Programming Languages in Software Engineering

Lecture 3

Komi Golova (she/her)
komi.golov@jetbrains.com

Constructor University Bremen

Administrative

- Next steps for homework will be announced via Telegram.
- Start reading each others' submissions and thinking of teams. :)

Plan for today

How do we interpret languages?

- Term-rewriting interpreters
- Tree-walk interpreters

Language

Example language: simple LISP


```
(def x 10)
(def f (fun (y) (+ x y)))
(def a (f 5))
(f (block
  (def f2
    (fun (x) (* a (f x))))
  (def b (f2 5))
  (* 3 b)))
```

Example language: simple LISP


```
(def x 10)
(def f (fun (y) (+ x y)))
(def a (f 5))
(f (block
  (def f2
    (fun (x) (* a (f x))))
  (def b (f2 5))
  (* 3 b)))
```

An *S-expression* is:

- A number
- A variable
- A list `(...)`

Example language: simple LISP


```
(def x 10)
(def f (fun (y) (+ x y)))
(def a (f 5))
(f (block
  (def f2
    (fun (x) (* a (f x))))
  (def b (f2 5))
  (* 3 b)))
```

An *S-expression* is:

- A number
- A variable
- A list `(...)`

Language features:

- Immutable variables
- Function (including closures)
- Blocks
- No input/output
  - Result is the last expression

Example language: simple LISP


```
(def x 10)
(def f (fun (y) (+ x y)))
(def a (f 5))
(f (block
  (def f2
    (fun (x) (* a (f x))))
  (def b (f2 5))
  (* 3 b)))
```

An *S-expression* is:

- A number
- A variable
- A list `(...)`

Language features:

- Immutable variables
- Function (including closures)
- Blocks
- No input/output
  - Result is the last expression

How would we evaluate this mentally?

Example language: simple LISP


```
(def x 10)
(def f (fun (y) (+ x y)))
(def a (f 5))
(f (block
  (def f2
    (fun (x) (* a (f x))))
  (def b (f2 5))
  (* 3 b)))
```

An *S-expression* is:

- A number
- A variable
- A list `(...)`

Language features:

- Immutable variables
- Function (including closures)
- Blocks
- No input/output
  - Result is the last expression

How would we evaluate this mentally?

- Try to inline simple definitions.
- Go through the code, remembering earlier results.

A little structure

Not every sequence of S-expressions is meaningful:

```
(fun (x) 10 10)       // two bodies
(def (x y) 10)        // two variables
(def a (def b (10))) // a does not refer to a value
```

A little structure

Not every sequence of S-expressions is meaningful:

```
(fun (x) 10 10)       // two bodies
(def (x y) 10)        // two variables
(def a (def b (10))) // a does not refer to a value
```

Let's be stricter:

- An *expression* is an S-expression with a value.
- A *statement* is a definition or expression.
- A *program* is a sequence of statements ending in an expression.
- A *block* is an expression containing a program.
- A *definition* associates a variable with an expression.
- A *function body* is an expression.

Rewriting interpreters

Term rewriting

Idea: rewrite a program to a “simpler” one.

Term rewriting

Idea: rewrite a program to a “simpler” one.

Example of rules:

$$\mathrm{Add}(x, Z) \rightsquigarrow x$$
$$\mathrm{Add}(x, Sy) \rightsquigarrow S(\mathrm{Add}(x, y))$$

Term rewriting

Idea: rewrite a program to a “simpler” one.

Example of rules:

$$
\begin{aligned}
\mathrm{Add}(x, Z) &\rightsquigarrow x \\
\mathrm{Add}(x, Sy) &\rightsquigarrow S(\mathrm{Add}(x, y))
\end{aligned}
$$

Let’s compute $2 + 2$:

$$
\begin{aligned}
\mathrm{Add}(SSZ, SSZ) &\rightsquigarrow S(\mathrm{Add}(SSZ, SZ)) \\
&\rightsquigarrow SS(\mathrm{Add}(SSZ, Z)) \\
&\rightsquigarrow SSSSZ
\end{aligned}
$$

Our language

We define two rewrite relations: `~>` for programs, `~>E` for expressions.

Rules for programs:

```
(def x e) ss ~> ss[x |-> e]
e ss ~> ss if |ss| > 0 and e is an expression
```

Our language

We define two rewrite relations: `~>` for programs, `~>E` for expressions.

Rules for programs:

```
(def x e) ss ~> ss[x |-> e]
e ss ~> ss if |ss| > 0 and e is an expression

if e ~>E e' then e ~> e'
```

Our language

We define two rewrite relations: `~>` for programs, `~>E` for expressions.

Rules for programs:

```
(def x e) ss ~> ss[x |-> e]
e ss ~> ss if |ss| > 0 and e is an expression

if e ~>E e' then e ~> e'
```

Rules for expressions:

```
((fun (xs) e) es) ~>E e[xs |-> es]
```

Our language

We define two rewrite relations: `~>` for programs, `~>E` for expressions.

Rules for programs:

```
(def x e) ss ~> ss[x |-> e]
e ss ~> ss if |ss| > 0 and e is an expression

if e ~>E e' then e ~> e'
```

Rules for expressions:

```
((fun (xs) e) es) ~>E e[xs |-> es]

if (ss1 ~> ss1') then (block ss1) ~>E (block ss1')
(block e) ~>E e
```

Our language

We define two rewrite relations: `~>` for programs, `~>E` for expressions.

Rules for programs:

```
(def x e) ss ~> ss[x |-> e]
e ss ~> ss if |ss| > 0 and e is an expression

if e ~>E e' then e ~> e'
```

Rules for expressions:

```
((fun (xs) e) es) ~>E e[xs |-> es]

if (ss1 ~> ss1') then (block ss1) ~>E (block ss1')
(block e) ~>E e

if es ~>E es' then (prim-op es) ~>E (prim-op es')
```

Example

```
(def x 10)
(def f (fun (y) (+ x y)))
(f (f 5))
```

Example


```
(def x 10)
(def f (fun (y) (+ x y)))
(f (f 5))
```

```
~>
```

```
(def f (fun (y) (+ 10 y)))
(f (f 5))
```

Example


```
(def x 10)
(def f (fun (y) (+ x y)))
(f (f 5))

              ~>

(def f (fun (y) (+ 10 y)))
(f (f 5))

              ~>

((fun (y) (+ 10 y))
  ((fun (y) (+ 10 y))
    5))
```

Example


```
(def x 10)
(def f (fun (y) (+ x y)))
(f (f 5))

              ~>

(def f (fun (y) (+ 10 y)))
(f (f 5))

              ~>

((fun (y) (+ 10 y))
  ((fun (y) (+ 10 y))
    5))

              ~>

(+ 10 (fun (y) (+ 10 5)))
```

Example


```
(def x 10)
(def f (fun (y) (+ x y)))
(f (f 5))

                ~>

(def f (fun (y) (+ 10 y)))
(f (f 5))

                ~>

((fun (y) (+ 10 y))
  ((fun (y) (+ 10 y))
    5))

                ~>

(+ 10 (fun (y) (+ 10 5)))
```

```
                ~>

(+ 10 (+ 10 5))
```

Example


```
(def x 10)
(def f (fun (y) (+ x y)))
(f (f 5))

~>

(def f (fun (y) (+ 10 y)))
(f (f 5))

~>

((fun (y) (+ 10 y))
  ((fun (y) (+ 10 y))
    5))

~>

(+ 10 (fun (y) (+ 10 5)))

~>

(+ 10 (+ 10 5))

~>

(+ 10 15)
```

Example


```
(def x 10)
(def f (fun (y) (+ x y)))
(f (f 5))

              ~>

(def f (fun (y) (+ 10 y)))
(f (f 5))

              ~>

((fun (y) (+ 10 y))
  ((fun (y) (+ 10 y))
    5))

              ~>

(+ 10 (fun (y) (+ 10 5)))
```

```
              ~>

(+ 10 (+ 10 5))

              ~>

(+ 10 15)

              ~>

25
```

Laziness

This evaluation system is lazy. How do we know?

Laziness

This evaluation system is lazy. How do we know?

```
((fun (x) 0) error-term) ~> 0
```

Formally, a semantics is strict if $f(\bot) = \bot$.

Laziness

This evaluation system is lazy. How do we know?

```
((fun (x) 0) error-term) ~> 0
```

Formally, a semantics is strict if $f(\bot) = \bot$.

To make these semantics strict, we need to distinguish values.

A value is:

- A natural number
- A function

We allow

- Allow evaluating `es` in `((fun (xs) e) es)`
- Only allow substitution (`as[x |-> e]`) when `e` is a value.

Conclusion

We now know what our programs do!

Rewrite semantics are great for formal specification.

Conclusion

We now know what our programs do!

Rewrite semantics are great for formal specification.

Practically speaking:

- Performance is abysmal.
- Side effects need separate handling.
  - Even for specification.

Tree-walking interpreters

Tree-walking

Idea: walk the tree and remember what we already computed.

Tree-walking

Idea: walk the tree and remember what we already computed.

What do we need to maintain?

What operations do we need to support?

Tree-walking

Idea: walk the tree and remember what we already computed.

What do we need to maintain? Values of our variables.

What operations do we need to support?

We can maintain state for the whole program, or just for the part we can see right now.

Tree-walking

Idea: walk the tree and remember what we already computed.

What do we need to maintain? Values of our variables.

What operations do we need to support?

```
interface TreeWalker<LocalState> {
  fun evalProgram(prog: Program, ls: LocalState): Value
  fun evalStatement(stmt: Statement, ls: LocalState)
  fun evalExpression(expr: Expression, ls: LocalState): Value
}
```

We can maintain state for the whole program, or just for the part we can see right now.

State-independent work

There are some things we can solve without worrying about values.

State-independent work

There are some things we can solve without worrying about values.

- Evaluate a program: evaluate all statements, return last expression.

State-independent work

There are some things we can solve without worrying about values.

- Evaluate a program: evaluate all statements, return last expression.
- Evaluate a block: treat contents as a program.

State-independent work

There are some things we can solve without worrying about values.

- Evaluate a program: evaluate all statements, return last expression.
- Evaluate a block: treat contents as a program.
- Evaluate a call: first evaluate all arguments to get their values.

State-independent work

There are some things we can solve without worrying about values.

- Evaluate a program: evaluate all statements, return last expression.
- Evaluate a block: treat contents as a program.
- Evaluate a call: first evaluate all arguments to get their values.
- Evaluate a primitive operation: use the argument values.

State-independent work

There are some things we can solve without worrying about values.

- Evaluate a program: evaluate all statements, return last expression.
- Evaluate a block: treat contents as a program.
- Evaluate a call: first evaluate all arguments to get their values.
- Evaluate a primitive operation: use the argument values.

What's left?

State-independent work

There are some things we can solve without worrying about values.

- Evaluate a program: evaluate all statements, return last expression.
- Evaluate a block: treat contents as a program.
- Evaluate a call: first evaluate all arguments to get their values.
- Evaluate a primitive operation: use the argument values.

What's left? We need to specify how we evaluate:

- The function call itself (jumping and returning)
- Entering and exiting block (`(block e) ~> e`)
- A definition
- A variable lookup
- A function expression (`(fun (x) e) ~> ?`)

Simplification

Let's forget about function calls.

Simplification

Let's forget about function calls.

```
((fun (x) eb) ea) ~> (block (def x ea) eb)
```

Simplification

Let's forget about function calls.

```
((fun (x) eb) ea) ~> (block (def x ea) eb)
```

Okay, so we only need these parts:

```
fun enterBlock()
fun exitBlock()
fun defineVar(x: Variable, v: Value)
fun lookupVar(x: Variable): Value
fun evaluateFunction(f: FunExpression): Value
```

Simplification

Let's forget about function calls.

```
((fun (x) eb) ea) ~> (block (def x ea) eb)
```

Okay, so we only need these parts:

```
fun enterBlock()
fun exitBlock()
fun defineVar(x: Variable, v: Value)
fun lookupVar(x: Variable): Value
fun evaluateFunction(f: FunExpression): Value
```

So far, `evaluateFunction` has been the identity, but we'll see other options have their benefits.

Attempt 1: Hashmap

Idea: let's store a `bindings: HashMap<String, Value>` in `TreeWalker`.

Attempt 1: Hashmap

Idea: let's store a `bindings: HashMap<String, Value>` in `TreeWalker`.

Operations:

- `enterBlock()`: no-op
- `exitBlock()`: no-op

Attempt 1: Hashmap

Idea: let's store a `bindings: HashMap<String, Value>` in `TreeWalker`.

Operations:

- `enterBlock()`: no-op
- `exitBlock()`: no-op
- `defineVar(x, v)`: `bindings.insert(x, v)`
- `lookupVar(x)`: `bindings[x]`
- `evaluateFunction(f)`: `f`

Attempt 1: Hashmap

Idea: let's store a `bindings: HashMap<String, Value>` in `TreeWalker`.

Operations:

- `enterBlock()`: no-op
- `exitBlock()`: no-op
- `defineVar(x, v)`: `bindings.insert(x, v)`
- `lookupVar(x)`: `bindings[x]`
- `evaluateFunction(f)`: `f`

Problem:

```
(def x 5)
(def f (fun (x) x))
rest of the program
```

When we call `f`, we overwrite the value of the global `x`.

Approach 2: Stack of Hashmaps

Idea: `stack: List<Frame>`, each frame is like a hashmap.

Approach 2: Stack of Hashmaps

Idea: `stack: List<Frame>`, each frame is like a hashmap.

Operations:

- `enterBlock()`: push empty frame
- `exitBlock()`: pop top frame

Approach 2: Stack of Hashmaps

Idea: `stack: List<Frame>`, each frame is like a hashmap.

Operations:

- `enterBlock()`: push empty frame
- `exitBlock()`: pop top frame
- `defineVar(x, v)`: add `x |-> v` to top frame
- `lookupVar(x)`: find topmost frame that contains `x`, look it up
- `evaluateFunction(f)`: `f`

Approach 2: Stack of Hashmaps

Idea: `stack: List<Frame>`, each frame is like a hashmap.

Operations:

- `enterBlock()`: push empty frame
- `exitBlock()`: pop top frame
- `defineVar(x, v)`: add `x |-> v` to top frame
- `lookupVar(x)`: find topmost frame that contains `x`, look it up
- `evaluateFunction(f)`: `f`

Now we can have multiple copies of `x`! But…

Problem


```
(def call (fun (x) (x)))
(define x 10)
(call (fun () x))
```

What should this return?

Problem


```
(def call (fun (x) (x)))
(define x 10)
(call (fun () x))
```

```
stack:
{}
{x: fun () x}
{x: 10, call: ...}
```

What should this return? 10

However, searching top-down, we find the wrong x.

Problem

```
(def call (fun (x) (x)))
(define x 10)
(call (fun () x))
```

What should this return? 10

However, searching top-down, we find the wrong `x`.

```
stack:
{}
{x: fun () x}
{x: 10, call: ...}
```

This is why we need evaluate functions differently: we need to view the stack based on where the function was created.

We store an extra value in function values, and add that value as a parameter to `enterBlock` to account for this.
By default, we pass the current stack frame for this parameter.

Example

```
(def f (fun (x) x))
(block
  (def x 5)
  (block
    (f x)))
```

Example


```
(def f (fun (x) x))
(block
  (def x 5)
  (block
    (f x)))
```

Evaluation:

- Evaluate `(fun (x) x)` in global scope
- Define `f` in global scope

Example

```
(def f (fun (x) x))
(block
  (def x 5)
  (block
    (f x)))
```

Evaluation:

- Evaluate `(fun (x) x)` in global scope
- Define `f` in global scope
- Enter block (call it A)
  - Parent: global scope

Example

```
(def f (fun (x) x))
(block
  (def x 5)
  (block
    (f x)))
```

Evaluation:

- Evaluate `(fun (x) x)` in global scope
- Define `f` in global scope
- Enter block (call it A)
  - Parent: global scope
- Define `x` in A

Example

```
(def f (fun (x) x))
(block
  (def x 5)
  (block
    (f x)))
```

Evaluation:

- Evaluate `(fun (x) x)` in global scope
- Define `f` in global scope
- Enter block (call it A)
  - Parent: global scope
- Define `x` in A
- Enter block (call it B)
  - Parent: A

Example


```
(def f (fun (x) x))
(block
  (def x 5)
  (block
    (f x)))
```

Evaluation:

- Evaluate `(fun (x) x)` in global scope
- Define `f` in global scope
- Enter block (call it A)
  - Parent: global scope
- Define `x` in A
- Enter block (call it B)
  - Parent: A
- Lookup `x`
  - Need to look up (A)

Example


```
(def f (fun (x) x))
(block
  (def x 5)
  (block
    (f x)))
```

Evaluation:

- Evaluate `(fun (x) x)` in global scope
- Define `f` in global scope
- Enter block (call it A)
  - Parent: global scope
- Define `x` in A
- Enter block (call it B)
  - Parent: A
- Lookup `x`
  - Need to look up (A)
- Call `f`
- Enter block (call it F)
  - Parent: global scope

Example


```
(def f (fun (x) x))
(block
  (def x 5)
  (block
    (f x)))
```

Evaluation:

- Evaluate `(fun (x) x)` in global scope
- Define `f` in global scope
- Enter block (call it A)
  - Parent: global scope
- Define `x` in A
- Enter block (call it B)
  - Parent: A
- Lookup `x`
  - Need to look up (A)
- Call `f`
- Enter block (call it F)
  - Parent: global scope
- Define `x` in F
- Lookup `x`

Example

```
(def f (fun (x) x))
(block
  (def x 5)
  (block
    (f x)))
```

Evaluation:

- Evaluate `(fun (x) x)` in global scope
- Define `f` in global scope
- Enter block (call it A)
  - Parent: global scope
- Define `x` in A
- Enter block (call it B)
  - Parent: A
- Lookup `x`
  - Need to look up (A)
- Call `f`
- Enter block (call it F)
  - Parent: global scope
- Define `x` in F
- Lookup `x`
- A lot of block exits…

Approach 3: Add an uplink

Idea: `stack: List<Frame>`, `Frame` has an `up` reference.

Approach 3: Add an uplink

Idea: `stack: List<Frame>`, `Frame` has an `up` reference.

Operations:

- `enterBlock(ix)`: push empty frame where `up = ix`.
  - `ix` is an index into `stack`
- `exitBlock()`: pop top frame

Approach 3: Add an uplink

Idea: `stack: List<Frame>`, `Frame` has an `up` reference.

Operations:

- `enterBlock(ix)`: push empty frame where `up = ix`.
  - `ix` is an index into `stack`
- `exitBlock()`: pop top frame
- `defineVar(x, v)`: add `x |-> v` to top frame
- `lookupVar(x)`:
  - find frame containing `x`: try top, then follow `up` references
  - lookup `x` in that frame

Approach 3: Add an uplink

Idea: `stack: List<Frame>`, `Frame` has an `up` reference.

Operations:

- `enterBlock(ix)`: push empty frame where `up = ix`.
  - `ix` is an index into `stack`
- `exitBlock()`: pop top frame
- `defineVar(x, v)`: add `x |-> v` to top frame
- `lookupVar(x)`:
  - find frame containing `x`: try top, then follow `up` references
  - lookup `x` in that frame
- `evaluateFunction(f)`: pair `f` with index of `stack.top()`

Approach 3: Add an uplink

Idea: `stack: List<Frame>`, `Frame` has an `up` reference.

Operations:

- `enterBlock(ix)`: push empty frame where `up = ix`.
  - `ix` is an index into `stack`
- `exitBlock()`: pop top frame
- `defineVar(x, v)`: add `x |-> v` to top frame
- `lookupVar(x)`:
  - find frame containing `x`: try top, then follow `up` references
  - lookup `x` in that frame
- `evaluateFunction(f)`: pair `f` with index of `stack.top()`

We find the correct `x`, if it exists. But…

Problem


```
(def const
  (fun (x)
    (fun (y) x)))
((const 5) 7)
```

We'd expect this to print 5.

Problem


```
(def const
  (fun (x)
    (fun (y) x)))
((const 5) 7)
```

We'd expect this to print 5.

Consider: `((fun (y) x) 7)`

We evaluate this in global scope, where there is no `x`. :(

Problem


```
(def const
  (fun (x)
    (fun (y) x)))
((const 5) 7)
```

We'd expect this to print 5.

Consider: `((fun (y) x) 7)`

We evaluate this in global scope, where there is no `x`. :(

Result: crash.

Problem


```
(def const
  (fun (x)
    (fun (y) x)))
((const 5) 7)
```

We'd expect this to print 5.

Consider: `((fun (y) x) 7)`

We evaluate this in global scope, where there is no `x`. :(

Result: crash.

Our approach if we only allow passing function objects down the stack.
For many purposes this is fine! But sometimes we want more.

Approach 4: Store frames directly

Idea: store a frame locally, each frame stores an optional parent frame.

Approach 4: Store frames directly

Idea: store a frame locally, each frame stores an optional parent frame.

Operations:

- `enterBlock(fr)`: continue with new frame, with `fr` as parent
- `exitFrame()` continue with parent of current frame

Approach 4: Store frames directly

Idea: store a frame locally, each frame stores an optional parent frame.

Operations:

- `enterBlock(fr)`: continue with new frame, with `fr` as parent
- `exitFrame()` continue with parent of current frame
- `defineVar(x, v)`: add `x |-> v` to top frame
- `lookupVar(x)`:
  - find frame containing `x`: try current, then follow parent references
  - lookup `x` in that frame

Approach 4: Store frames directly

Idea: store a frame locally, each frame stores an optional parent frame.

Operations:

- `enterBlock(fr)`: continue with new frame, with `fr` as parent
- `exitFrame()` continue with parent of current frame
- `defineVar(x, v)`: add `x |-> v` to top frame
- `lookupVar(x)`:
  - find frame containing `x`: try current, then follow parent references
  - lookup `x` in that frame
- `evaluateFunction(f)`: pair `f` with current frame

Approach 4: Store frames directly

Idea: store a frame locally, each frame stores an optional parent frame.

Operations:

- `enterBlock(fr)`: continue with new frame, with `fr` as parent
- `exitFrame()` continue with parent of current frame
- `defineVar(x, v)`: add `x |-> v` to top frame
- `lookupVar(x)`:
  - find frame containing `x`: try current, then follow parent references
  - lookup `x` in that frame
- `evaluateFunction(f)`: pair `f` with current frame

Minor problem: captured frames are never garbage collected.

Approach 5: Precompute captures

Idea: store a frame locally, each frame stores an optional parent frame.

Approach 5: Precompute captures

Idea: store a frame locally, each frame stores an optional parent frame.

Same operations as before, except `evaluateFunction(f)`:

- Identify the variables needed by `f`
- Create a new frame `fr` with just those variables
- Pair `f` with `fr`

Approach 5: Precompute captures

Idea: store a frame locally, each frame stores an optional parent frame.

Same operations as before, except `evaluateFunction(f)`:

- Identify the variables needed by `f`
- Create a new frame `fr` with just those variables
- Pair `f` with `fr`

Potential problem: if we add mutation, we won't be able to mutate captures.

Approach 5: Precompute captures

Idea: store a frame locally, each frame stores an optional parent frame.

Same operations as before, except `evaluateFunction(f)`:

- Identify the variables needed by `f`
- Create a new frame `fr` with just those variables
- Pair `f` with `fr`

Potential problem: if we add mutation, we won't be able to mutate captures.

Potential solution: capture by reference.

What happens in practice?

Interpretation style aside - how do languages manage closures?

What happens in practice?

Interpretation style aside - how do languages manage closures?

- Languages with manual memory management are more likely to have precomputed captures.
  - No need for keeping the stack alive.
  - Examples: C++, Rust
- Languages with GC are more likely to maintain frame references.
  - May capture less than a full frame to avoid memory leaks.
  - Examples: Python, C#

Analysis

Tree-walking is a flexible approach.

Analysis

Tree-walking is a flexible approach.

However, due to cache behaviour, it is still slow:

- Walking the tree means jumping around in memory.
- Hashmaps are also spread out in memory.
- Looking up variables by name is slow.

We can make things faster with a bit of compilation.

Bytecode

Idea: figure out what we'll do ahead of time and write it down.

Bytecode

Idea: figure out what we'll do ahead of time and write it down.

Example:

```
(def x (+ (- 5 3) 2))
(* x 3)
```

Our interpreter is recursive:

- Expression results go on the (host) call stack.
- Variables go on the (explicit) stack discussed earlier.

Bytecode

Idea: figure out what we'll do ahead of time and write it down.

Example:

```
(def x (+ (- 5 3) 2))
(* x 3)
```

Our interpreter is recursive:

- Expression results go on the (host) call stack.
- Variables go on the (explicit) stack discussed earlier.

Next week: let's unify those stacks into a single one!