Inlining removes fun calls

? stack space (for non-tail calls)

+ removes jumps (ret is an indirect, to direct)

+ no caller/ee var saving

− fun grows, big fun ⇒ lots of registers ⇒ register pressure

− code size hurts i-cache
(matters most in loop bodies)

+ exposes other optimizations

```
(define (fx) (if x 42 99))
(+ (+ (f #t) (f #f)) (fy))
```

− code size makes compile longer

−
```
(define (fac n)
  (if (<= n 0) 1 (* n (fac (sub1 n)))))
(fac 5)
```

Program is a set

Optimize$_i$: Prog ⇒ Prog

P_0 is user-input

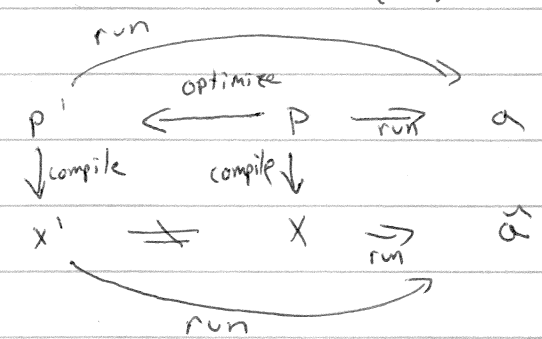
$O_* = \text{Compiler}_* (P_0) = \text{Optimize}_3 (O_2 (O_1 (O_0\ P_0)))$

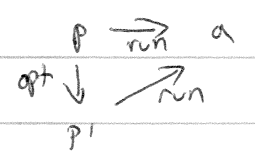
$\text{Compiler}_F (P) = P_x$

sort $O_* (P_x) = P_x$

$\hat{O}_* (P) = P_x$

accurate





run : prog ⇒ ans

opt(p) = p

run' : prog ⇒ ans, cost

8-2/ guarantee linear time

polyvariant — every fun call is inlined independently
monovariant — funs are inlined, not fun calls

online — fast enough to be interleaved w/ rest of compile

context-sensitive — way the fun result is used changes the optimization

```
(≤ (f x) 7)
(vector-ref (f x) 3)
```

boolean, effect, value

demand-driven — function bodies are not optimized until called

"flow-insensitive"

```
(define V (vector add1))   (let ([old (vr V 0)])
(define (flip! X)            (vector-set! V 0
(vector-set! V 0 (if X add1 sub1)))     (λ x. (+ 1 (old x)))))
(flip! t) (flip! f) (flip t) (flip t)
((vector-ref V 0) 7)
```

```
e ::= (const c) | (ref x) | (primref p)
    | (if e e e) | (seq e e) | (assign x e)
    | (λ (x) e) | (letrec ([x e] ...) e)
    | (call e e)                    x ∈ Var
```

I : Expression e → Context γ → Env ρ → Continuation κ → Store σ → e

ρ ∈ Env = Var → Var Var := (Id × Operand⊥, VarFlags, Loc_x)

κ ∈ Cont = Exp → Store → Exp

σ ∈ Store = (Loc_x → VarFlags) × (Loc_γ → ContextFlags) × (Loc_e → Exp⊥)

VarFlags ⊆ {ref, assign} ContextFlags ⊆ {inlined}

γ ∈ Context = Test | Effect | Value | App(Operand, γ, Loc_γ)

Operand = Opnd(Exp, Env, Loc_e)