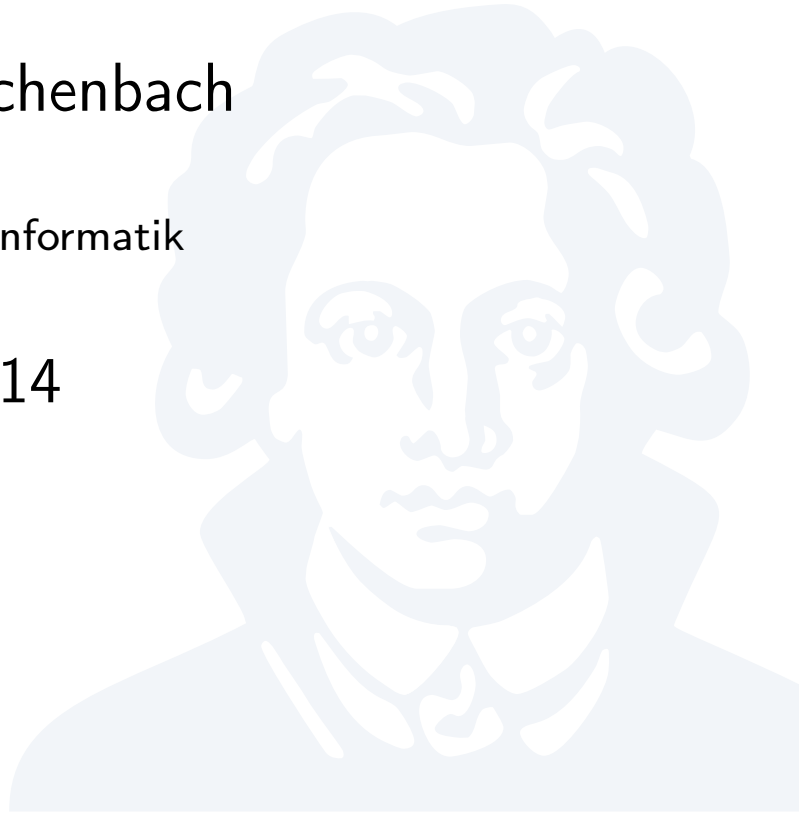
Foundations of Programming Languages

Closures and Thunks

Prof. Dr. Christoph Reichenbach

Fachbereich 12 / Institut für Informatik

14. November 2014

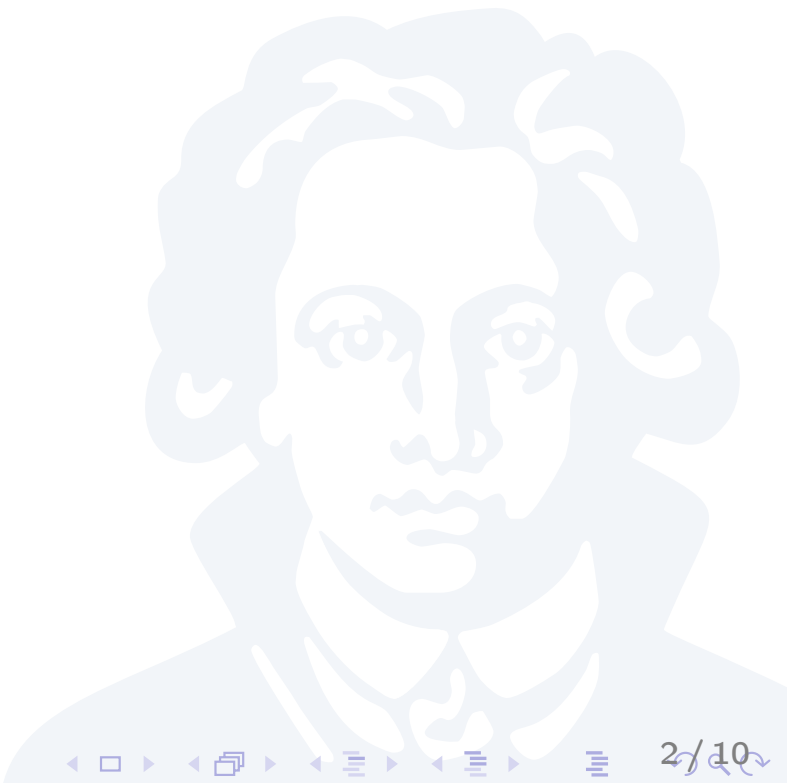


Partially Applied Functions

Haskell

```
add x =
    let add2 y = x + y
    in add2
```

```
add 2 3 --> 5
add 2 7 --> 9
```



Partially Applied Functions

Haskell

```
add x =
    let add2 y = x + y
    in add2
```

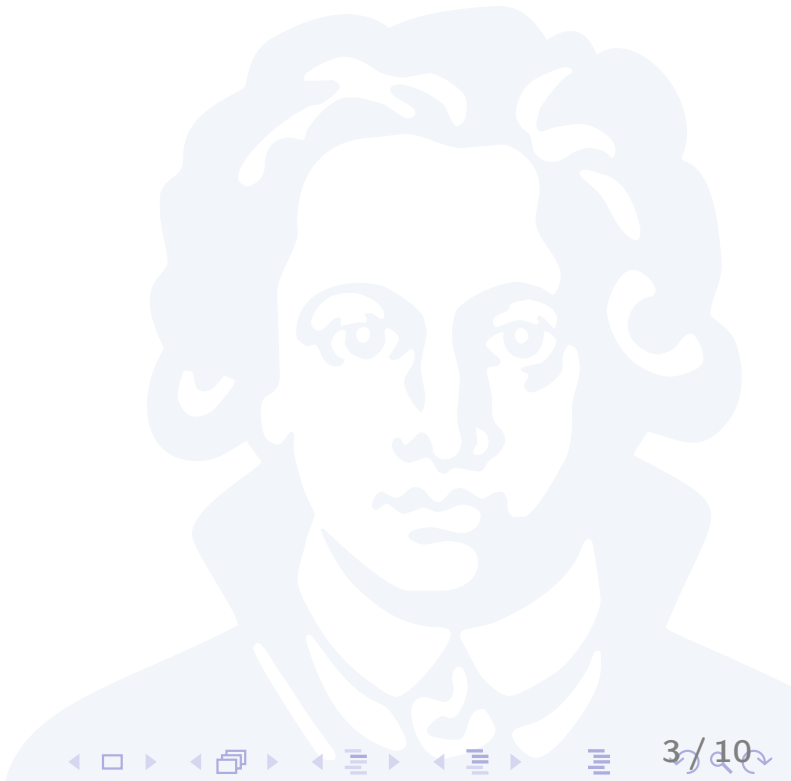
```
add 2 3 --> 5
add 2 7 --> 9
add 3   --> function: x ↦ x + 3
```

- Can pass ‘add 3’ as function parameter
- Other languages: store ‘add 3’ in variable

How can we *implement* such a thing?

Naive implementation

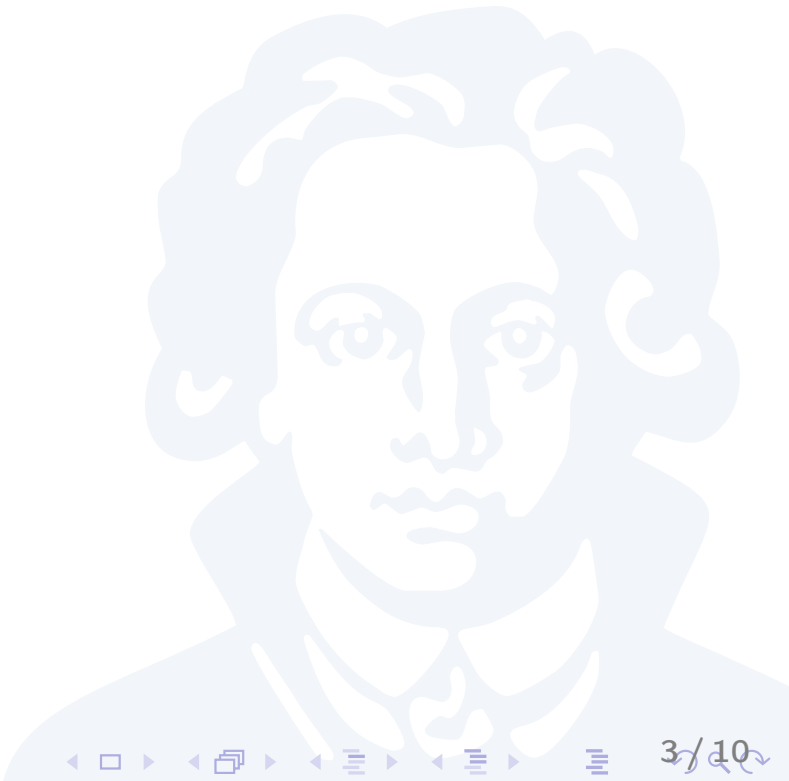
```
add 3
```



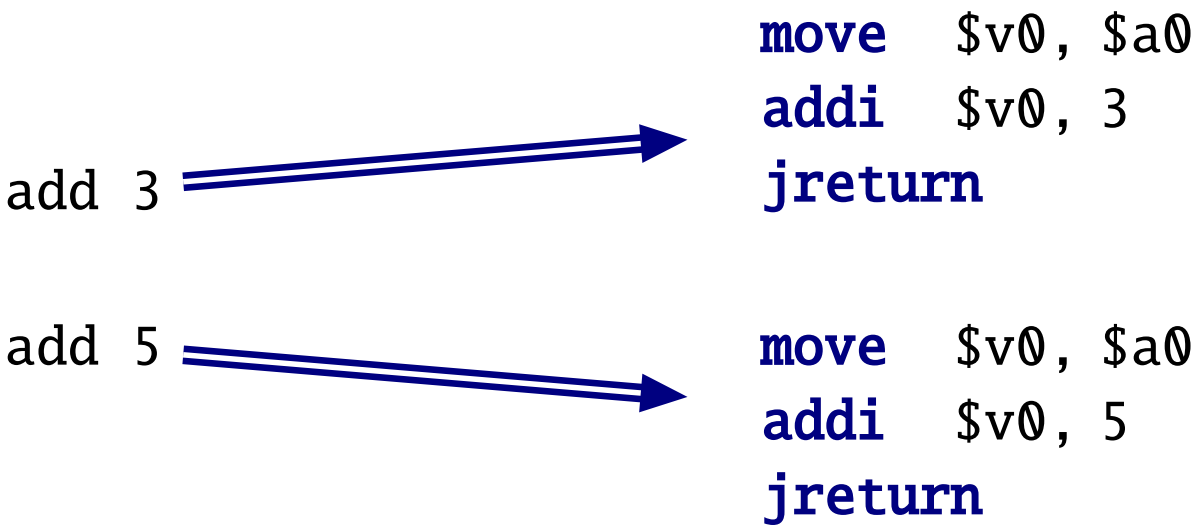
Naive implementation

```
add 3
```

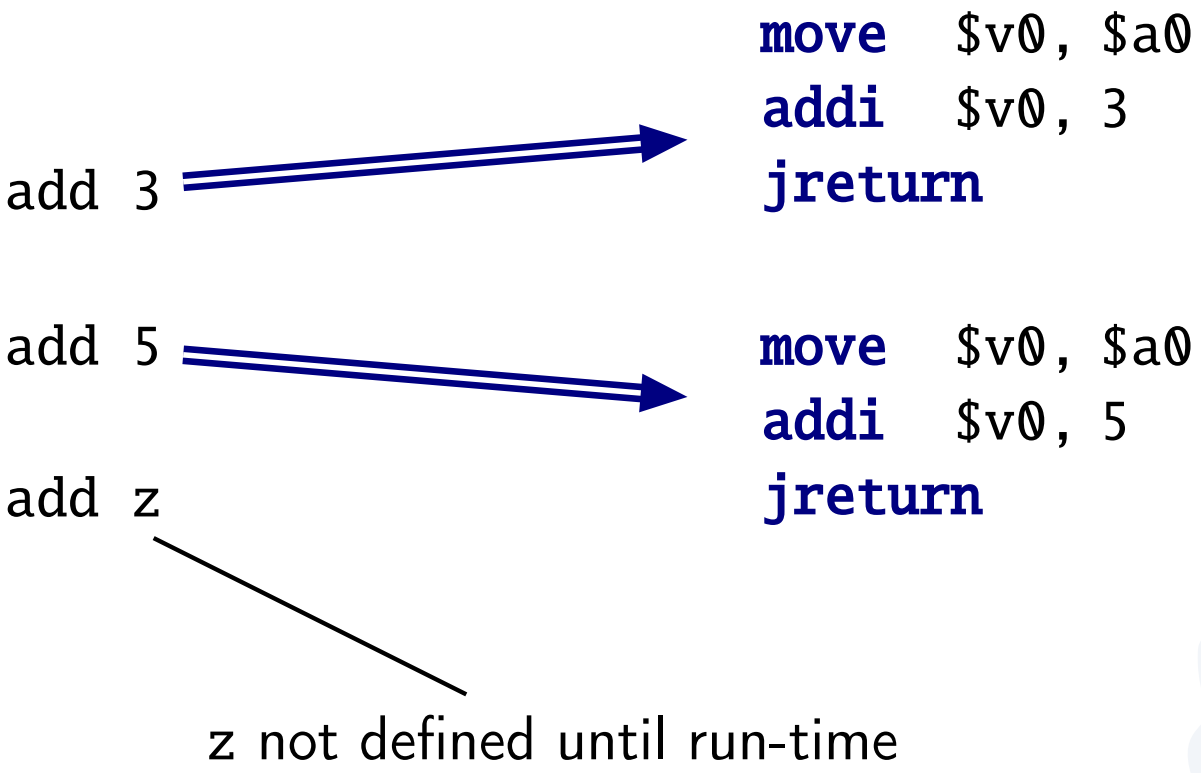
```
move   $v0, $a0
addi   $v0, 3
jreturn
```



Naive implementation



Naive implementation



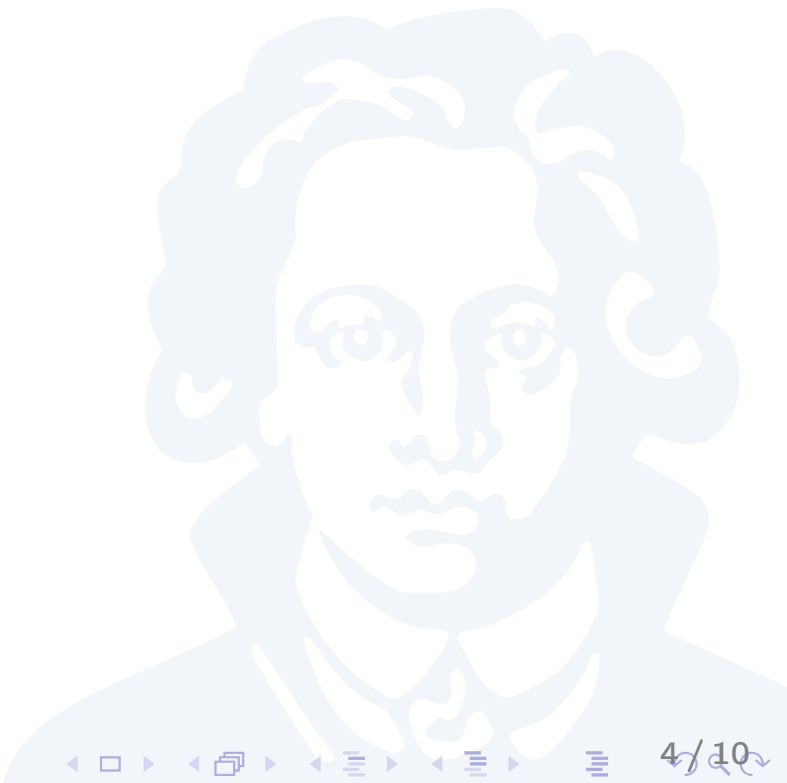
Not a fully general implementation scheme

Closures

Idea: store function + *environment*:

$$\langle f, e \rangle$$

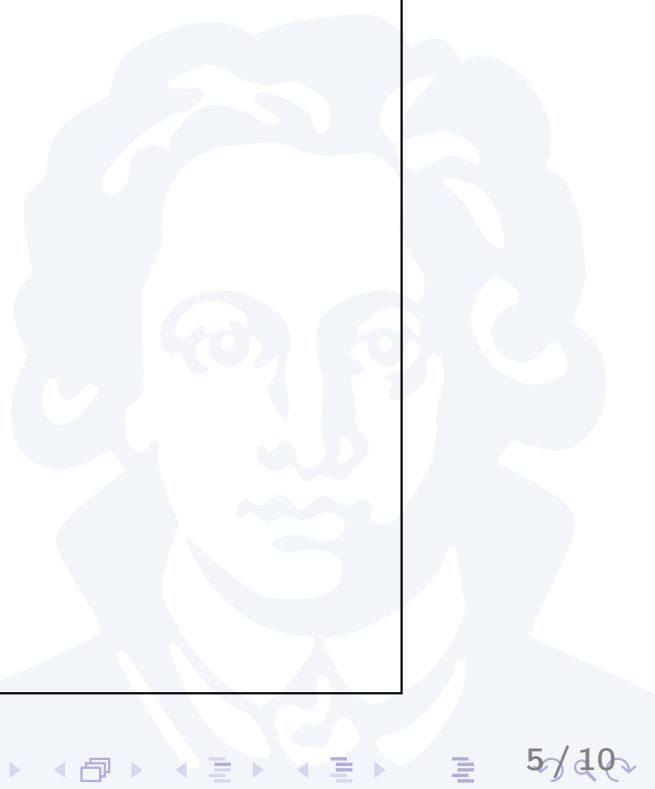
- f: Code pointer
- e: Environment
  captures variables we know



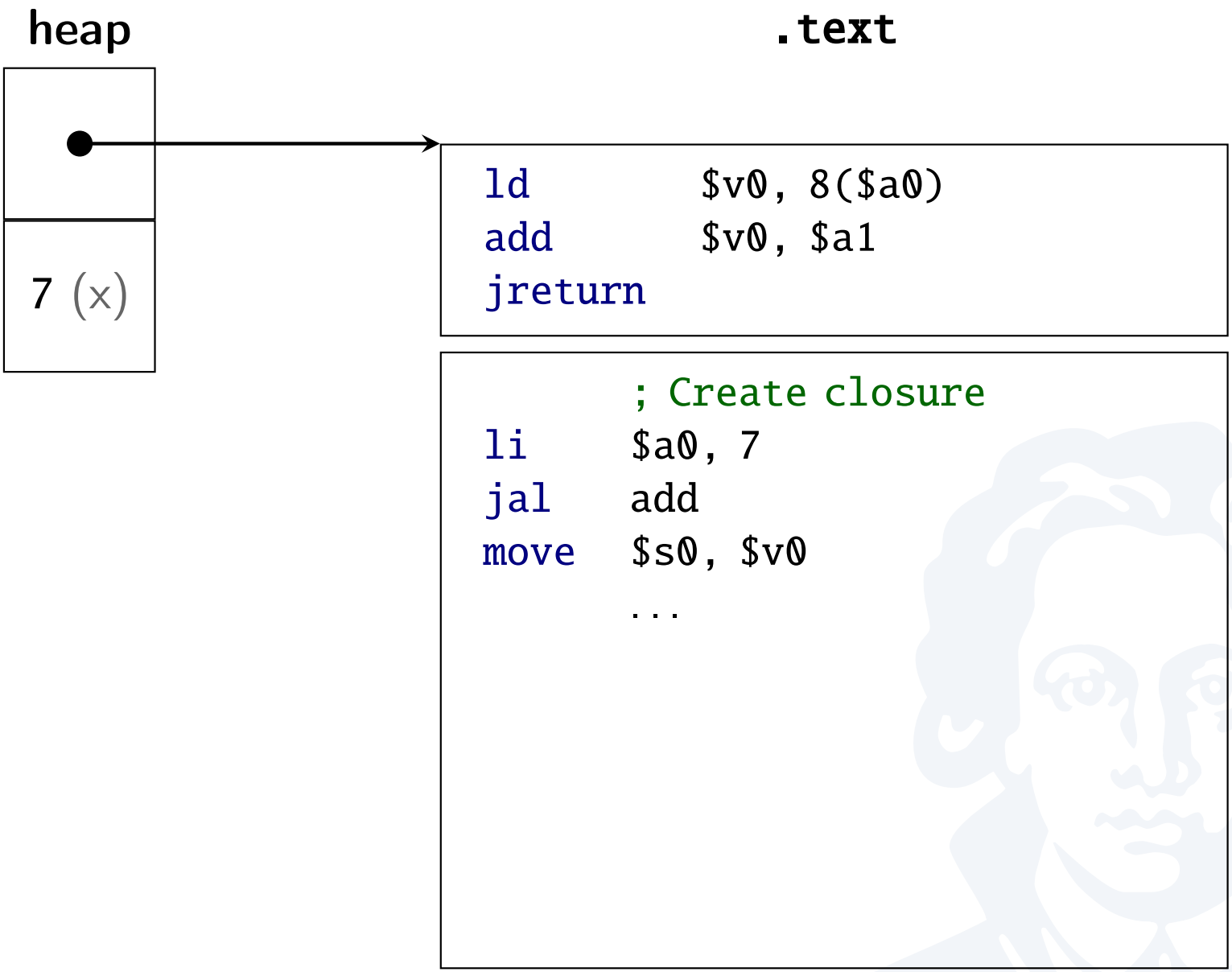
Executing Closures

heap **.text**

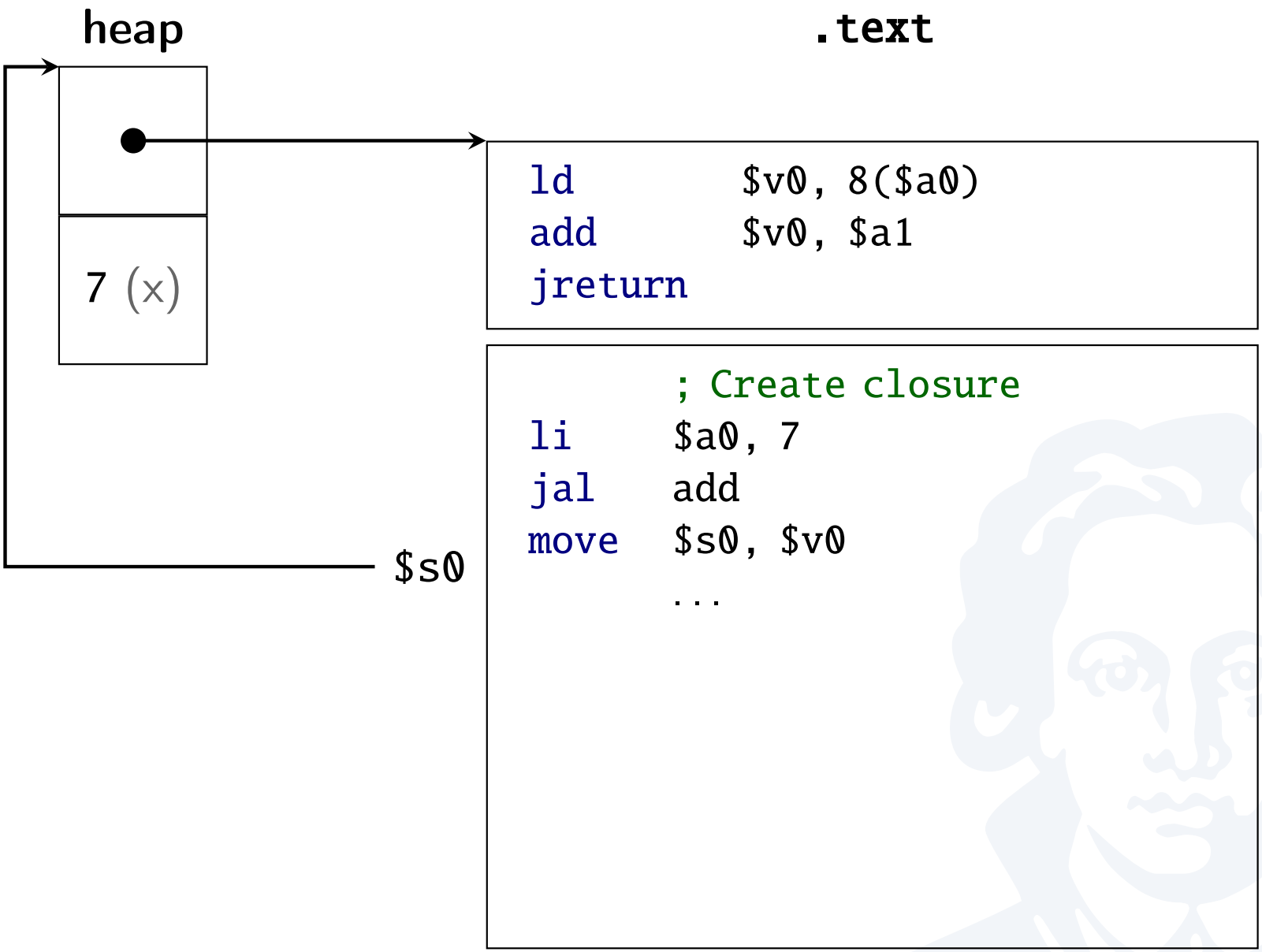
```
        ; Create closure
li      $a0, 7
jal     add
move    $s0, $v0
        ...
```



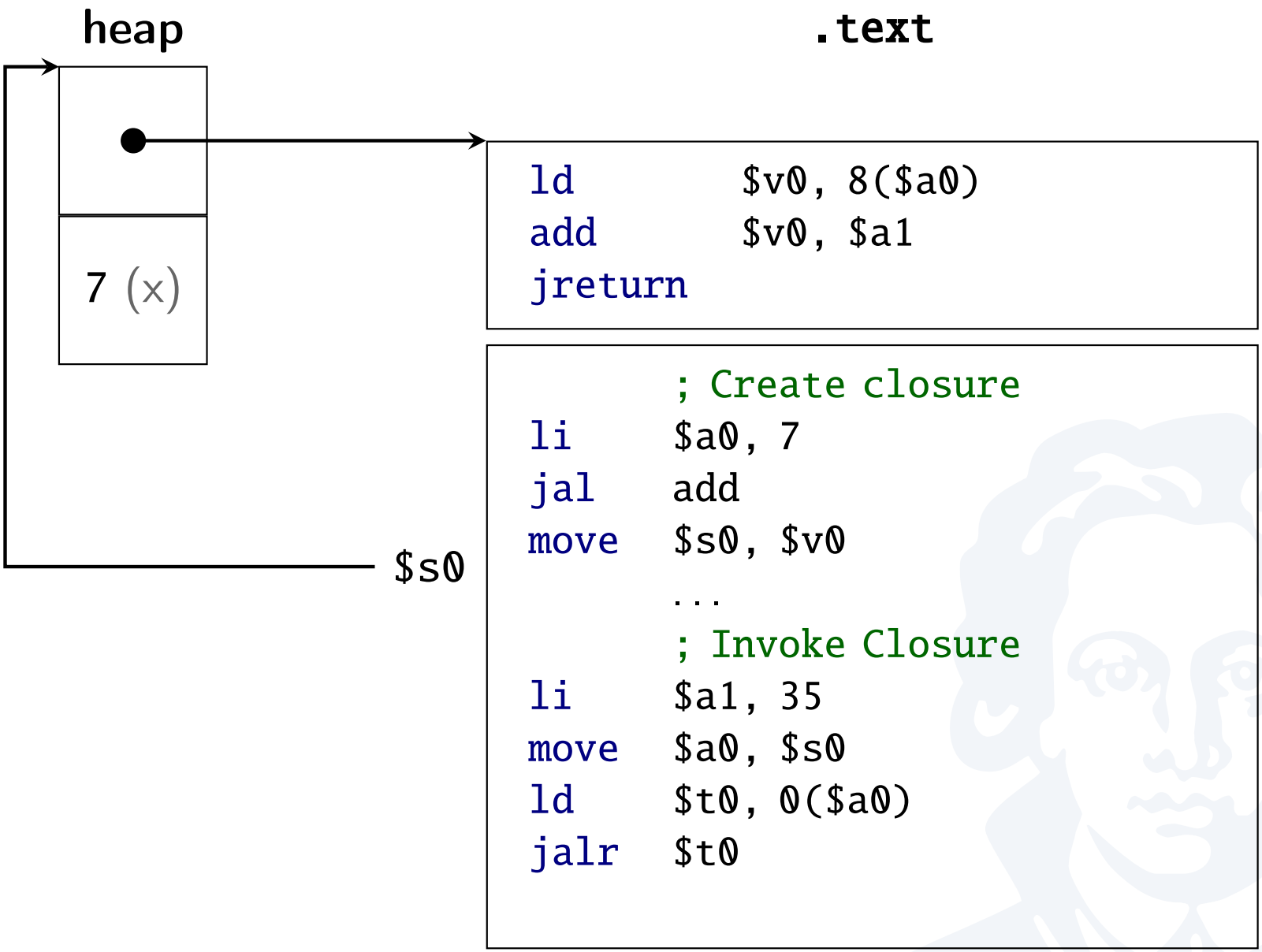
Executing Closures



Executing Closures

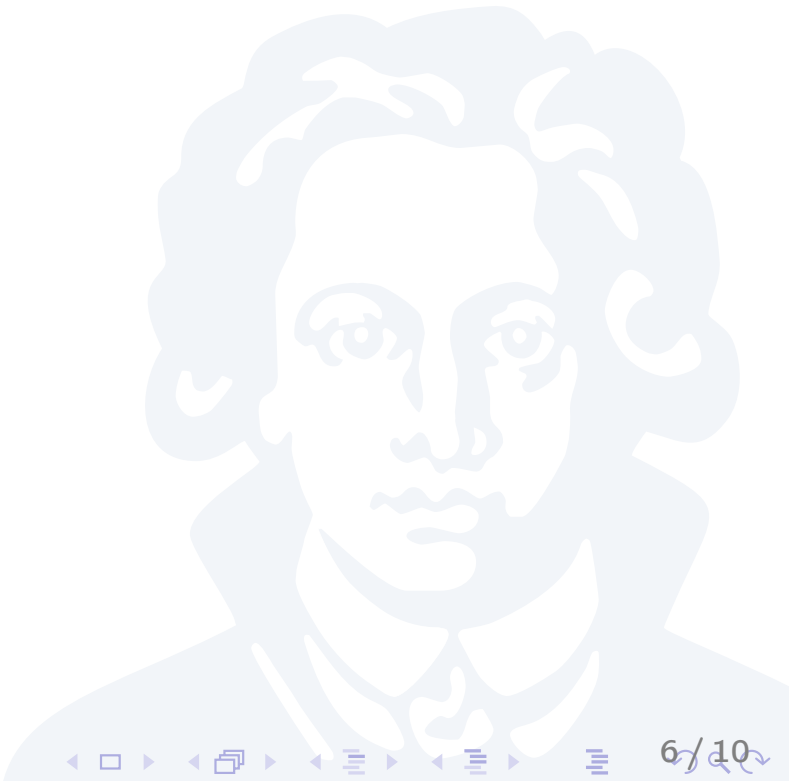


Executing Closures



Thunks

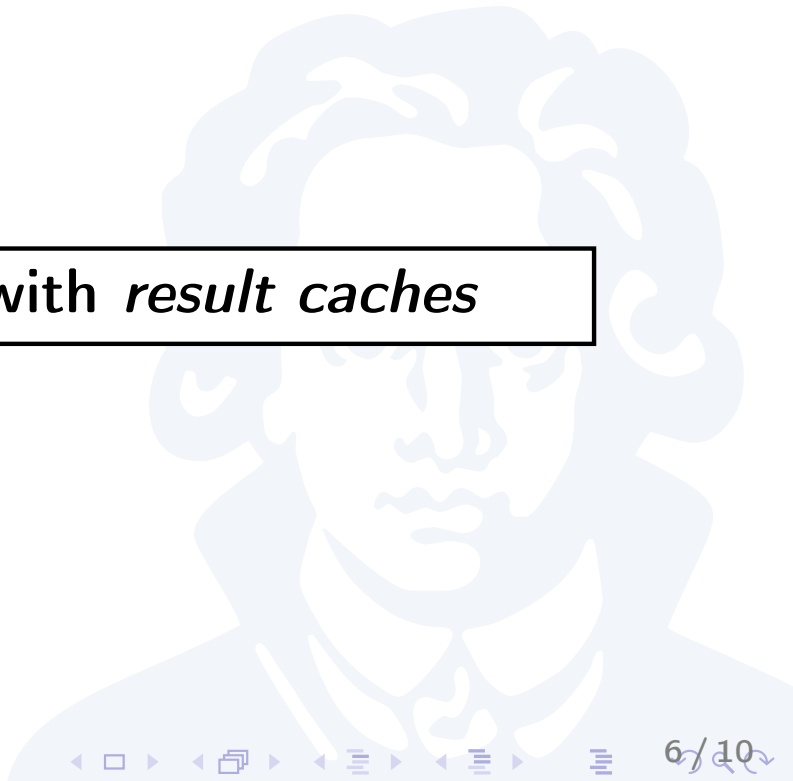
- *Thunk*: Closure that expects no parameters
- Used to implement *call-by-name*



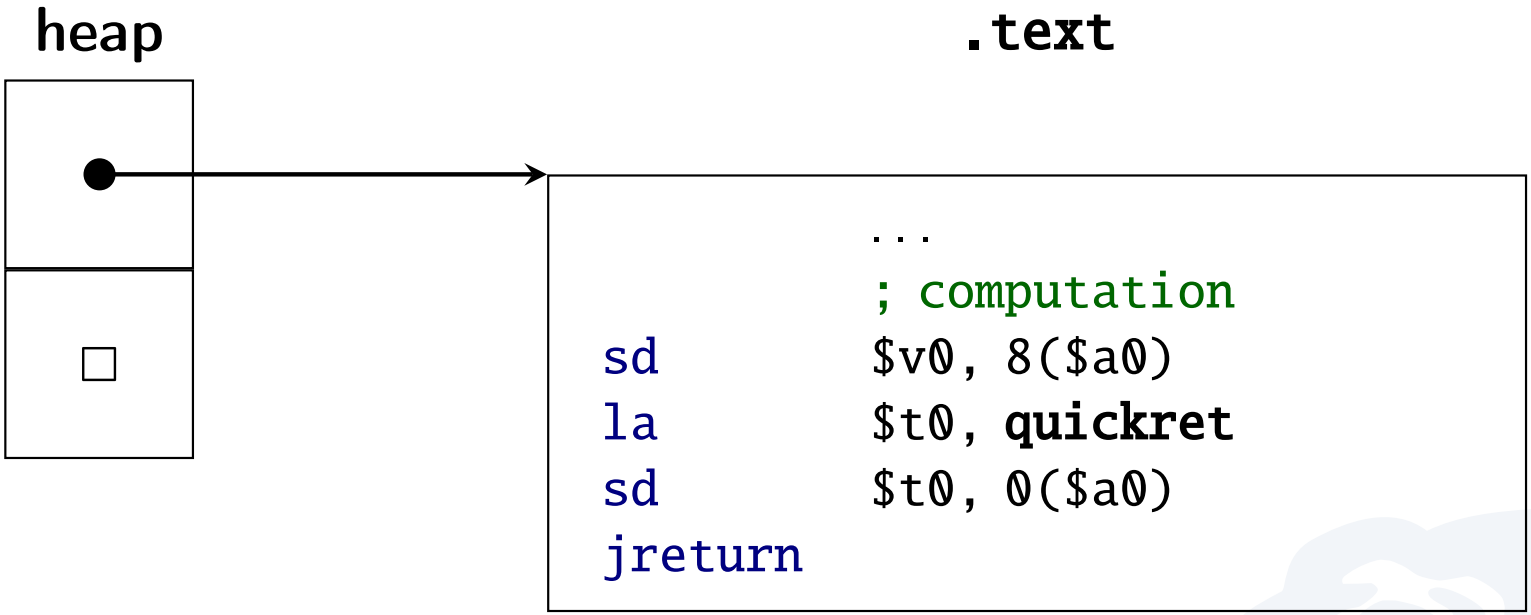
Thunks

- *Thunk*: Closure that expects no parameters
- Used to implement *call-by-name*

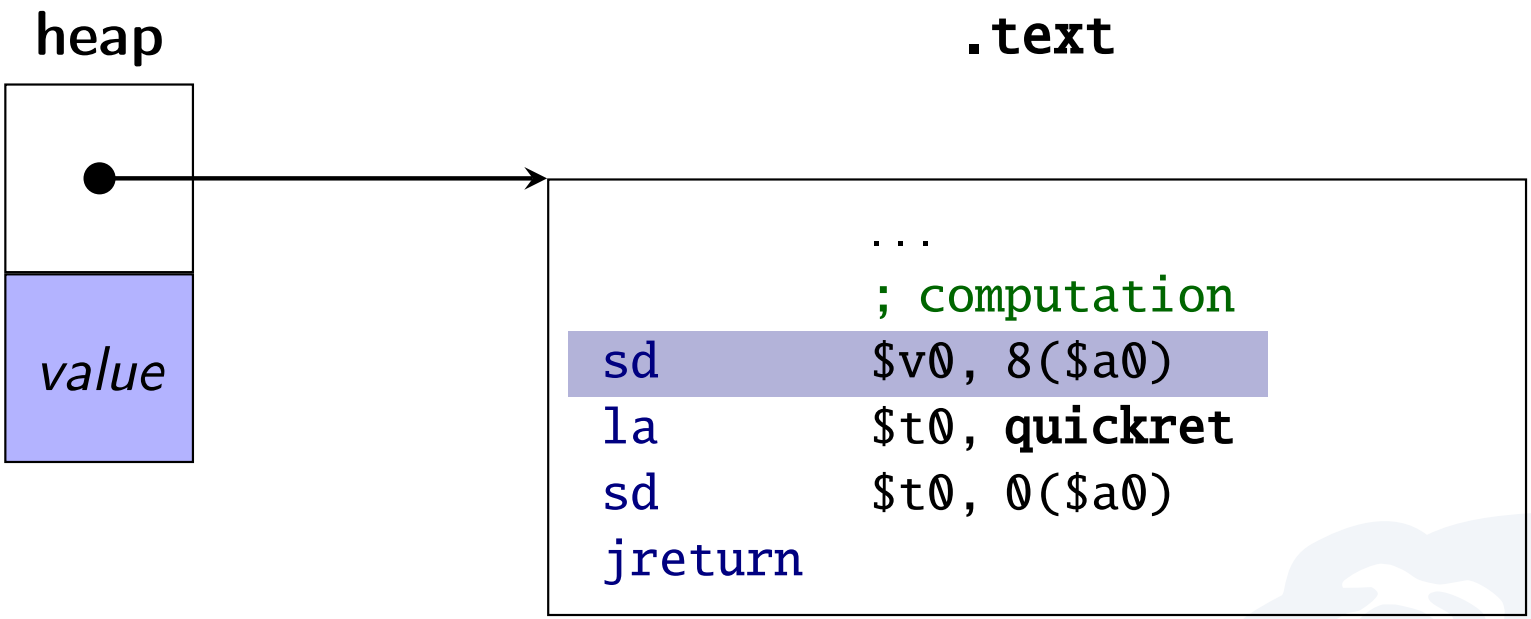
Some languages extend thunks with *result caches*



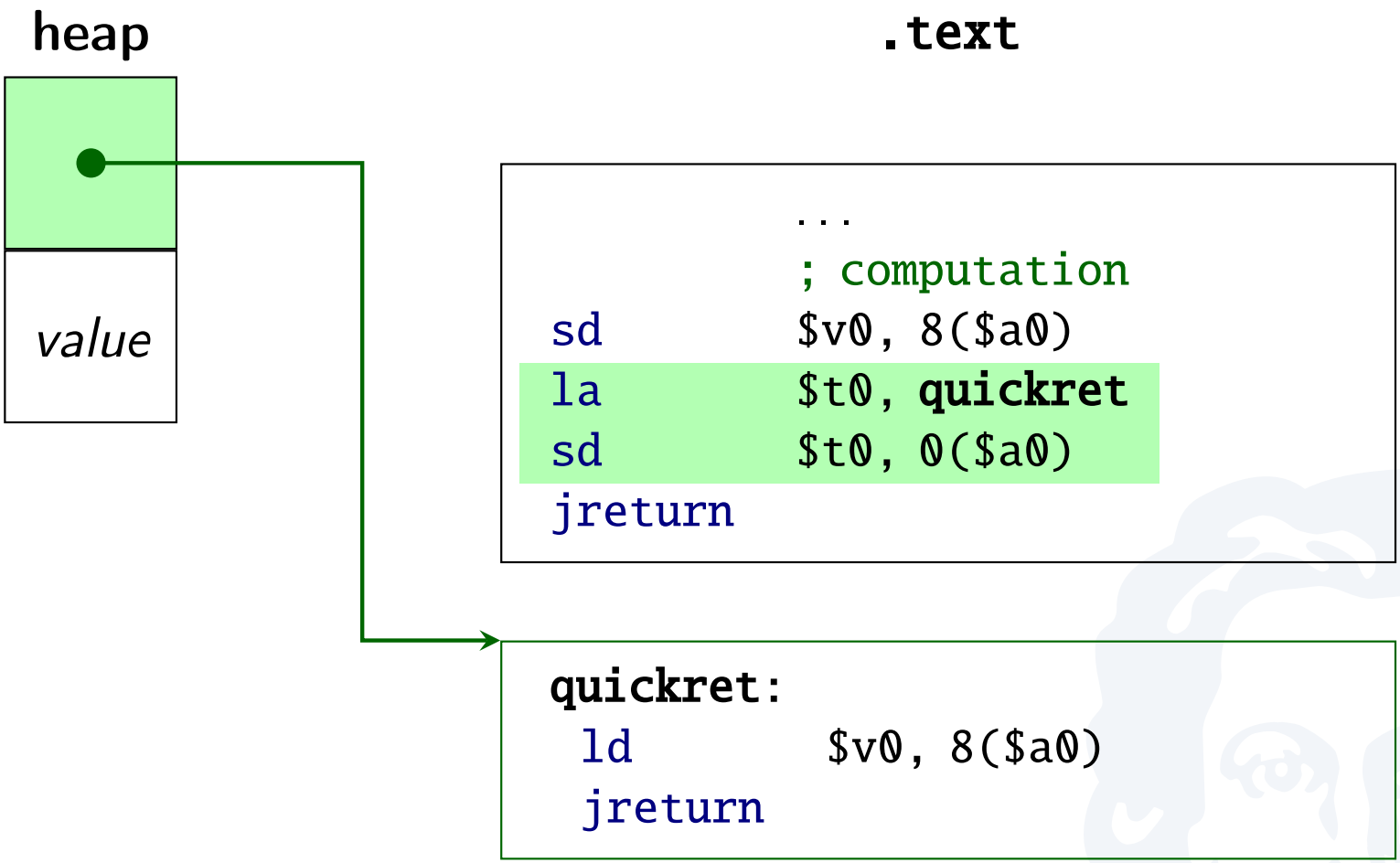
Thunks with Result Cache



Thunks with Result Cache



Thunks with Result Cache



After first evaluation: result stored, code pointer changed to fast read subroutine

Uses of Closures and Thunks

Closures:

- Partial application/currying (functional programming):

```
add 7
```

- Passing subprogram parameters

Thunks:

- Pass-By-Name
- Pass-By-Need (with updates)

Closures in the Wild

Python

```
def add(x):
    def add2(y):
        return x + y
    return add2
```

C++

```
std::function<int(int)> add(int i)
{  return [=] (int j) { return i + j; }; }
```

Java

```
static Function<Integer, Integer>
add(int x)
{ return y -> x + y;  }
```

Summary

- *Closure*: $\langle f, e \rangle$
  - Subroutine pointer f
  - Environment e that captures local variables
- Closures can be used like functions
- Callers must use special operations
- *Thunk*: Closure without parameters
  - May *cache* result
- *Call-by-name*: implemented with uncached thunks
- *Call-by-need*: implemented with cached thunks