiplexers) have multiple predecessors
  - *Collectors* interleave inputs to a single consumer
  - *Combiners* process multiple input to produce a single result
- *Conduits* have both multiple predecessors and consumers

Flow Policies

Flow can be *pull-based*, *push-based*, or a mixture:

- *Pull-based flow:* Consumers *take* results from Producers
- *Push-based flow:* Producers *put* results to Consumers
- Buffers:
  - *Put-only* buffers (*relays*) connect push-based stages
  - *Take-only* buffers (*pre-fetch buffers*) connect pull-based stages
  - *Put-Take* buffers connect (adapt) push-based stages to pull-based stages



Limiting Flow

Unbounded buffers: If *producers* are *faster* than consumers, buffers may *exhaust* available memory

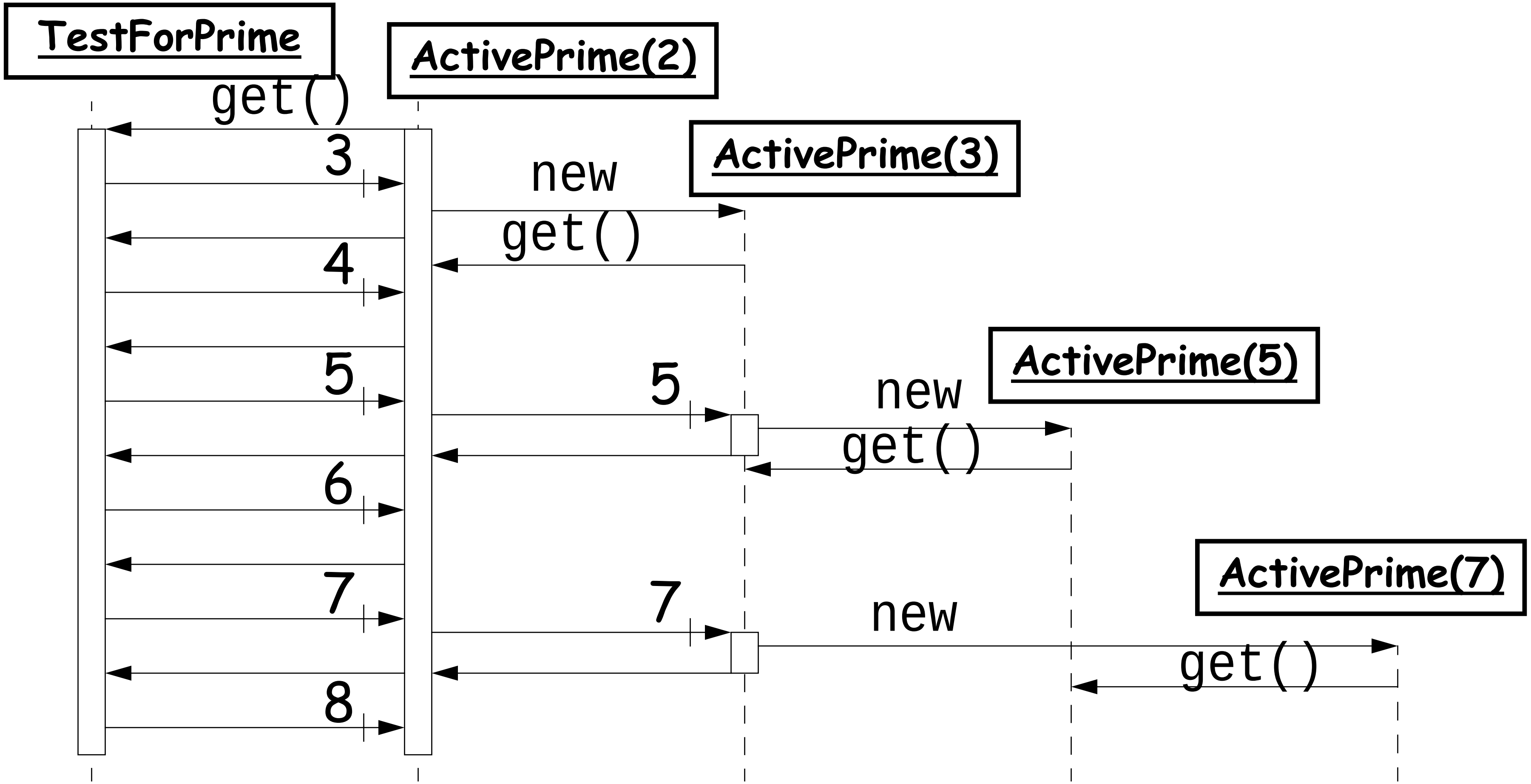
Unbounded threads: Having *too many threads* can exhaust system resources more quickly than unbounded buffers

Bounded buffers: Tend to be either *always full* or *always empty*, depending on relative speed of producers and consumers

Bounded thread pools: *Harder to manage* than bounded buffers

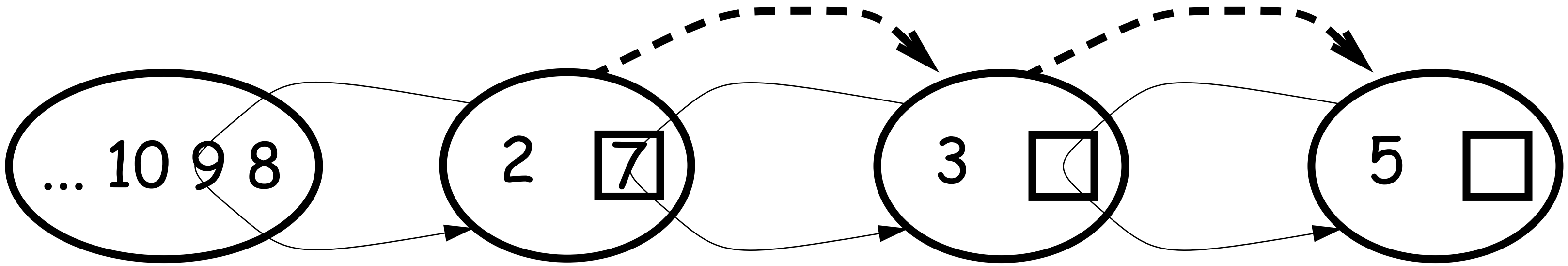
Example: a Pull-based Prime Sieve

Primes are agents that reject non-primes, pass on candidates, or instantiate new prime agents:



Using Put-Take Buffers

Each ActivePrime uses a one-slot buffer to feed values to the next ActivePrime.



The first ActivePrime *holds* the seed value 2, *gets* values from a TestForPrime, and *creates* new ActivePrime instances whenever it detects a prime value.

The PrimeSieve

The main PrimeSieve class creates the initial configuration

```
public class PrimeSieve {
  public static void main(String args[]) {
    genPrimes(1000);
  }
  public static void genPrimes(int n) {
    try {
      ActivePrime firstPrime =
        new ActivePrime(2, new TestForPrime(n));
    } catch (Exception e) { }
  }
}
```

Pull-based integer sources

Active primes get values to test from an `IntSource`*:*

```
interface IntSource { int getInt(); }
class TestForPrime implements IntSource {
  private int nextValue;
  private int maxValue;
  public TestForPrime(int max) {
    this.nextValue = 3; this.maxValue = max;
  }
  public int getInt() {         // not synched!
    if (nextValue < maxValue) { return nextValue++; }
    else { return 0; }
  }
}
```

The ActivePrime Class

ActivePrimes themselves implement IntSource

```
class ActivePrime
    extends Thread implements IntSource {
  private static IntSource lastPrime; // shared
  private int value;             // this prime
  private int square;            // its square
  private IntSource intSrc; // ints to test
  private Slot slot;             // to pass values on
...
```

The ActivePrime Class

```
...
  public ActivePrime(int value, IntSource intSrc)
    throws ActivePrimeFailure
  {
    this.value = value;
    ...
    slot = new Slot();       // NB: private
    lastPrime = this;        // unsynchronized (safe!)
    this.start();            // become active
  }
...
```

It is impossible for primes to be discovered out of order!

The ActivePrime Class ...

```
...
  public int value() {
    return this.value;
  }
  private void putInt(int val) {       // may block
    slot.put()(new Integer(val));
  }
  public int getInt() {                // may block
    return ((Integer) slot.get()).intValue();
  }
...
```

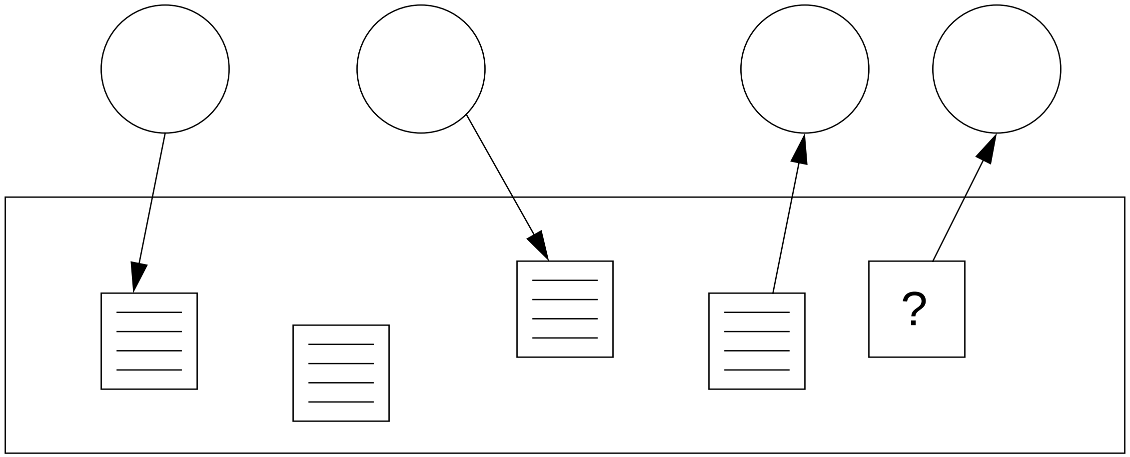
The only synchronization is hidden in the Slot class.

The ActivePrime Class ...

```
public void run() {
  int testValue = intSrc.getInt(); // may block
  while (testValue != 0) {          // stop
    if (this.square > testValue) {  // got a prime
      try {
        new ActivePrime(testValue, lastPrime);
      } catch (Exception e) { break; } // exit loop
    } else if ((testValue % this.value) > 0) {
      this.putInt(testValue);       // may block
    }
    testValue = intSrc.getInt();    // may block
  }
  putInt(0);                        // stop next
}
```

Blackboard Architectures

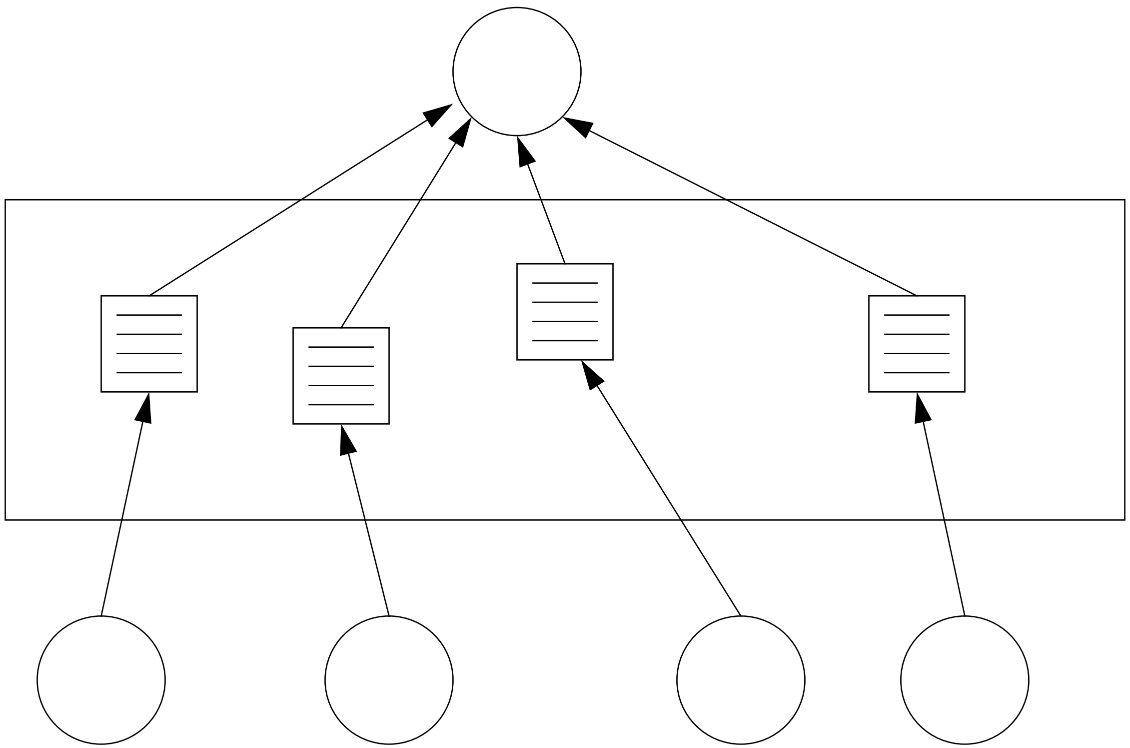
Blackboard architectures put all synchronization in a "coordination medium" where agents can exchange messages.



Agents do not exchange messages directly, but post messages to the blackboard, and retrieve messages either by reading from a specific location (i.e., a *channel*), or by posing a query (i.e., a *pattern* to match).

Result Parallelism

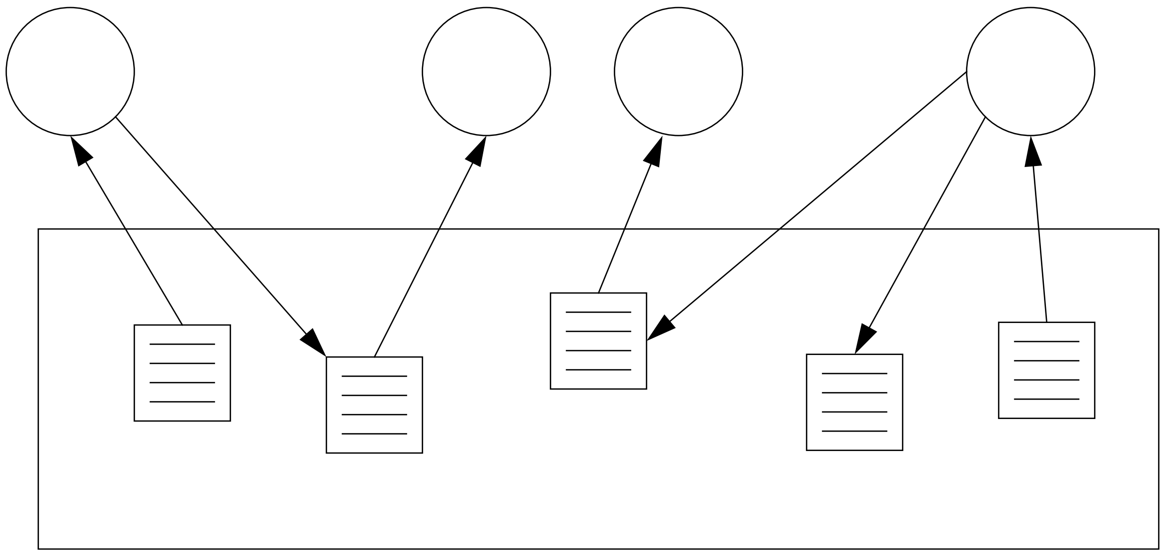
Result parallelism is a blackboard architectural style in which *workers* produce *parts* of a more complex whole.



Workers may be arranged hierarchically ...

Agenda Parallelism

Agenda parallelism is a blackboard style in which workers *retrieve tasks* to perform from a blackboard, and may *generate* new tasks to perform.

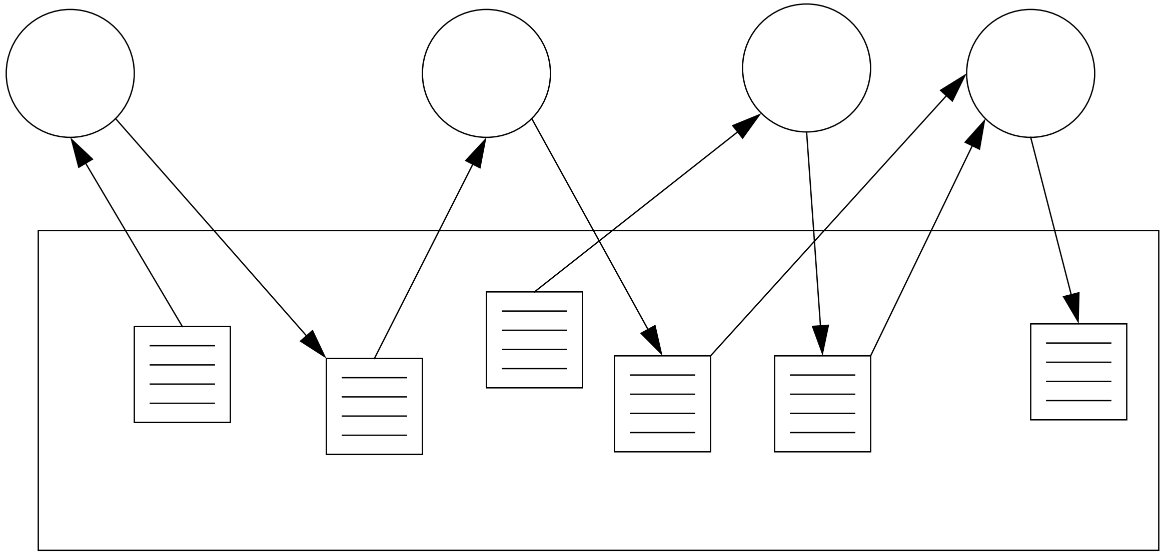


Workers repeatedly retrieve tasks until everything is done.

Workers are typically able to perform *arbitrary tasks*.

Specialist Parallelism

Specialist parallelism is a style in which each worker is *specialized* to perform a particular task.



Specialist designs are equivalent to message-passing, and are often organized as *flow architectures*, with each specialist producing results for the next specialist to consume.

Linda

Linda is a *coordination medium*, with associated primitives for coordinating concurrent processes, that can be *added to an existing programming language*.

The coordination medium is a *tuple-space*, which can contain:

- *data tuples* — tuples of primitives vales (numbers, strings ...)
- *active tuples* — expressions which are evaluated and eventually turn into data tuples

Linda primitives

Linda's coordination primitives are:

`out(T)`	*output* a tuple T to the medium (non-blocking) e.g., `out("employee", "pingu", 35000)`
`in(S)`	*destructively input* a tuple matching S (blocking) e.g., `in("employee", "pingu", ?salary)`
`rd(S)`	*non-destructively input* a tuple (blocking)
`inp(S)` `rdp(S)`	*try* to input a tuple *report success* or failure (non-blocking)
`eval(E)`	evaluate E in a *new process* leave the result in the tuple space

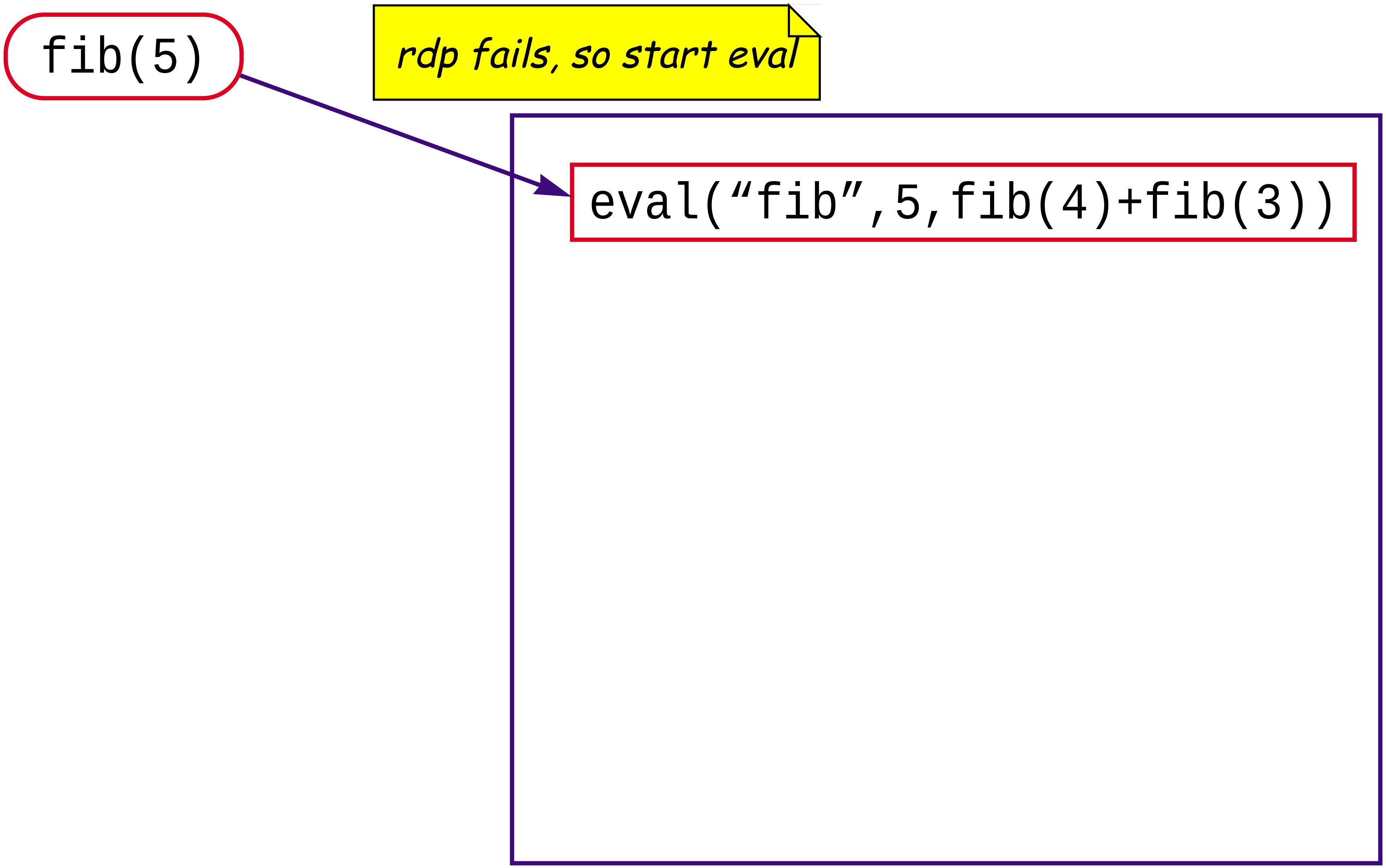
Example: Fibonacci

A (convoluted) way of computing Fibonacci numbers with Linda:

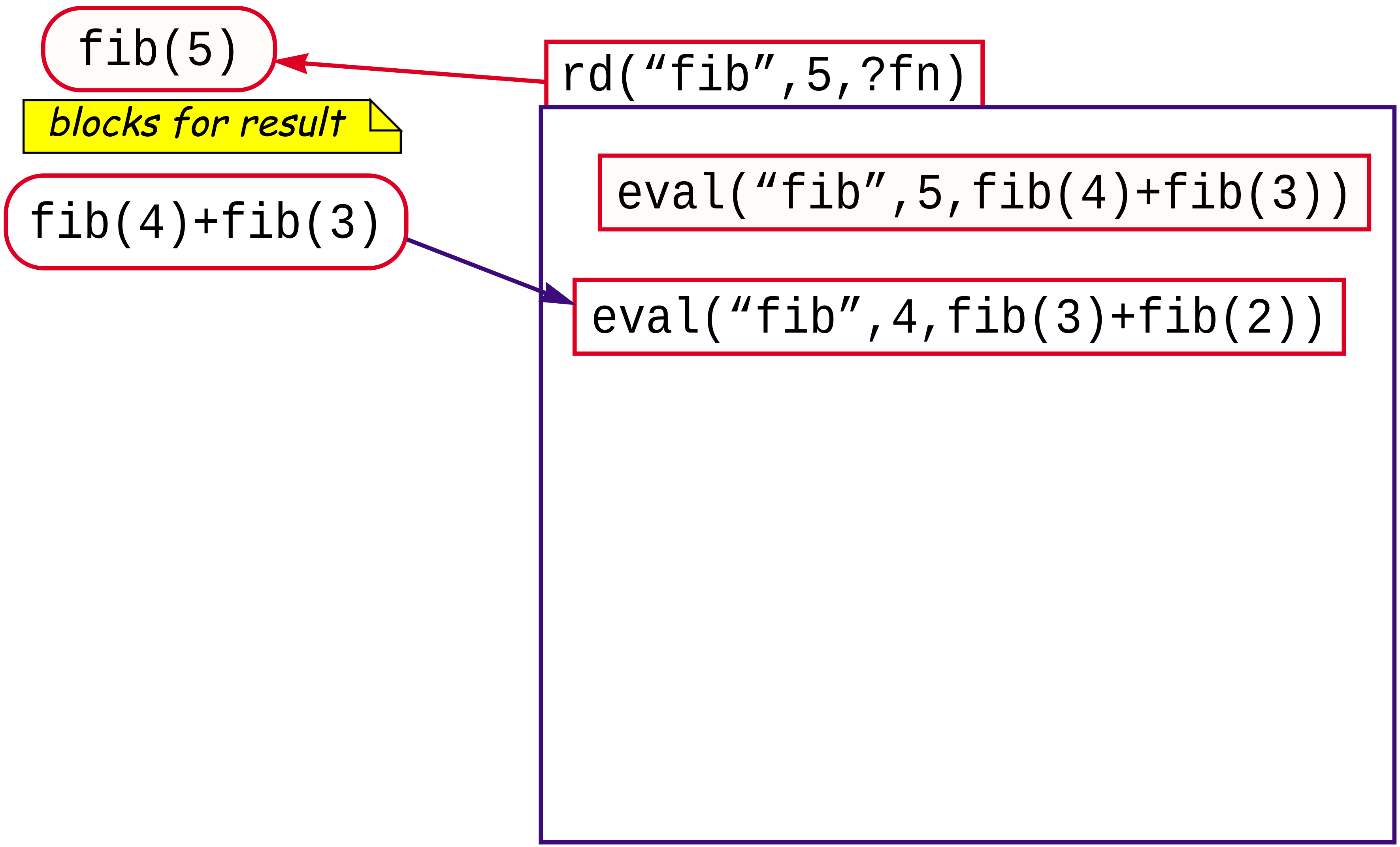
```
int fib(int n) {
  if (rdp("fib", n, ?fibn))     // non-blocking
    return fibn;
  if (n<2) {
    out("fib", n, 1);           // non-blocking
    return 1;
  }
  eval("fib", n, fib(n-1) + fib(n-2)); // asynch
  rd("fib", n, ?fibn);          // blocks
  return(fibn);

} // Post-condition: rdp("fib",n,?fibn) == True
```

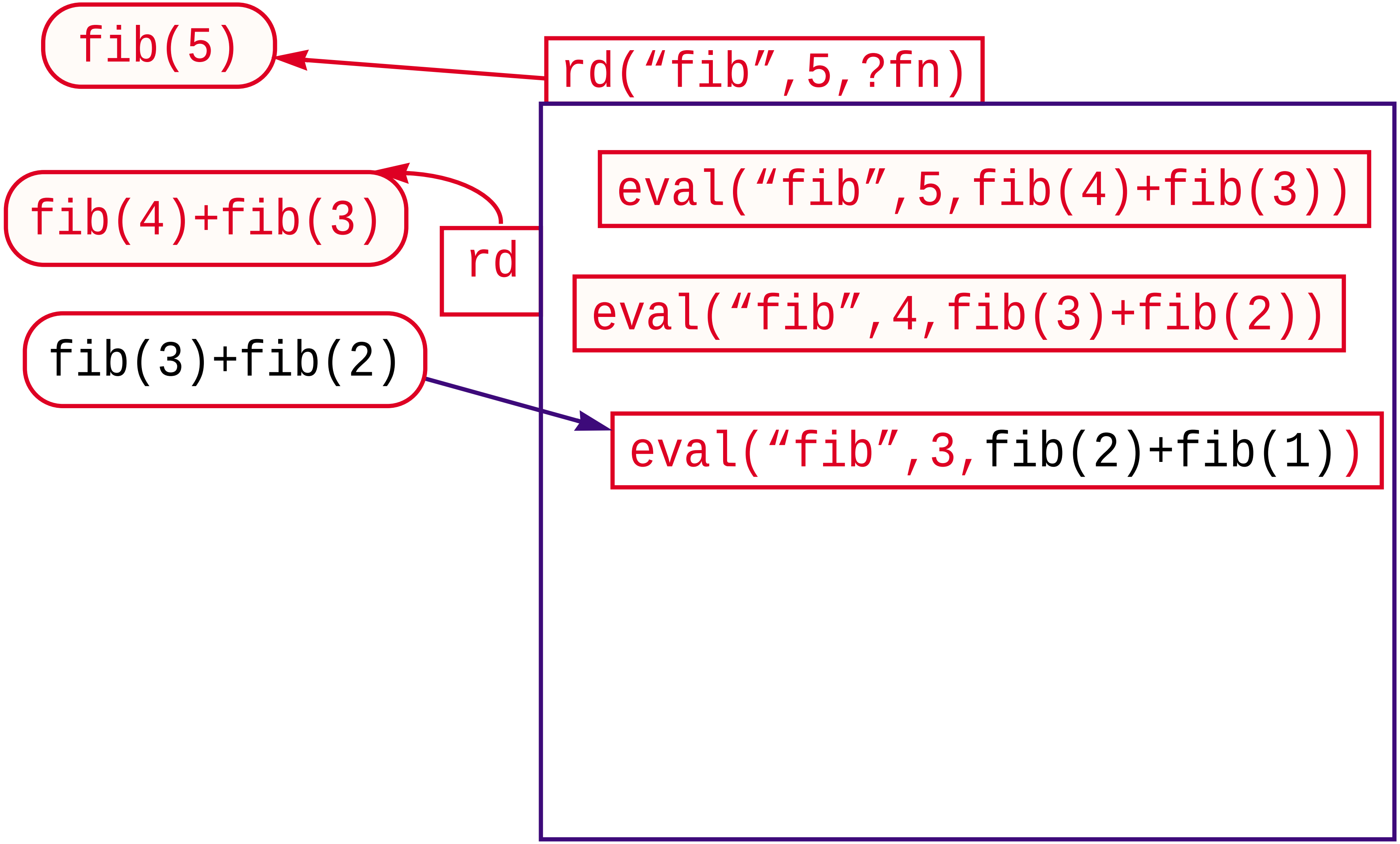
Evaluating Fibonacci



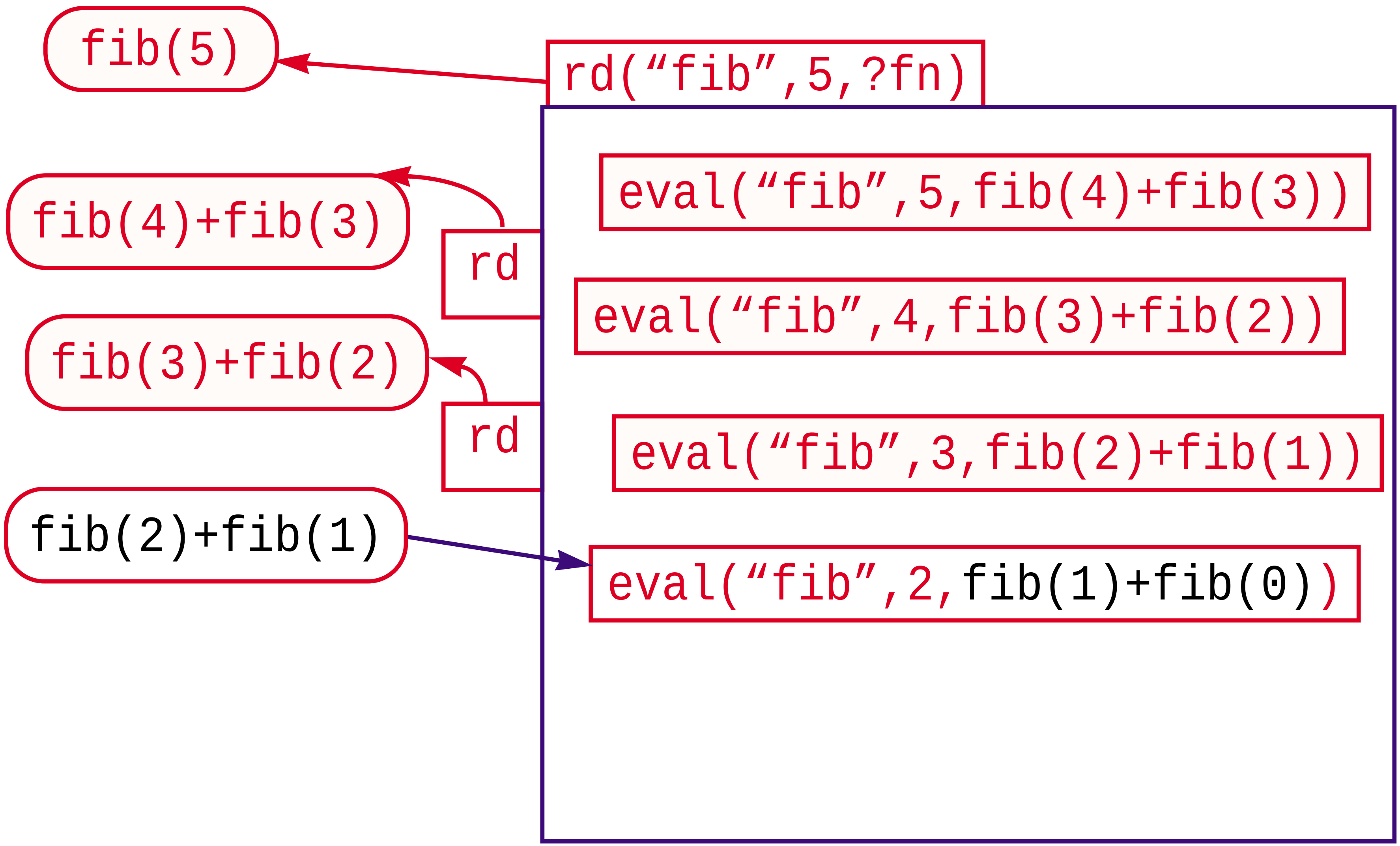
Evaluating Fibonacci



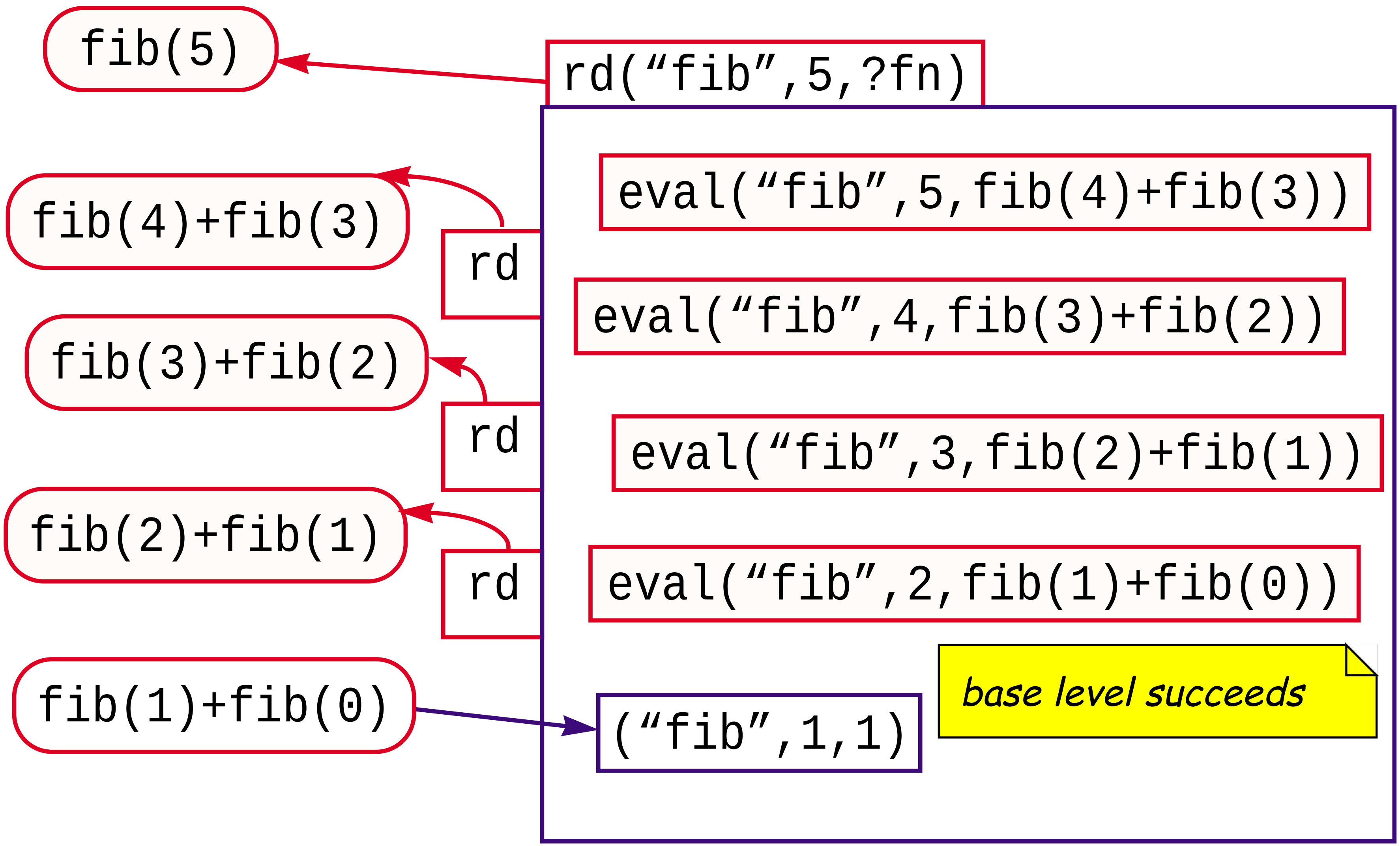
Evaluating Fibonacci



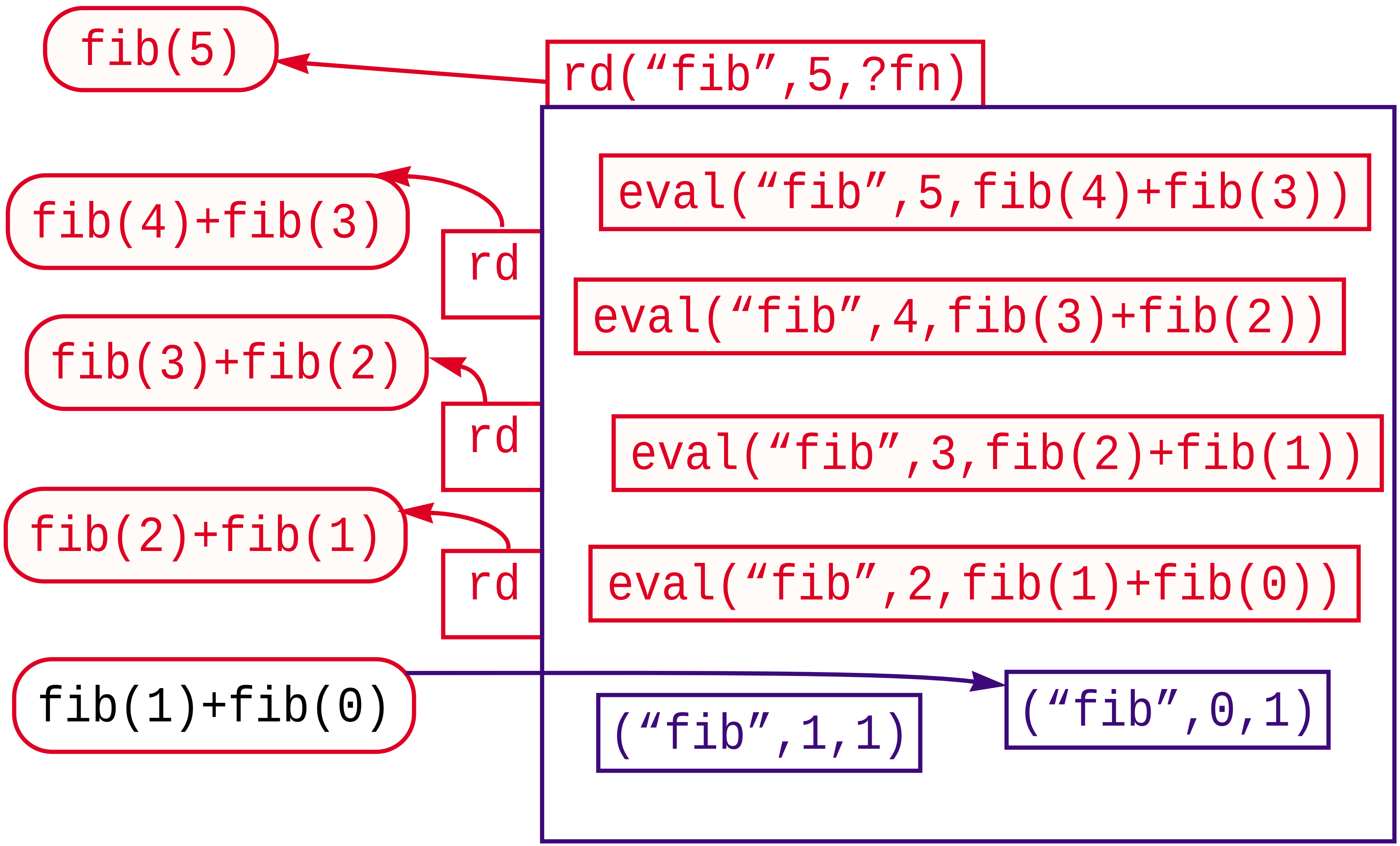
Evaluating Fibonacci



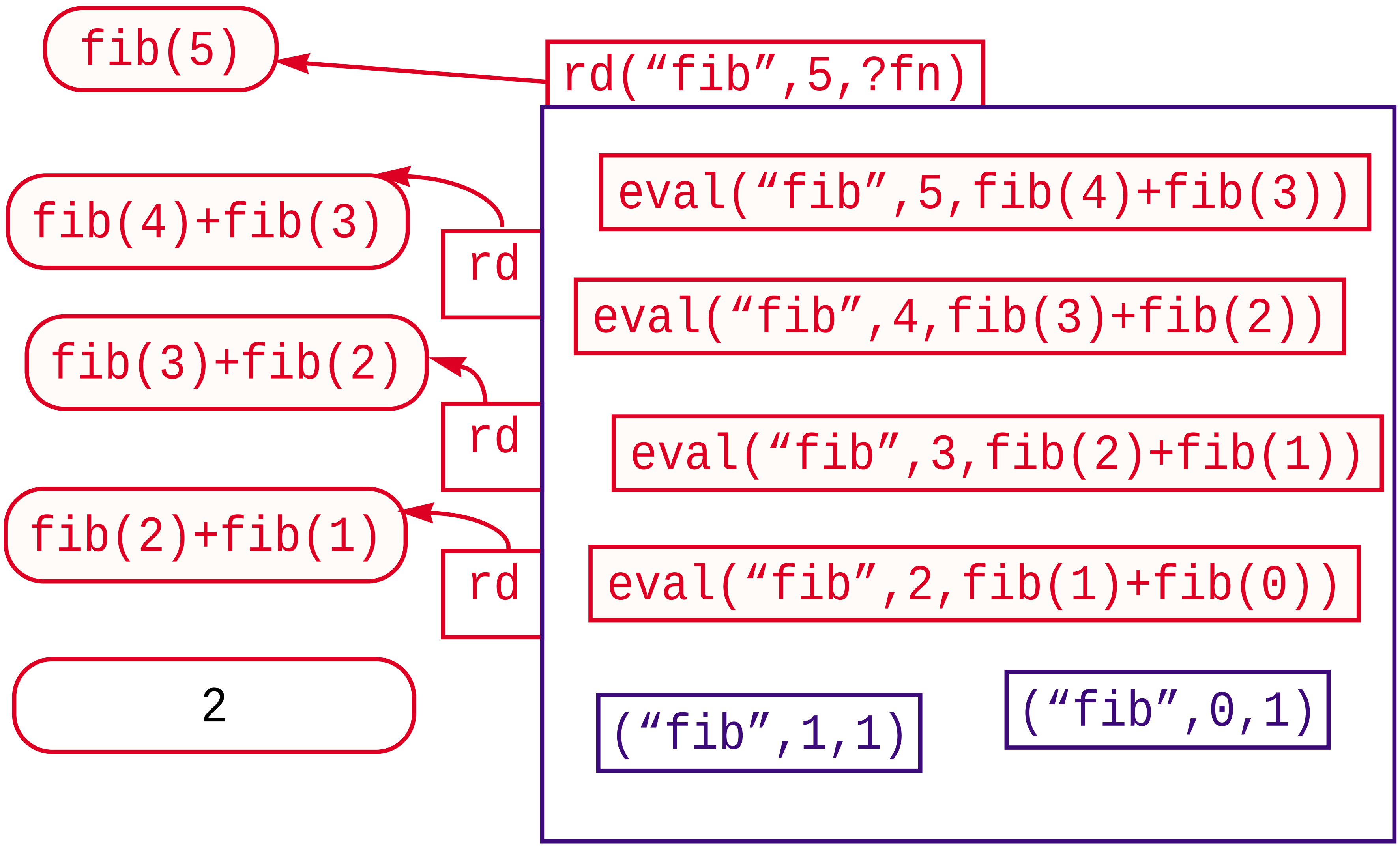
Evaluating Fibonacci



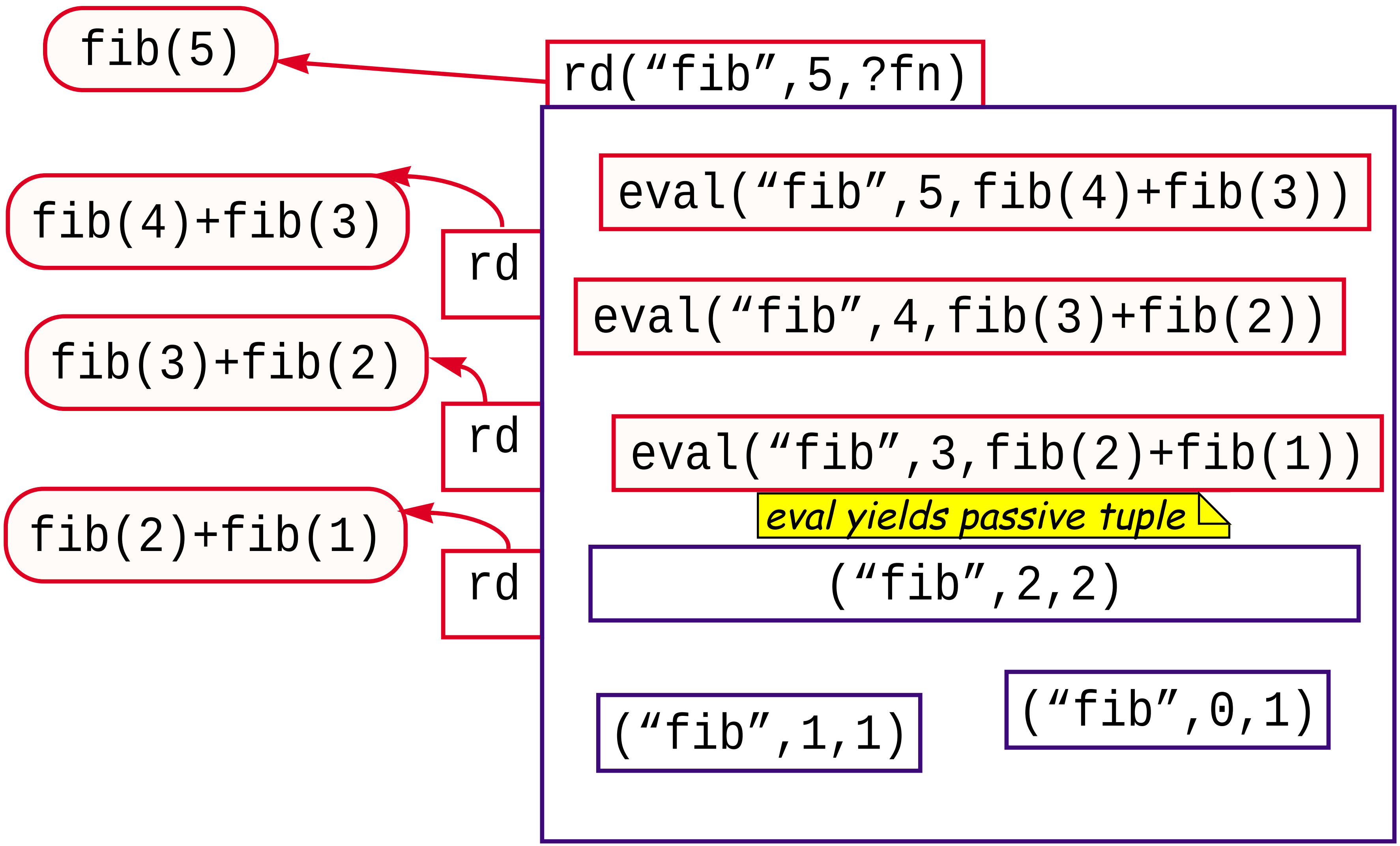
Evaluating Fibonacci



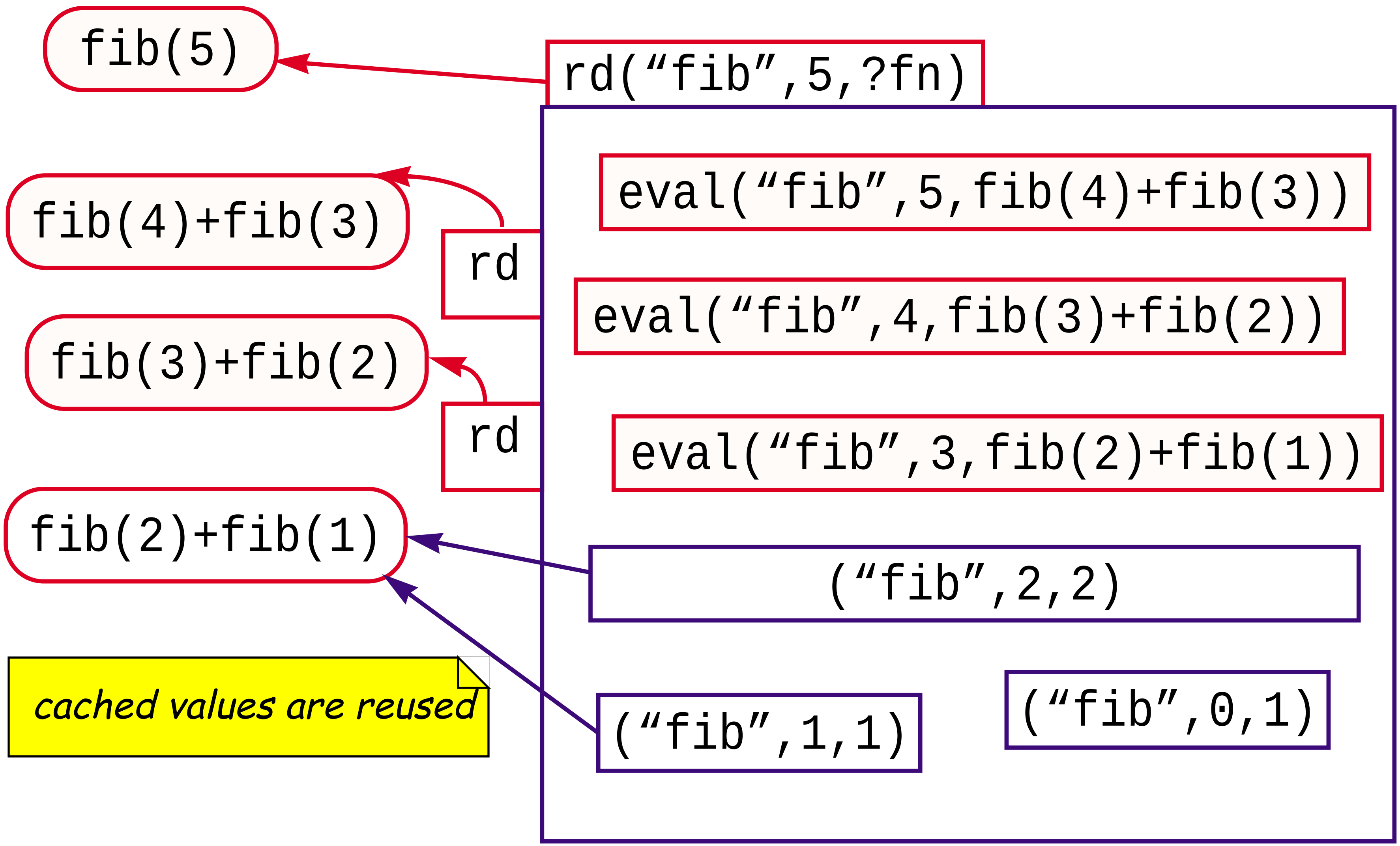
Evaluating Fibonacci



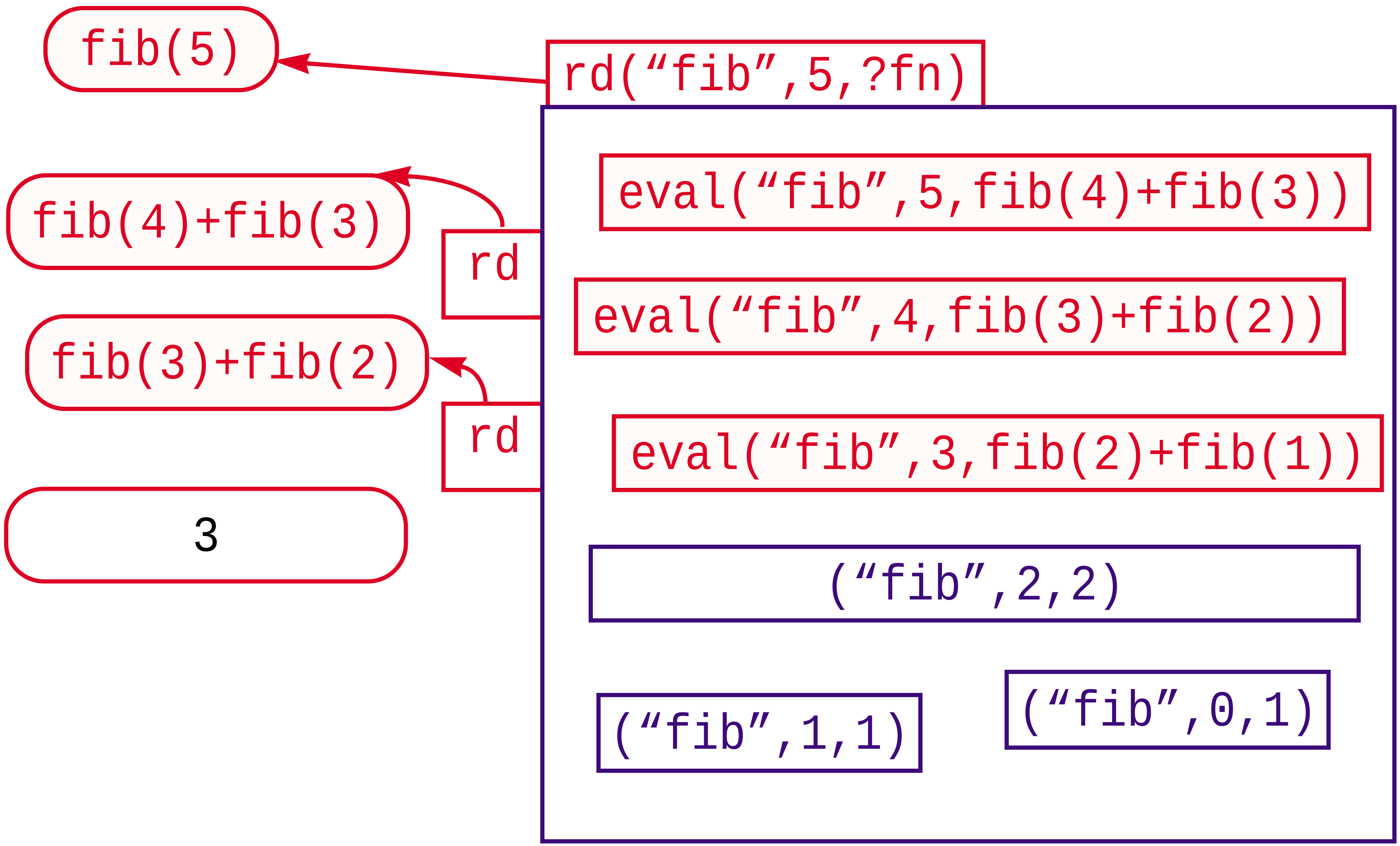
Evaluating Fibonacci



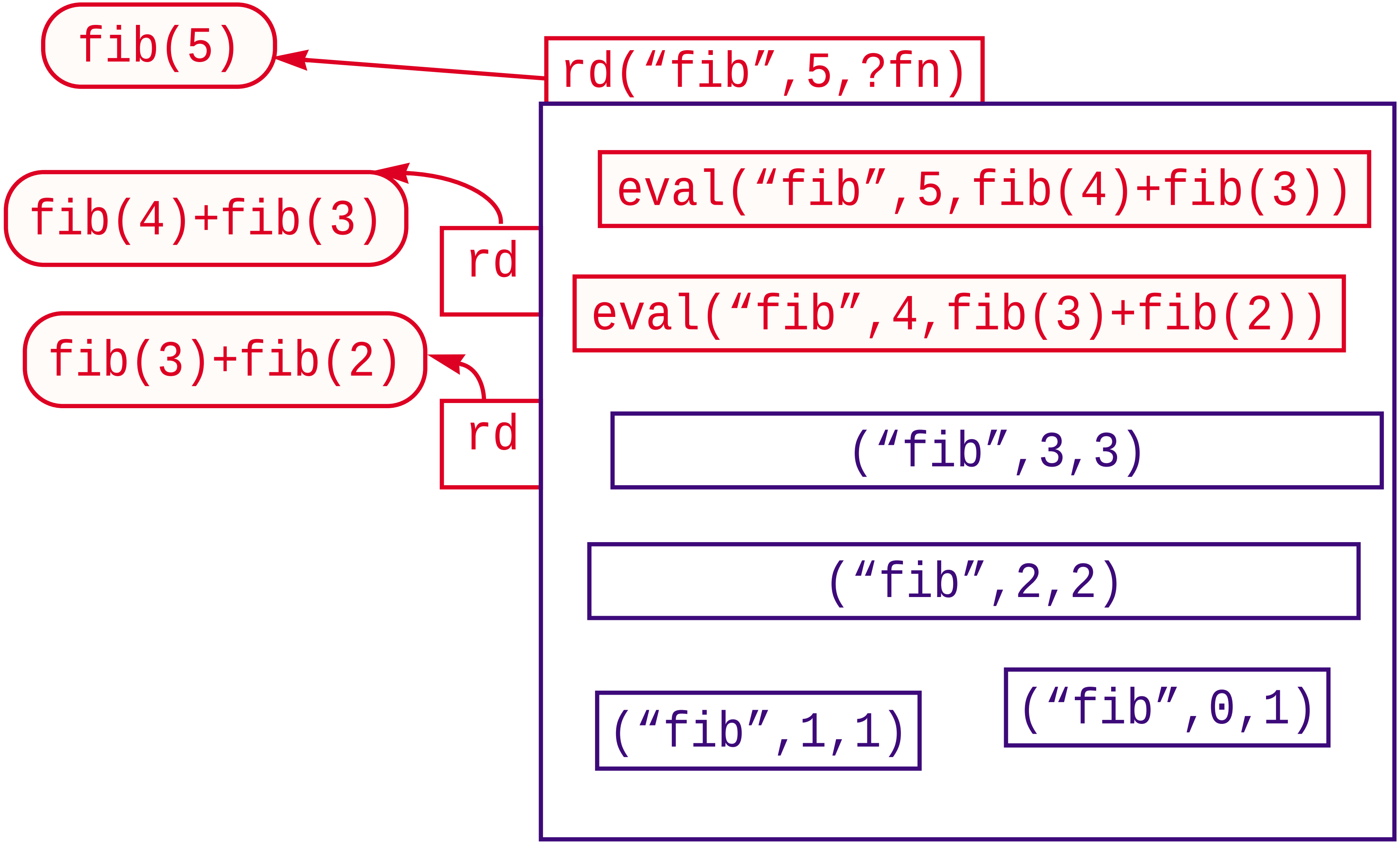
Evaluating Fibonacci



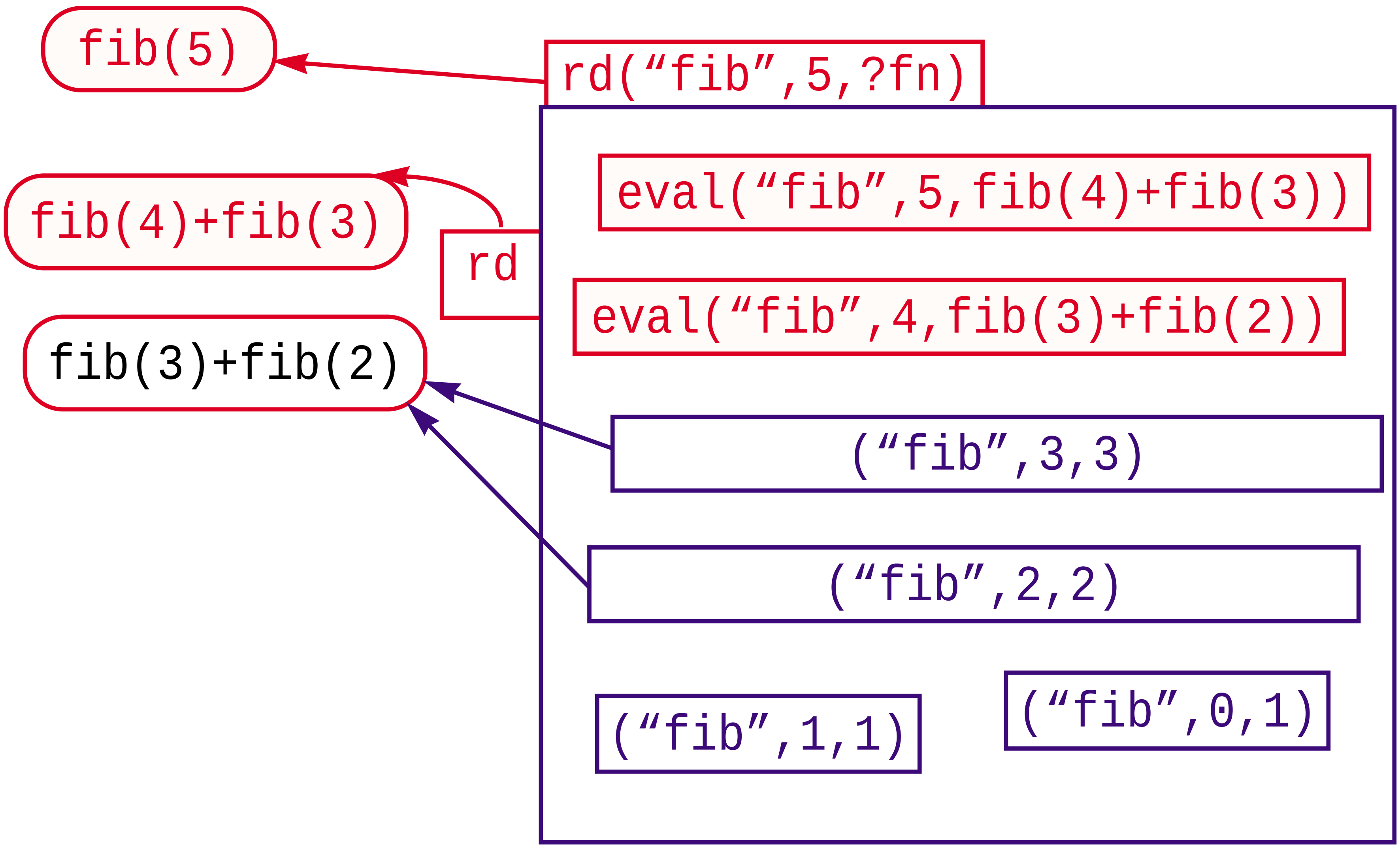
Evaluating Fibonacci



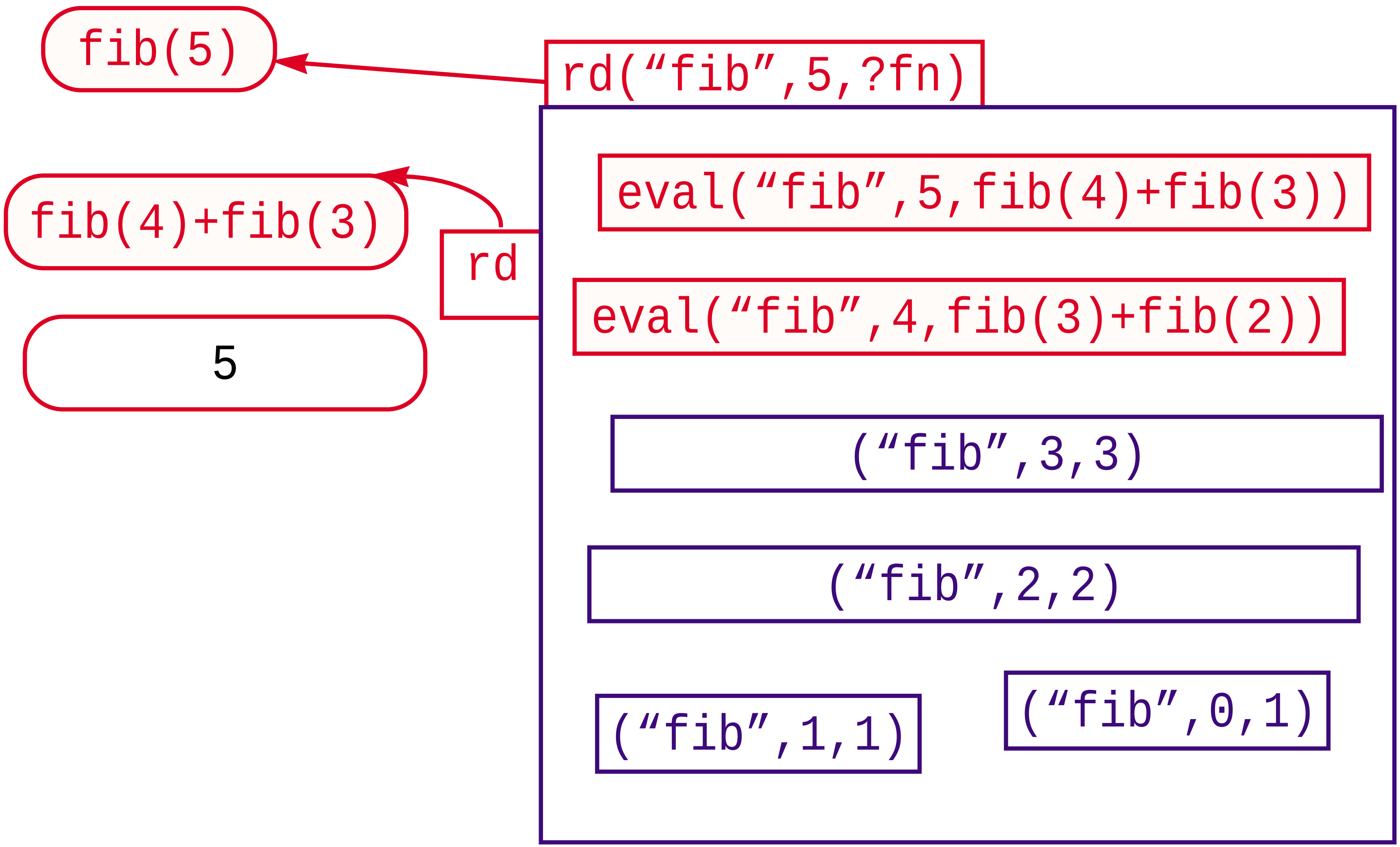
Evaluating Fibonacci



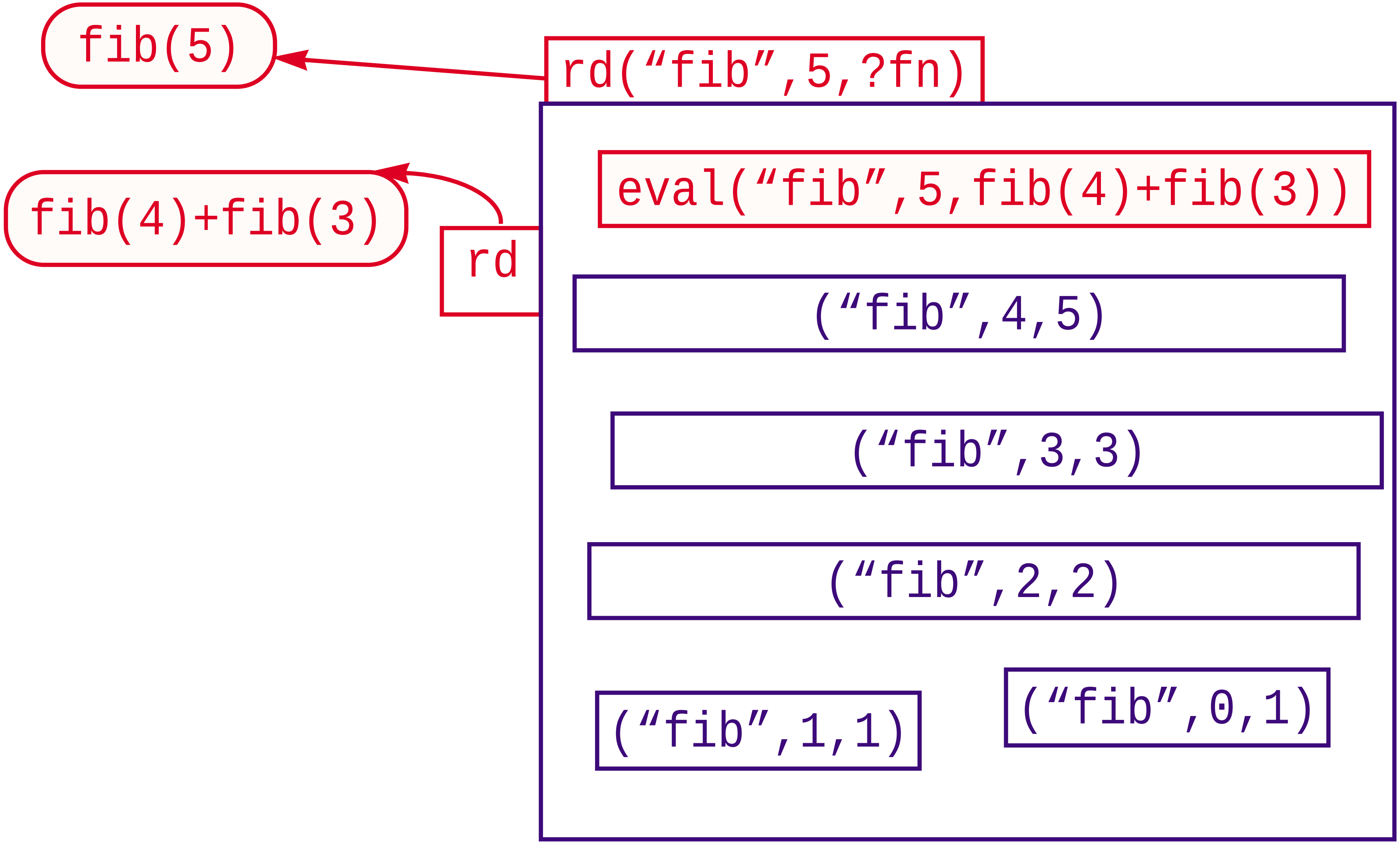
Evaluating Fibonacci



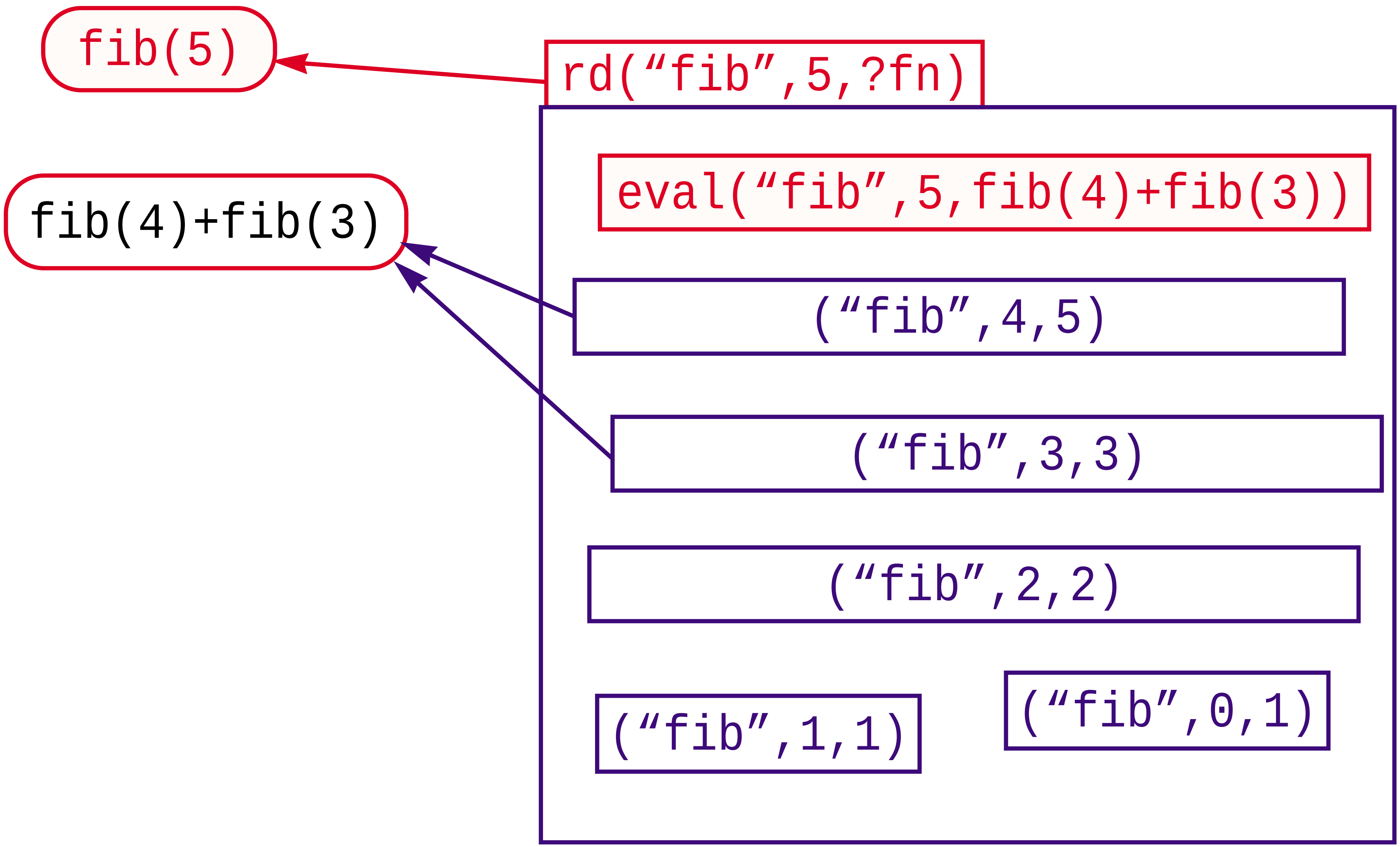
Evaluating Fibonacci



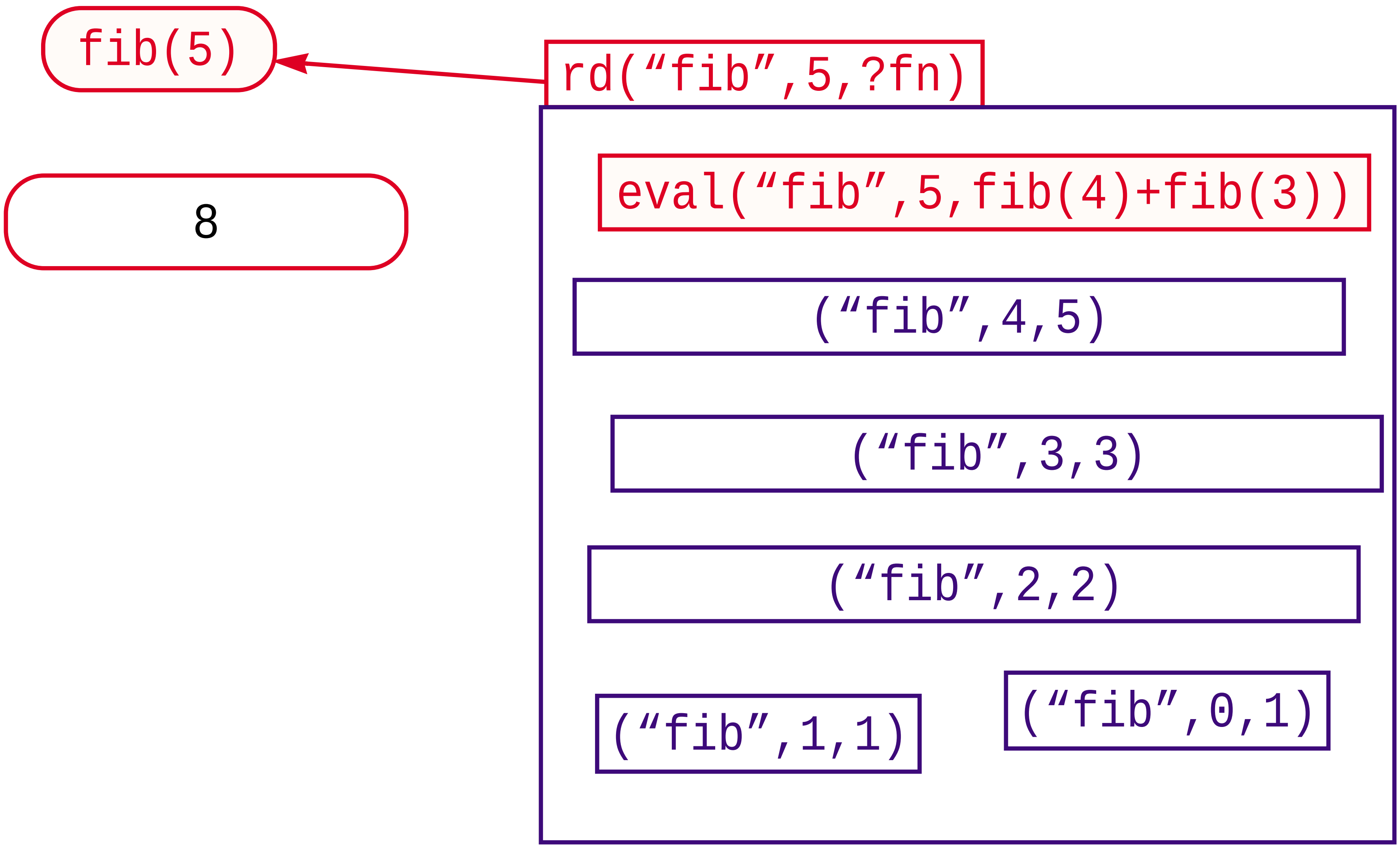
Evaluating Fibonacci



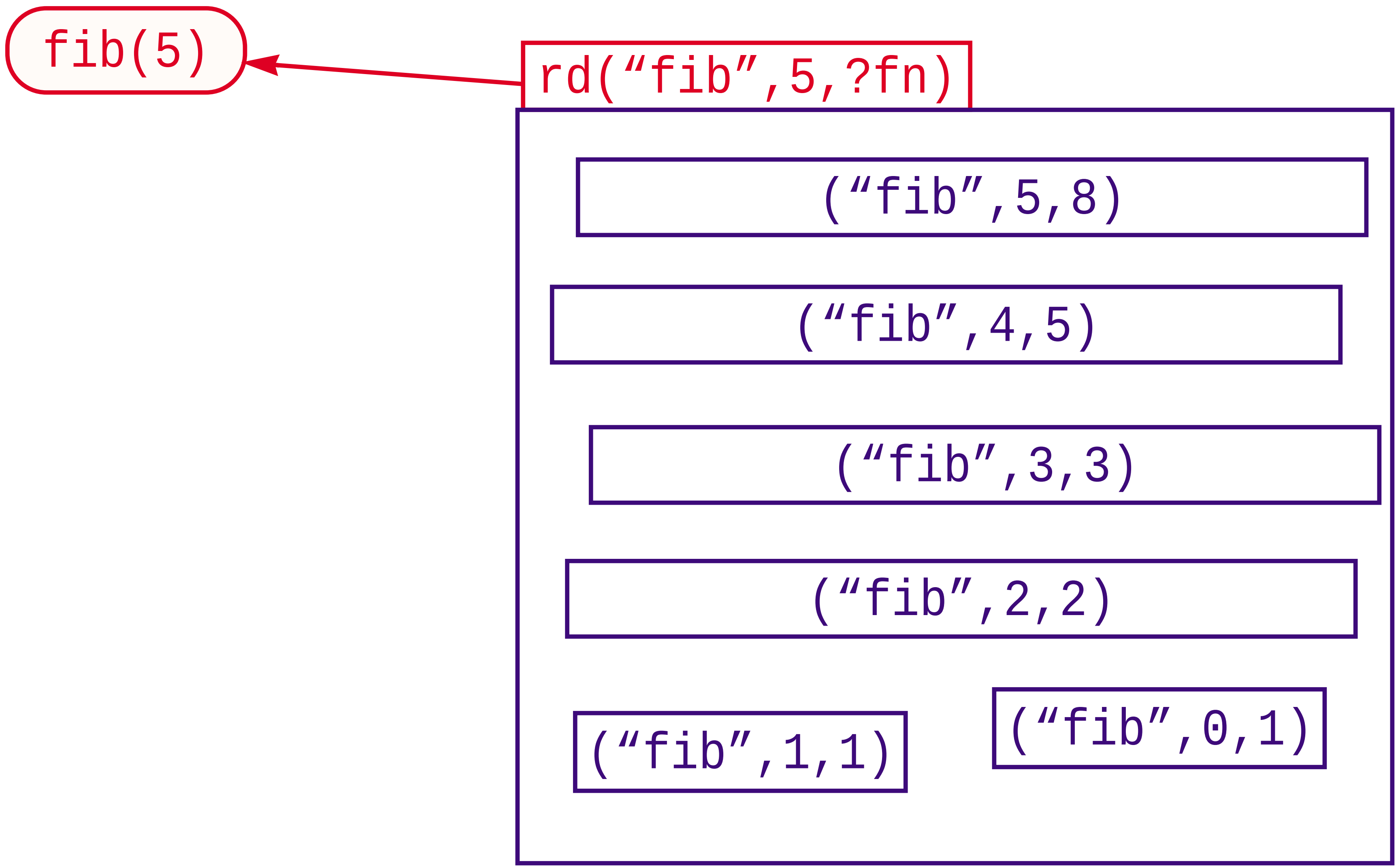
Evaluating Fibonacci



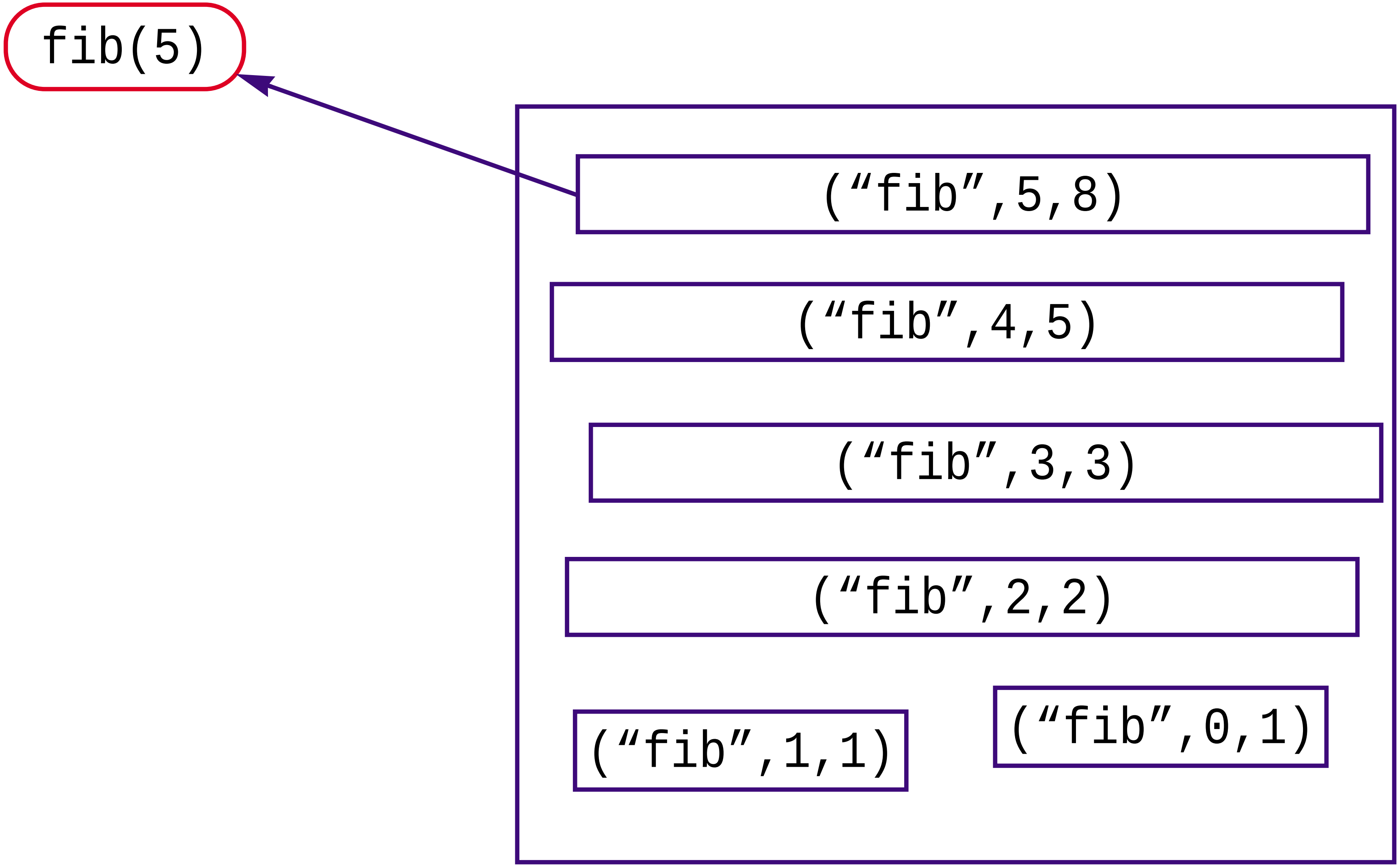
Evaluating Fibonacci



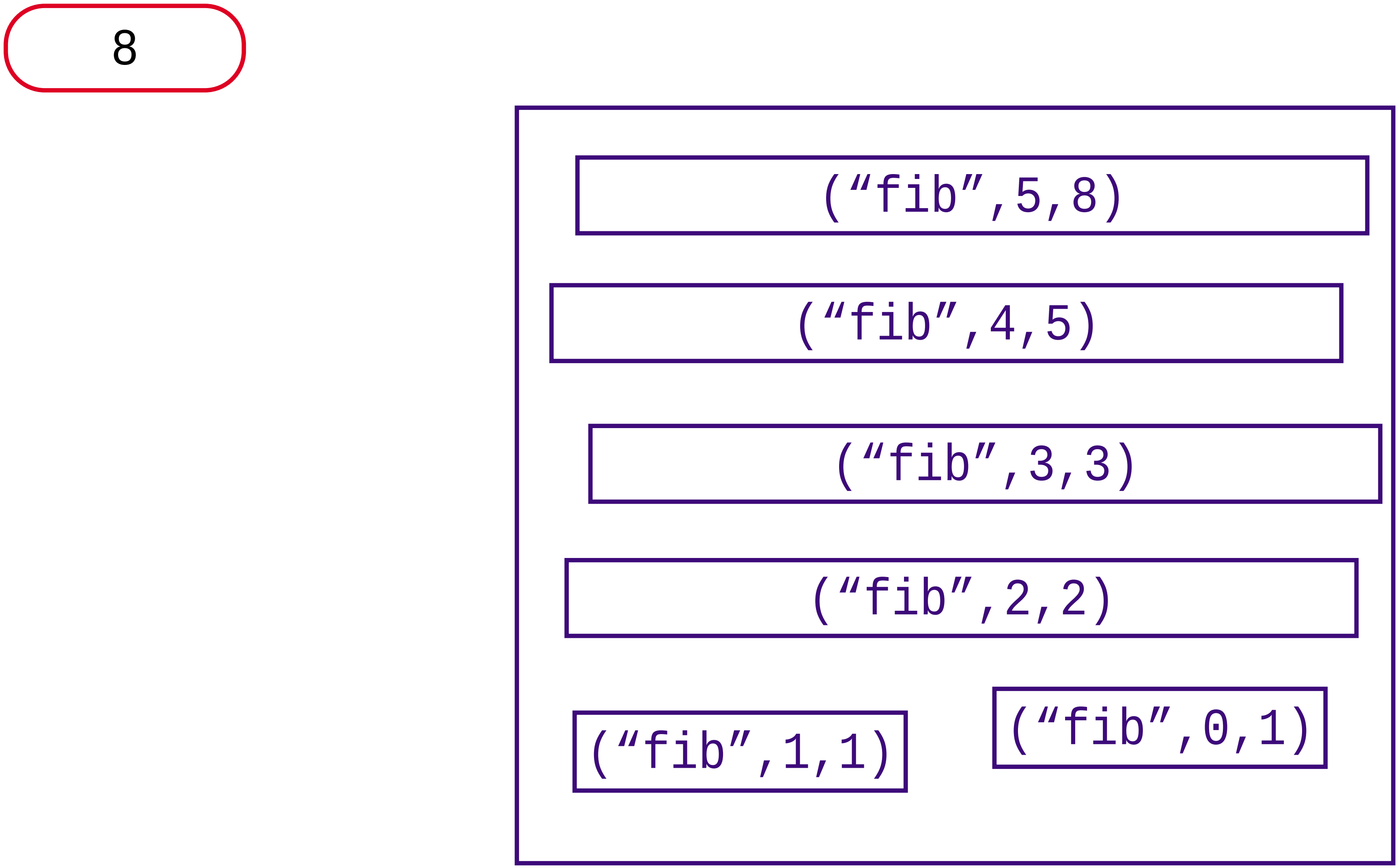
Evaluating Fibonacci



Evaluating Fibonacci



Evaluating Fibonacci



What you should know!

- What is a *Software Architecture*?
- What are advantages and disadvantages of *Layered Architectures*?
- What is a *Flow Architecture*? What are the options and tradeoffs?
- What are *Blackboard Architectures*? What are the options and tradeoffs?
- How does *result parallelism* differ from *agenda parallelism*?
- How does *Linda* support *coordination* of concurrent agents?

Can you answer these questions?

- How would you model message-passing agents in Java?
- How would you classify Client/Server architectures?
- Are there other useful styles we haven't yet discussed?
- How can we prove that the Active Prime Sieve is correct? Are you sure that new Active Primes will join the chain in the correct order?
- Which Blackboard styles are better when we have multiple processors?
- Which are better when we just have threads on a monoprocessor?
- What will happen if you start two concurrent Fibonacci computations?