27-1

parsing : string ⇒ tree of a program

"return 1 + 2 * 3;"

⇒ return → + → (1, * → (2, 3))

flex/yacc
antlr
packrat-parsing
parser combinator library

27-2 / bytecode

compiler : expr → binary_format

cek : expr → ans

cek.c

void cek_eval (expr_t * o)

hl : expr → c-program

(+ 1 2) ⇒ #include "cek.c"

```
void main() {
  cek_eval (make_add(make_num(1)
                     make_num(2))
```

27-3

~~Stop~~ cc : C → x86

pentium : x86 → ans

jc : j ⟷ jb

jum : jb → ans

"return 1 + 2 * 3" → "(+ 1 (* 2 3))"

0 → Num (1)   4 → +

1 → App (2)   5 → *

2 → Empty (0)

3 → Cons (2)

↓ 1 4 3 0 1 3 1 5 3 0 2 3 0 3

2 2

↓ [1 3 □] 4

27-4/

hl: - parse, desugar, emit ll code

(optimization — bracketed under "desugar, emit")

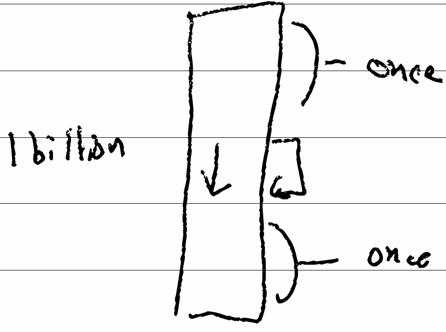
e_1 into e_2 iff $\forall C.\ C[e_1] \equiv C[e_2]$
$\wedge\ m(e_1) < m(e_2)$

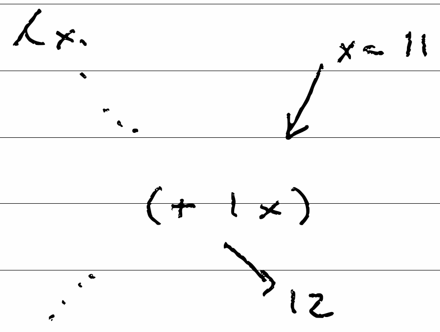
```
(+ 1 (x 2 3))  →  7
(+ 1 (+ 3 z))  →  (+ 4 z)
opt (Add (Num x) (Add (Num y) e))
   = (Add (Num (x + y)) (opt e)))
```

27-5

```
opt (Add lhs rhs) =
  let lhs' = opt lhs
      rhs' = opt rhs in
  case lhs' of
      Num 0 -> rhs'
      Num x -> case rhs' of
                 Num y -> Num (x + y)
                 Add (Num y) z -> Add (Num (x+y)) z
                 _ -> Add lhs' rhs'
    _ -> case rhs' of
           Num 0 -> lhs'
           Num y -> Add (Num y) lhs'
```

27-6 / JIT – a just in time compiler



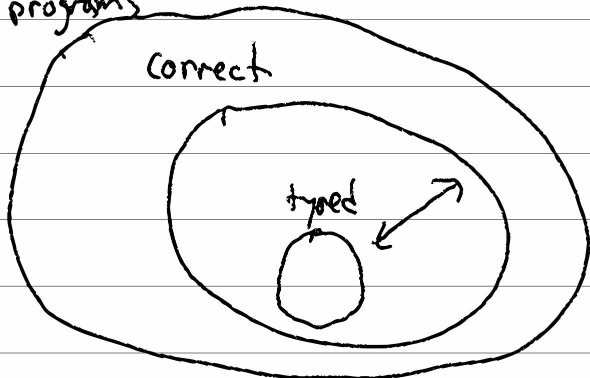


27-7/

editors
syntax highlighting
packages
documentation

software engineering

27-8/ more advanced type system



Haskell

↳ type engineering

type classes

Dependent types

via like Coq or

Agda

: any ← JS, Py, Ruby

27-9 | sort : fun

fun 2 → 1

↙ C/Pascal

List × (Arg × Arg → Bool) → List

↙ Haskell

∀X. List<X> × (X × X → Bool) → List<X>

↙ Coq

∀X. (il : List<X>) × (lt : X × X → Bool)
→ { ol : List<X> | Permutation il ol
∧ Ordered lt ol }

& runs in O(n lg n)