Meeting 11: Data Types and Polymorphism

IF ANYONE HITS ME WITH A SNOWBALL, I'LL HIT *HIM* WITH 250 SNOWBALLS!
WHAT IF SOMEBODY HITS *YOU* WITH 250 SNOWBALLS?
..SIGHHH..

Announcements

HW4 due Friday 10/13

HW3

- mean 13.0 hours, stdev 6.1; median 12 hours
- 4.6 hardness mean, 5 hardness median

Comments

- The most difficult part of this assignment was understanding how to fit the abstract presentation of the strategies and algorithms in the notes into the concrete situation in our compiler.

- The hardest part for me was figuring out how to keep track of available registers during the DSATUR algorithm. I also had to do some refactoring to get the x86 IR, which added a bit of time to my effort.

- Our data structures got a bit out of hand. We definitely could have been more creative/clever and avoided unneeded bloat.

- No parts were particularly hard, but there were a lot of them, and we had to get all of them right at the same time.

    - I think a few more tips on how x86IR should be designed would be nice.

- My code needed significant refactoring to handle the scanning and manipulation required in the third homework assignment. Once you have a very flexible representation of your assembly and understand the chapter things started to fall into place.

- This was surprisingly easier than I expected.

- Because spill code is only generated in large(ish) programs, it was hard to find out what was going wrong without sorting through heaps of code. Debugging took forever. Aside from technical difficulties, the theory and motivation was pretty easy.

- It's really hard to write unit tests!

- I believe including more explicit notes about the importance of choosing the x86 IR representation carefully, and the specific ways the end-to-end flow changes the original HW1 implementation would have alleviated some initial frustration and false starts. In particular, the system at the end of chapter 3 replaces some pieces of the system described in chapter 1, but I didn't know up front I needed to replace those pieces.

Process of designing / thinking about an IR

- What is output — x86

- How easy to maintain — avoid copy-and-paste

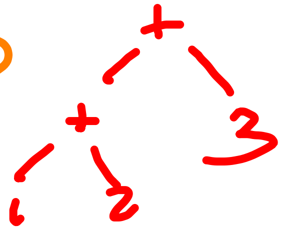
↓ abstract / modularize

→ An AST language is represents the structure of a PL

→ What is the structure of that language

Recursive tree structure Python P0

vs

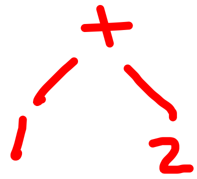


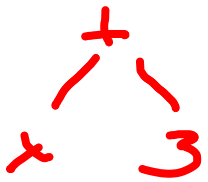
x86 with variables

registers, limited add x, y

vs flat P0/Python

unlimited variables $\underline{1+3}$





$e ::= \underbrace{n} \mid \underbrace{e + e} \mid x$ Python-esque AST

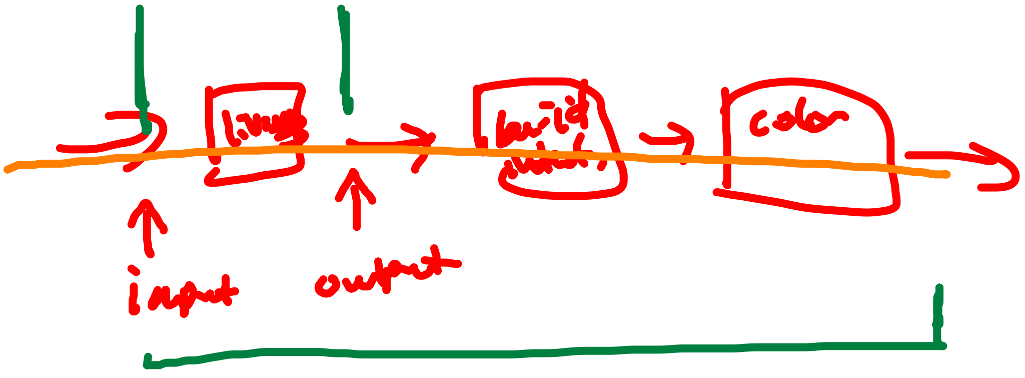
$a ::= n \mid x$

$e ::= a \mid a + a$

```
class Atomic
    pass

class Num(Atomic):
    ≡

class Var(Atomic):
    def __init__(self, x):
        self.x = x
```



assert expected = liveness(input)

unit
↓
more complicated

"growing the language"

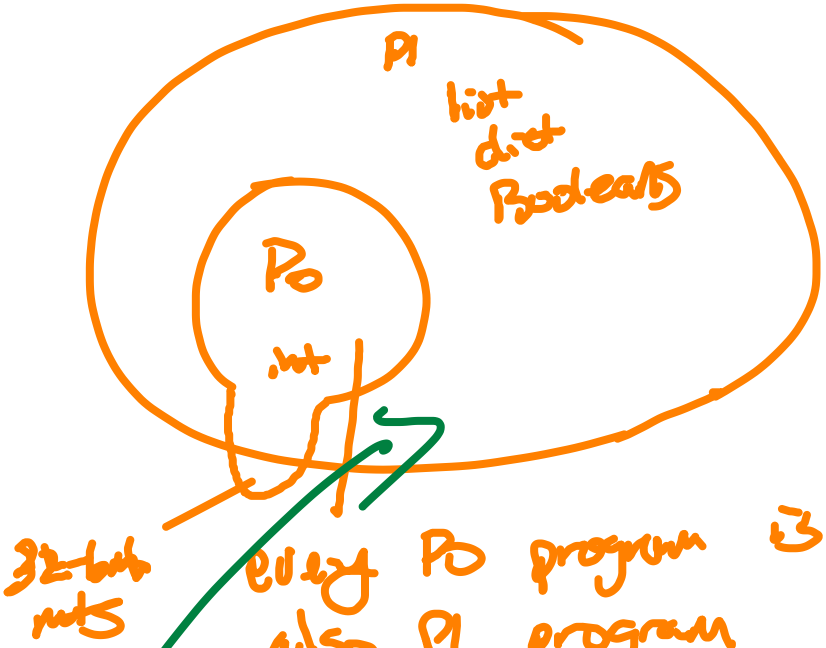
$$\frac{3}{x = 3}$$

Questions

① O(n lg n) is about coloring]

② Explicit

HW4



every P0 program is also P1 program

ad-hoc polymorphism

- P1 code that applies and does different things depending on the run-time type

x + y

int + int add

3 + 4

list + list

[3] + [4] = [3,4]

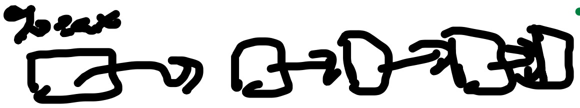
~~1 + [2,3]~~ not in P1 — no conversions but ad-hoc polymorphism

What needs to change in P1?

① variable x could store an int, list, dict, bool

→ compilation: dispatch based on the run-time type

② if x holds a list, the whole list cannot register — value of a list is the address of a heap-allo list



③ need encode the run-time type information

— tags = implement sum type

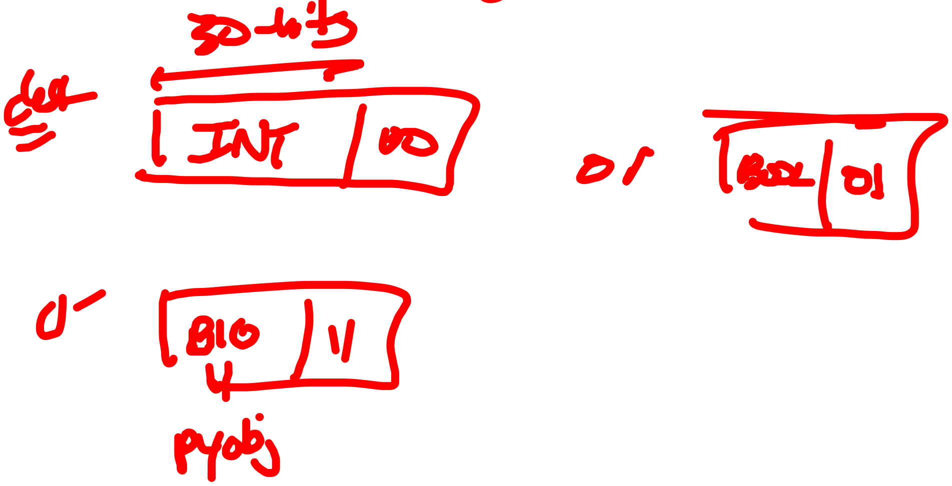
```
typedef struct {
    kind tag;
    union {
        list l
        dict d
    }
} pyobj;
```

← points to "big things"

```
typedef enum {
~~INT,~~ LIST, DICT

} kind;
```

pyval def


30 bits
INT
00
or
BOOL
01
or
BIG
11
pyobj