6.001 SICP
Explicit-control evaluator

- Big ideas: how to connect evaluator to machine instructions
  how to achieve tail recursion
- Obfuscation: tightly optimized instruction sequence

- Background
- **`eval-dispatch`** & helpers
- **`define`**, **`if`**, **`begin`**
- Applications

Code example: `sfact`

```
(define sfact (lambda (n prod)
    (display prod)
    (if (= n 1) prod
        (sfact (- n 1) (* n prod)))))
```

- What is displayed when **`(sfact 4 1)`** executes?

  `1 4 12 24`

- What is returned as the value?

  `24`

- Does **`sfact`** describe an iterative or recursive process?

  iterative

Goal: a tail-recursive evaluator

- The stack should not grow if the procedure being evaluated is iterative
  - Most Java, Pascal systems are not tail-recursive, so they cannot use recursive procedures as loops

- Key technique: tail-call optimization
  - If optimization not used, stack grows each time around the loop:

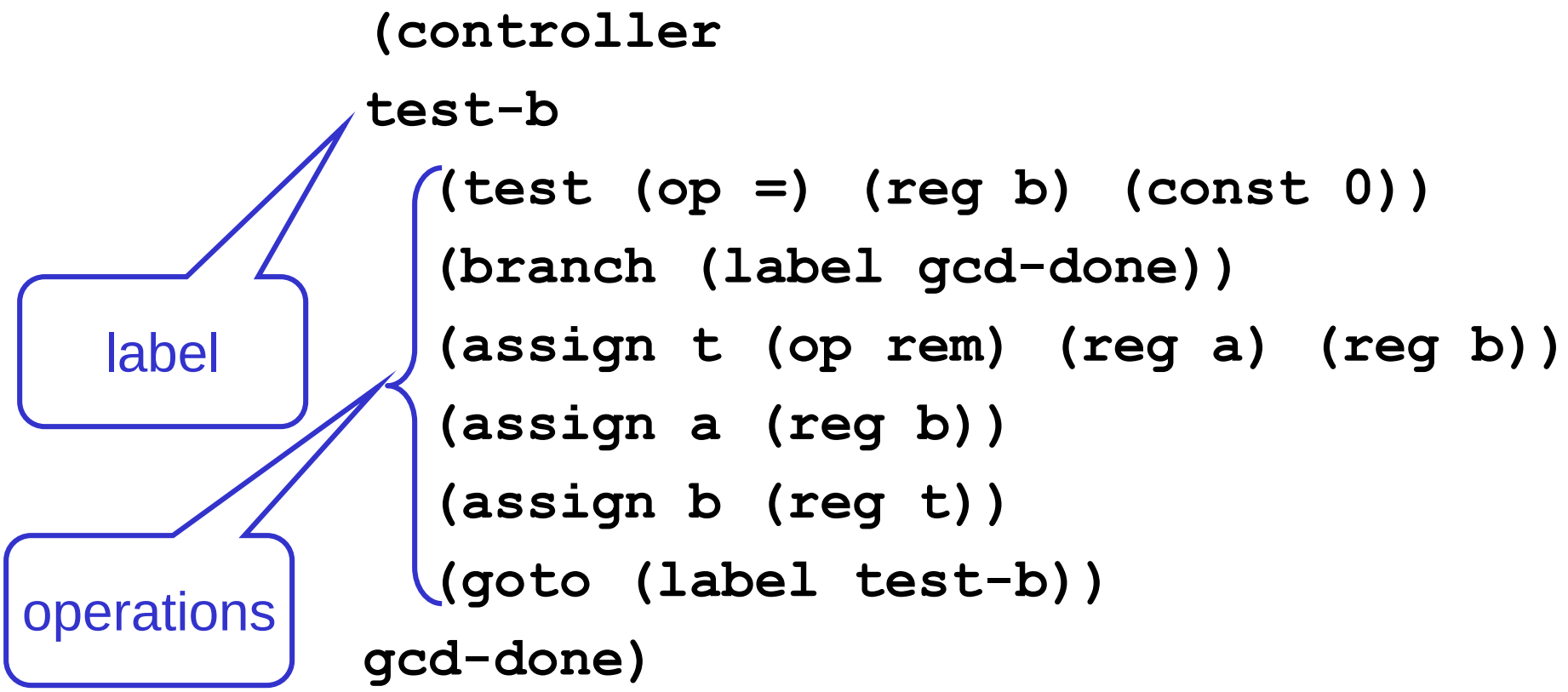
```
(eval-application '(sfact 4 1) GE)
(eval-sequence '((display n) (if ...)) E1)
(eval '(if (= n 1) ...) E1)
(eval-if '(if (= n 1) ...) E1)
(eval '(sfact 3 4) E1)
(eval-application '(sfact 3 4) E1)
```

BOTTOM

TOP

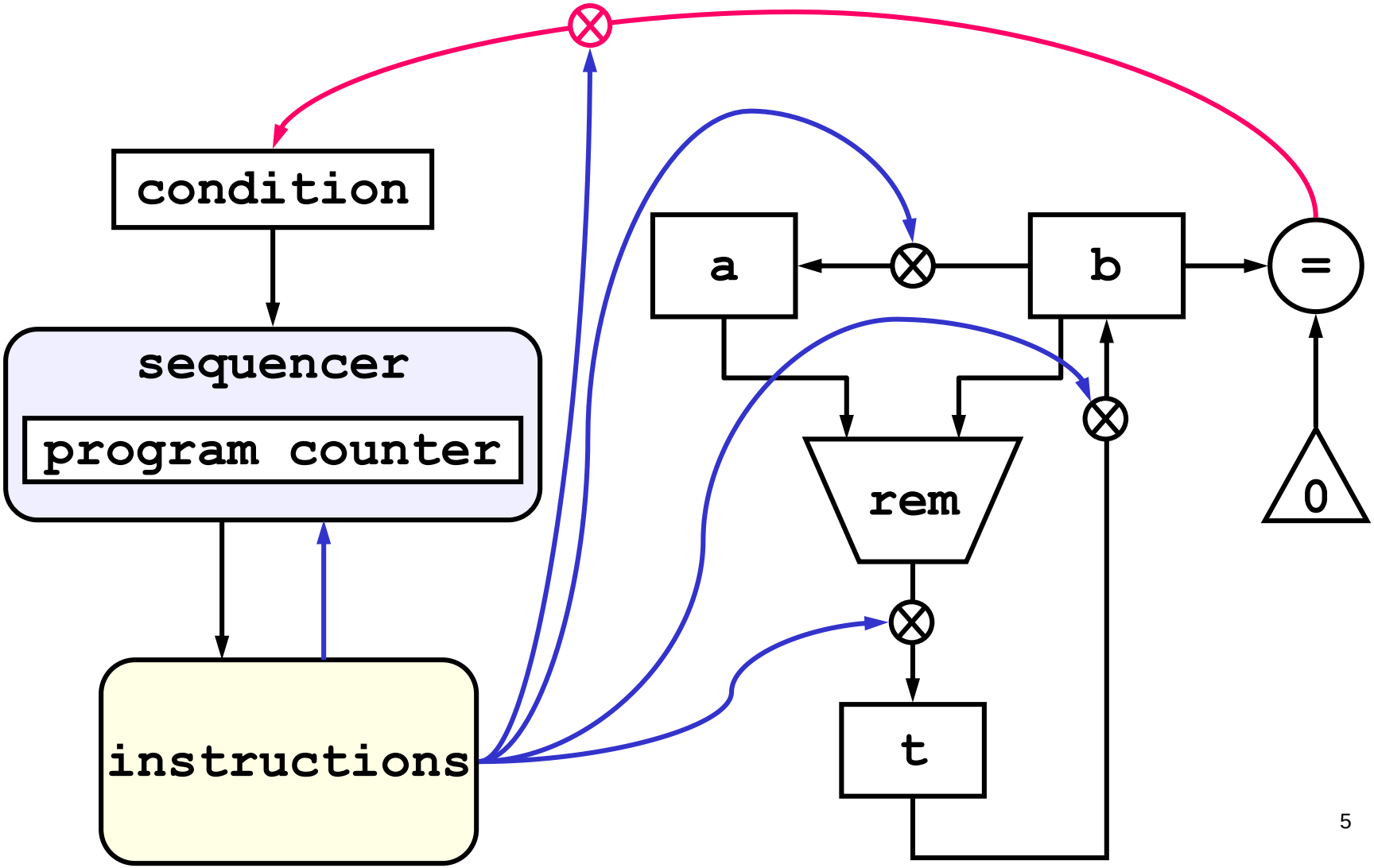
Value needed at start is the same as value returned here

Example register machine: instructions



Register machine

```
sequencer:
   nextPC <- PC + 1
   activate instruction at PC
   PC <- nextPC
   start again
```



Machine for EC-EVAL

- 7 registers

Eval

- exp — expression to be evaluated
- env — current environment
- continue — return point
- val — resulting value
- unev — list of unevaluated expressions (operand lists, sequences)

temporary register (elsewhere)

Apply

- proc — operator value (apply only)
- argl — argument values (apply only)

- Many abstract operations
  - syntax, environment model, primitive procedures

Main entry point: `eval-dispatch`

```
; inputs:   exp        expression to evaluate
;           env        environment
;           continue   return point
; output:   val        value of expression
; writes:   all        (except continue)
; stack:    unchanged

eval-dispatch
   (test (op self-evaluating?) (reg exp))
   (branch (label ev-self-eval))
   (test (op variable?) (reg exp))
   (branch (label ev-variable))
   ...
   (goto (label unknown-expression-type))
```

Eval helpers: same contract as `eval-dispatch`

`ev-self-eval`

```
(assign val (reg exp))
(goto (reg continue))
```

- return value is expression itself

`ev-variable`

```
(assign val (op lookup-variable-value)
            (reg exp) (reg env))
(goto (reg continue))
```

- uses abstract op which is part of environment model

Eval helpers

ev-lambda

```
(assign unev (op lambda-parameters)
             (reg exp))
(assign exp (op lambda-body) (reg exp))
(assign val (op make-procedure)
            (reg unev) (reg exp) (reg env))
(goto (reg continue))
```

- remember our Scheme code for this:

```
(define (eval-lambda exp env)
  (make-procedure (lambda-parameters exp)
                  (lambda-body exp)
                  env))
```

- **exp** and **unev** both used as temporary registers

Recursive call to eval: `ev-definition`

```
ev-definition
  (assign unev (op definition-variable) (reg exp))



  (assign exp (op definition-value) (reg exp))
  (assign continue (label ev-definition-1))
  (goto (label eval-dispatch))
ev-definition-1



  (perform (op define-variable!)
           (reg unev) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue))
```

Ev-definition

- Why are **`unev`**, **`env`**, and **`continue`** saved?
  - Used after recursive call, written by **`eval-dispatch`**

- Why is **`exp`** used in the recursive call?
  - Specified as input register by **`eval-dispatch`** contract

- **`env`** is also specified as an input register, but not assigned
  - Expression of define is evaluated in current environment

- Why is **`unev`** used in line 1?
  - Temporary storage. Could use any other register.

Optimized recursive call to eval: `ev-if`

```
ev-if
   (save exp)
   (save env)
   (save continue)
   (assign continue (label ev-if-decide))
   (assign exp (op if-predicate) (reg exp))
   (goto (label eval-dispatch))
ev-if-decide
   (restore continue)
   (restore env)
   (restore exp)
   (test (op true?) (reg val))
   (branch (label ev-if-consequent))
ev-if-alternative
   (assign exp (op if-alternative) (reg exp))
   (goto (label eval-dispatch))
ev-if-consequent
   (assign exp (op if-consequent) (reg exp))
   (goto (label eval-dispatch))
```

Note – no stack usage for alternative or consequent

ev-if

- Normal recursive call to eval for predicate
- Tail-call optimization in both consequent and alternative
  - no saves or restores
  - this is necessary to make loops like **sfact** iterative

- Alternative case without the optimization:

```
ev-if-alternative
   (save continue)
   (assign continue (label alternative1))
   (assign exp (op if-alternative) (reg exp))
   (goto (label eval-dispatch))
alternative1
   (restore continue)
   (goto (reg continue))
```

Sequences (1)

```
; an eval helper, same contract as eval-dispatch
ev-begin
   (save continue)
   (assign unev (op begin-actions) (reg exp))
   (goto (label ev-sequence))

; ev-sequence:  used by begin and apply (lambda bodies)
;
; inputs:     unev    list of expressions
;             env     environment in which to evaluate
;             stack   top value is return point
; writes:     all     (calls eval without saving)
; output:     val
; stack:      top value removed
```

Sequences (2)

```
ev-sequence
   (assign exp (op first-exp) (reg unev))
   (test (op last-exp?) (reg unev))
   (branch (label ev-sequence-last-exp))
   (save unev)
   (save env)
   (assign continue (label ev-sequence-continue))
   (goto (label eval-dispatch))
ev-sequence-continue
   (restore env)
   (restore unev)
   (assign unev (op rest-exps) (reg unev))
   (goto (label ev-sequence))
ev-sequence-last-exp
   (restore continue)
   (goto (label eval-dispatch))
```

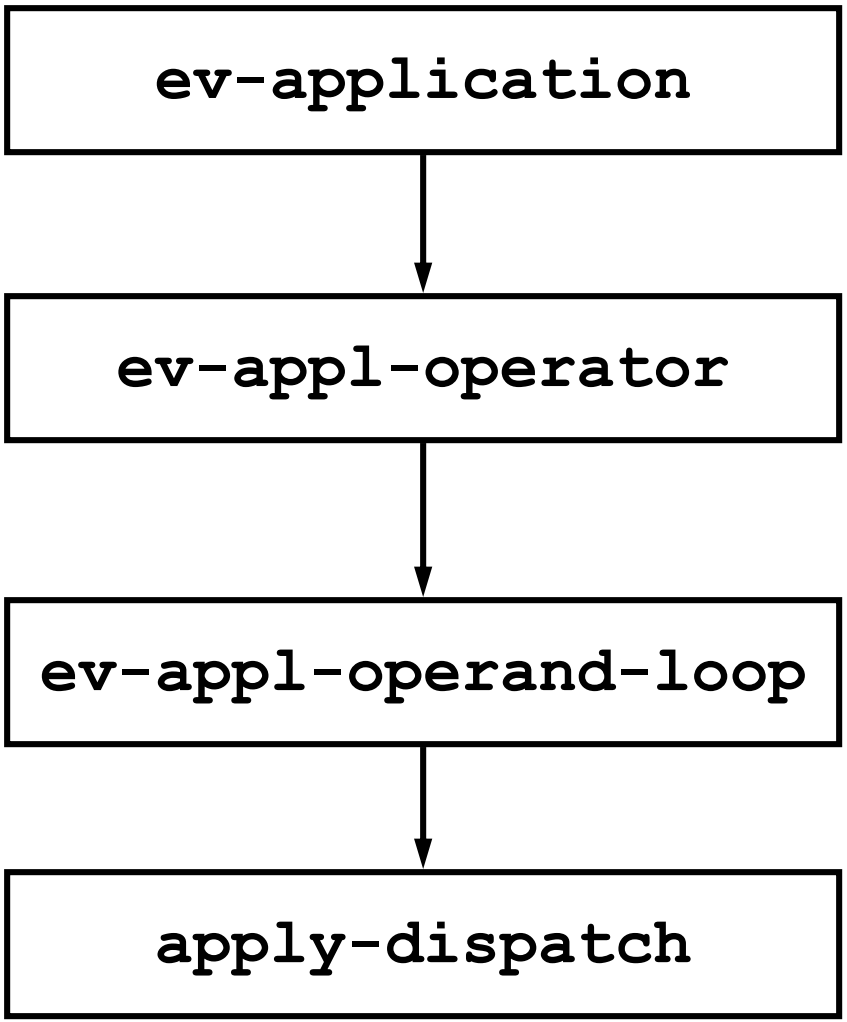
ev-sequence

- Tail-call optimization on eval of last expression in sequence
  - necessary so loops like **`sfact`** are iterative

- Result should be in **`val`**, but never use **`val`**
  - tail call to eval puts final result in **`val`**
  - results of earlier calls to eval are ignored

- Why have return point on top of stack?
  - avoid saving and restoring every time around loop
  - purely a performance optimization **aka – a HACK!**
  - can't do the same with **`unev`** and **`env`** because they are used inside the loop

Applications



eval helper

```
(eval (operator exp) env)
```

```
(map (lambda (e)
             (eval e env))
     (operands exp))
```

```
apply
```

apply-dispatch

```
; inputs:   proc    procedure to be applied
;           argl    list of arguments
;           stack   top value is return point
; writes:   all     (calls ev-sequence)
; output:   val
; stack:    top value removed

apply-dispatch
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-apply))
  (test (op compound-procedure?) (reg proc))
  (branch (label compound-apply))
  (goto (label unknown-procedure-type))
```

Apply helpers

```
primitive-apply
  (assign val (op apply-primitive-procedure) (reg proc)
                                             (reg argl))
  (restore continue)
  (goto (reg continue))

compound-apply
  (assign unev (op procedure-parameters) (reg proc))
  (assign env (op procedure-environment) (reg proc))
  (assign env (op extend-environment)
              (reg unev) (reg argl) (reg env))
  (assign unev (op procedure-body) (reg proc))
  (goto (label ev-sequence))
```

`apply-dispatch`

- Why have return point on top of stack?
  - **`ev-sequence`** needs it on top of stack
  - has to be saved on stack to do **`ev-appl-operator`**
  - performance optimization: leave it on stack if possible

`compound-apply`

- Calls **`ev-sequence`** rather than **`eval-dispatch`**
  - Body of procedure might be a sequence
- Tail-call optimization
  - Necessary for tail recursion
- **`Env`** and **`unev`** used as part of call
  - required by **`ev-sequence`** contract
- **`Env`** and **`unev`** used in first two lines
  - Local temporaries. Could use any register.

ev-application

```
ev-application
   (save continue)

ev-appl-operator
   (assign unev (op operands) (reg exp))
   (save env)
   (save unev)
   (assign exp (op operator) (reg exp))
   (assign continue (label ev-appl-did-operator))
   (goto (label eval-dispatch))
ev-appl-did-operator
   (restore unev)
   (restore env)
   (assign proc (reg val))
```

ev-application

ev-application

- Leave **continue** on the stack, untouched, until
  - **primitive-apply**, OR
  - end of **ev-sequence** of body in **compound-apply**

ev-appl-operator

- Normal call to **eval-dispatch**
  - **unev**: save the list of operand expressions
  - **env**: will be needed to evaluate operand expressions
- At end:
  - Put operator in **proc**. Why use **proc**?
  - Answer: If there are no arguments, will call **apply-dispatch** immediately (next slide)

Map over list of operand expressions

```
   (assign argl (op empty-arglist))
   (test (op no-operands?) (reg unev))
   (branch (label apply-dispatch))
   (save proc)
ev-appl-operand-loop
   (save argl)
   (assign exp (op first-operand) (reg unev))
   (test (op last-operand?) (reg unev))
   (branch (label ev-appl-last-arg))
   ;; eval one operand (next slide)
ev-appl-last-arg
   (assign continue (label ev-appl-accum-last-arg))
   (goto (label eval-dispatch))
ev-appl-accum-last-arg
   (restore argl)
   (assign argl (op adjoin-arg) (reg val) (reg argl))
   (restore proc)
   (goto (label apply-dispatch))
```

Eval one operand

```
   (save env)
   (save unev)
   (assign continue (label ev-appl-accumulate-arg))
   (goto (label eval-dispatch))
ev-appl-accumulate-arg
   (restore unev)
   (restore env)
   (restore argl)
   (assign argl (op adjoin-arg) (reg val) (reg argl))
   (assign unev (op rest-operands) (reg unev))
   (goto (label ev-appl-operand-loop))
```

`ev-appl-operand-loop`

- First three lines:
  - check for no operands (avoid **`first-operand`** on empty)
- Why save **`proc`** at beginning, restore at very end?
  - call eval in loop, its contract says it writes **`proc`**
  - one of the operand expressions might be an application
- Same reasoning applies to **`argl`**
- Why save **`argl`** inside the loop, **`proc`** outside it?
  - need to change **`argl`** every time around the loop
- Why is **`(save argl)`** before the branch to ev-appl-last-arg?
  - logically goes with the saves in eval one operand
  - a needless optimization that saves one instruction

Trial simulation

Label	Exp	Env	Val	Proc	Argl	Unev	Cont	Stack
Eval	(fact 3)	GE					REP	
Eval	fact	GE				(3)	didop	REP GE (3)
Didop	fact	GE	[proc]			(3)	didop	REP GE (3)
Oploop	fact	GE	[proc]	[proc]	()	(3)	didop	REP [proc]
Lastarg	3	GE	[proc]	[proc]	()	(3)	didop	REP [proc] ()
Eval	3	GE	[proc]	[proc]	()	(3)	a-l-a	REP [proc] ()
A-l-a	3	GE	3	[proc]	()	(3)	a-l-a	REP [proc] ()
Apply	3	GE	3	[proc]	(3)	(3)	a-l-a	REP
Seq	3	E1	3	[proc]	(3)	((if.))	a-l-a	REP
Seqlst	(if..)	E1	3	[proc]	(3)	((if.))	a-l-a	REP
Eval	(if..)	E1	3	[proc]	(3)	((if.))	REP	

Trial simulation

Label	Exp	Env	Val	Proc	Argl	Unev	Cont	Stack
Eval	(fact 3)	GE					REP	
…skip some steps…								
Eval	(if..)	E1	3	[proc]	(3)	((if.))	REP	
Eval	(= n 1)	E1	3	[proc]	(3)	((if.))	dec	(if.) E1 REP
…skip some steps – contract says that when get to decide we have…								
Dec	(= n 1)	E1	#f	[proc]	(3)	((if.))	dec	(if.) E1 REP
Eval	(* n (f.))	E1	#f	[proc]	(3)	((if.))	REP	
Eval	*	E1	#f	[proc]	(3)	(n (f.))	did	REP E1 (n (f.))
Did	*	E1	[mul]	[proc]	(3)	(n (f.))	did	REP E1 (n (f.))
Oploop	*	E1	[mul]	[mul]	()	(n (f.))	did	REP [mul]
…skip some steps – just look up value of n, then get to…								
Eval	(f..)	E1		[mul]	(3)	((f.))	a-l-a	REP [mul] (3)

Trial simulation

Label	Exp	Env	Val	Proc	Argl	Unev	Cont	Stack
Eval	(fact 3)	GE					REP	
…skip some steps…								
Eval	(fact (- n 1))	E1					a-l-a	REP [mul] (3)
…skip some steps – by contract, know that we get to…								
Eval	(fact (- n 1))	E2					a-l-a	REP [mul] (3) a-l-a [mul] (2)

Summary

- Have seen details of EC-EVAL
- Differentiated
  - necessary optimizations for tail recursion
  - performance optimizations

- Key idea is that we can connect evaluation through a machine model to support idea of universal evaluation