

Synthesis

Reading Reflection

Discuss in groups

- If you could express your intent to the computer in any way at all, how would you want to write programs?
 - What input would you have the computer take?
 - How would the interaction between you and the computer work?
- What was confusing about synthesis from the first reading/your understanding of synthesis so far?
 - It's ok if this is lots of things! We'll be getting hands-on soon, which should clear up a lot of confusions. :)
- Are there applications that you'd expect are amenable to synthesis but that haven't made it into the literature yet? (Weren't mentioned in Chapter 2.)

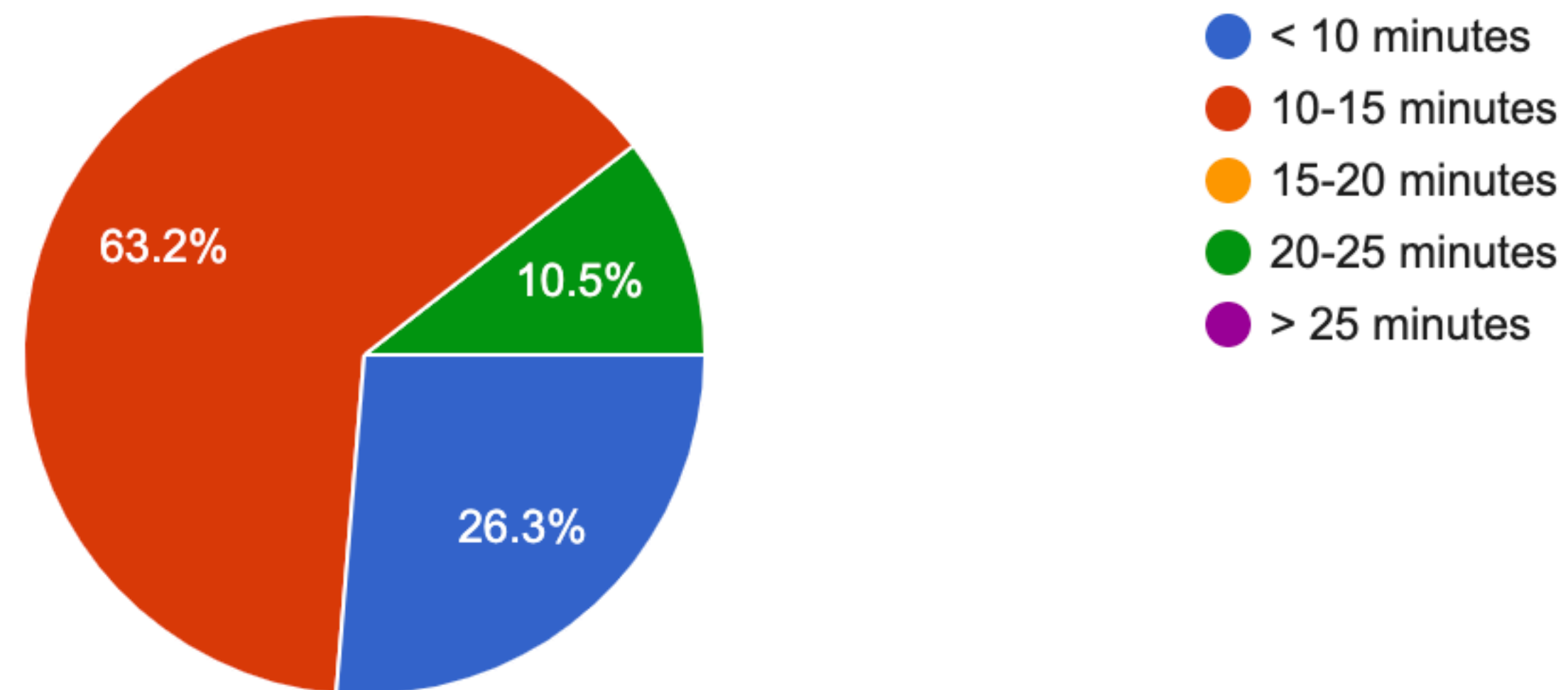
Reading Key Takeaways

- The core challenges in synthesis:
 - Scalability/size of the program space
 - Capturing user intent—What's a good spec? How do we get it?
- The variety of plausible specs we can get from users
 - I/O examples, demonstrations, logical specs, natural language, programs with holes, equivalent programs (!)
- The variety of search techniques
 - Enumerative, constraint-based, deductive, statistical
 - And at a higher level, the fact that synthesis is not just **one** technique
- A general sense of the problems to which synthesis has been applied so far

Thank you for your survey
answers!

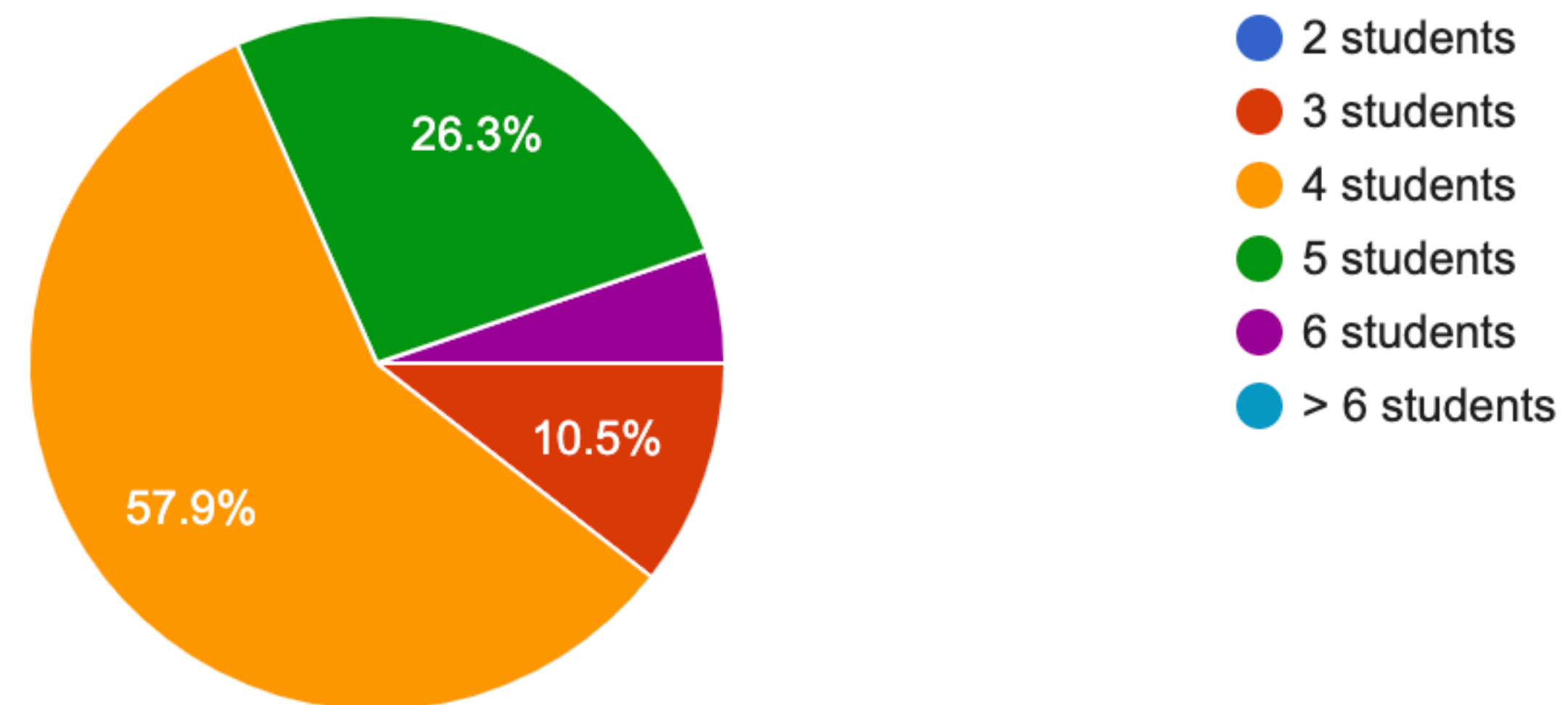
How much time should we spend in the reading discussion breakout rooms?

19 responses



How big should the reading discussion breakout rooms be?

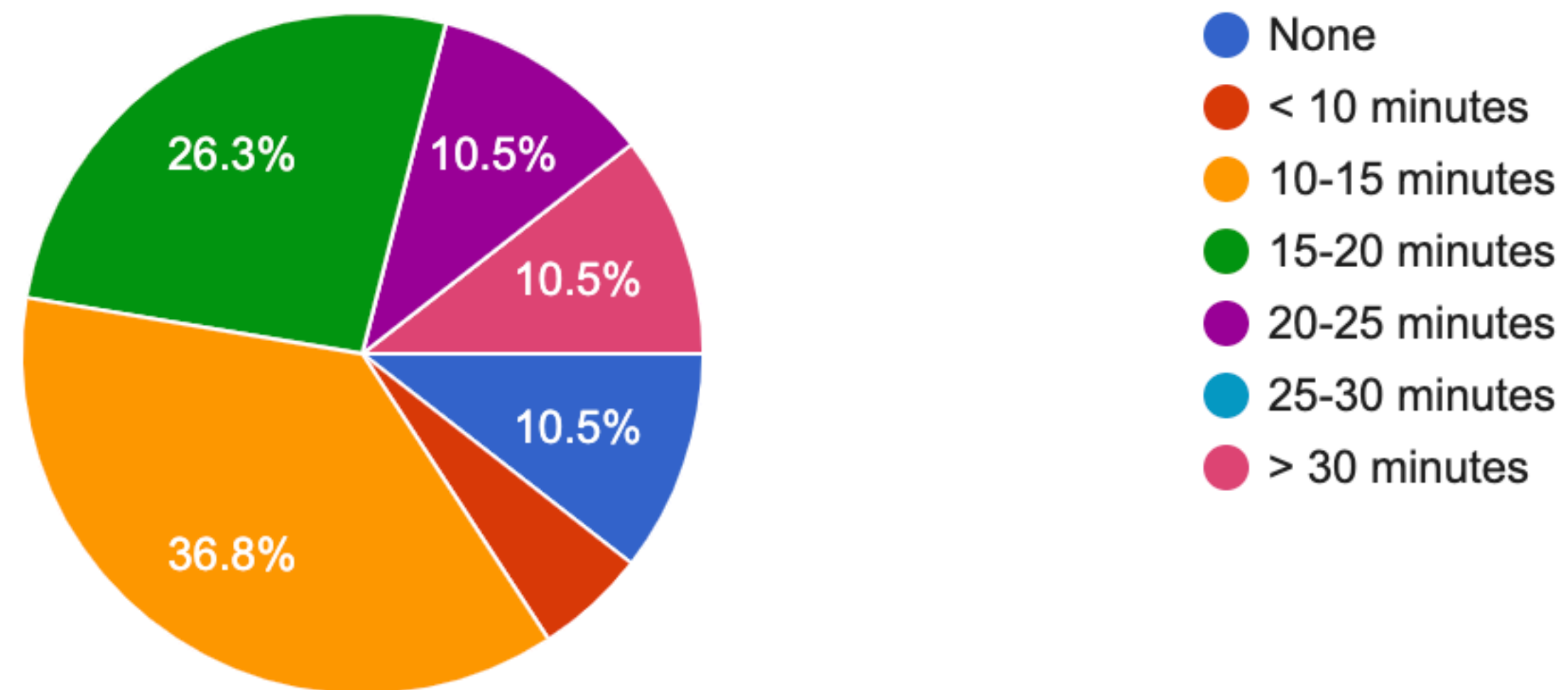
19 responses



I forgot to ask if it's ok that I "walk around" during group discussion time (come visit breakout rooms). Feel free to let me know if you have strong feelings about this.

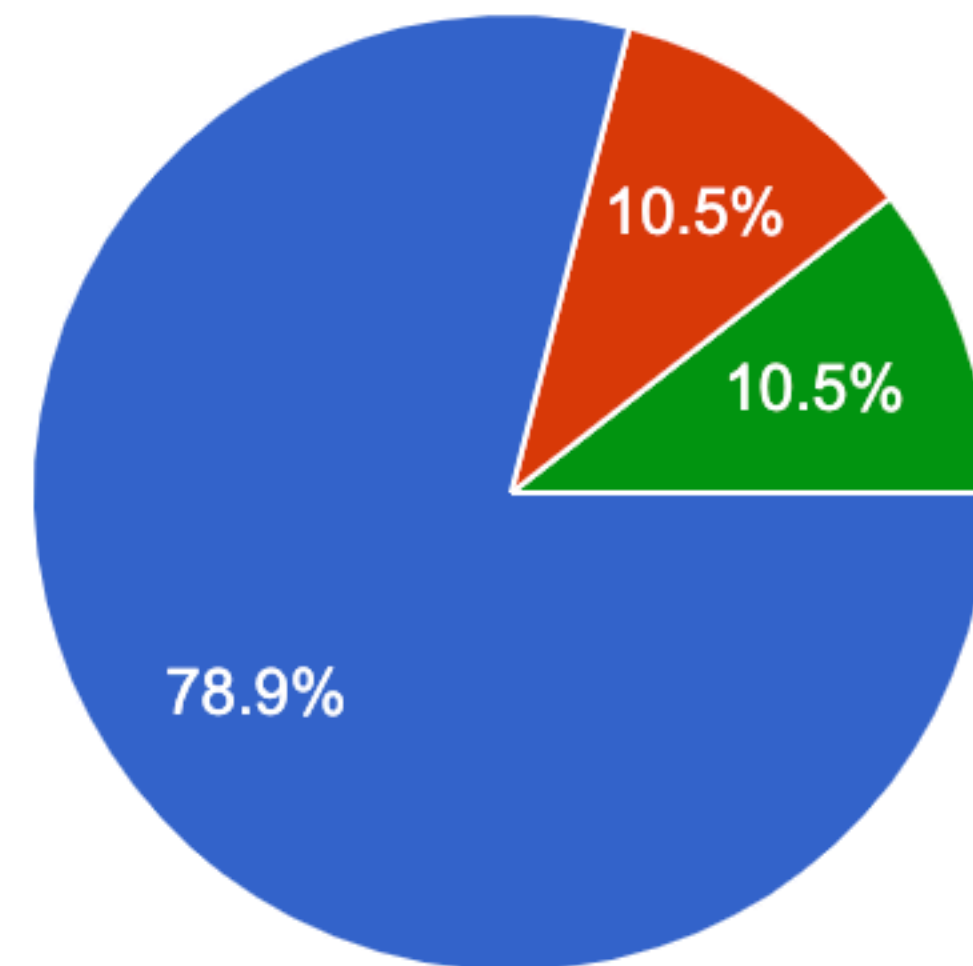
How much time should we spend in the assignment work breakout rooms?

19 responses



How do you feel about the mini breaks in the middle?

19 responses



- Please keep. I need coffee/water/just a break from staring into the zoom void/whatever.
- Keep the breaks, but only about 5 minutes.
- Don't need 'em!
- Keep the breaks, but only on days when we've been sitting passively/listening t...
- Keep the breaks, but only on days when we've been active/doing activities.

Reading Discussion Format

- Seems like folks are really enjoying the breakout discussion groups (awesome!)
- Also liking the whole-group processing after the breakout discussions, want even more time for this
- (Sorry I didn't ask about the time allotted to that!)
- So we'll be expanding the part where we come together to synthesize what we discussed in the breakout rooms
- Also wanted a little more summary from me, so I'll be adding that

Other changes

- I'll stop yanking you out of breakout rooms like that! The 60 second callback will return.
 - This was the number 1 proposed change. Sorry!!
- Slide speed.
 - I'll slow it down. Feel free to send feedback through the semester.
- Assignments building on each other.
 - This was unpopular. :) Future assignments will be standalone.
- Prepping for future readings.
 - I'll try to end each class with some ideas of what to look for in the next day's reading.

Why synthesis?



People doing HCI
stuff...

Synthesis

There are a few PL techniques that just keep coming up in HCI tasks!

- Program synthesis
- Projection/Structure editors
- Program slicing

Others come up, but these seem to come up all the time.

Demo time

FlashFill

Do you have Excel installed? You can probably run this demo on your own laptop while I run it on mine!

Scythe

To run this one, head to: <https://scythe.cs.washington.edu/demo>

Helena

If you want to run this one, you have to install an extension: <http://helena-lang.org/install>

LENS

before		after	
cmp	r1, #0		
mov	r3, r1, asr #31	asr	r3, r1, #2
add	r2, r1, #7	add	r2, r1, r3, lsr #29
mov	r3, r3, lsr #29	ldrb	r0, [r0, r2, asr #3]
movge	r2, r1	and	r3, r2, #248
ldrb	r0, [r0, r2, asr #3]	sub	r3, r1, r3
bic	r1, r2, #248	asr	r1, r0, r3
sub	r3, r1, r3	and	r0, r1, #1
asr	r1, r0, r3		
and	r0, r1, #1		
(b)		(c)	

I know, I know, not as photogenic, but it makes programs much faster!!

Falx

Coming soon to <https://falx.cs.washington.edu/>

5 min break

Back up. What's program synthesis?

Find a program P that meets a spec $\phi(\text{input}, \text{output})$:

Correctness Condition

$$\exists P. \forall x. \phi(x, P(x))$$

Find P

- When to use synthesis:
 - **Ease-of-use/productivity:** When writing ϕ is faster or easier than writing P
 - **Correctness:** when proving ϕ is easier than proving P

Hey, I've seen this before

I give computer a high-level
description of what I want it to do

Computer gives me back a low-
level program for doing it



Synthesis vs. compilation

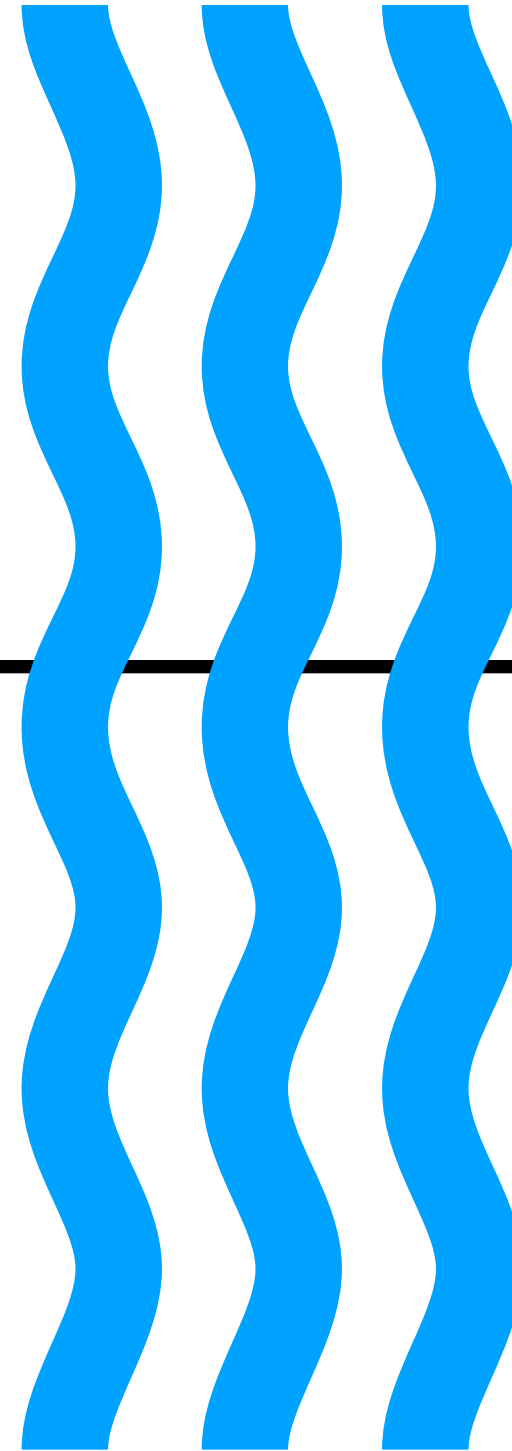
Compilation

Synthesis



Typically deterministic

Typically performs lowering
via a sequence of rewrite
rules



Searches a space of
possible programs

...or sometimes a space of
possible sequences of
rewrite rules! look, the line
is blurry $\backslash_(\text{ツ})_/_$

If it involves search, we
usually call it synthesis

Even if you don't take away anything else from today's lecture, take away that you can write a synthesizer!

Even if you don't take away anything else from today's lecture, take away that you can write a synthesizer!



What do we need to decide to make a synthesizer?

Hint: 3 things

How does the user express what they want the program to do?

What space of programs is the synthesizer allowed to use?

What algorithm will the synthesizer use to search that space?

What do we need to decide to make a synthesizer?

Hint: 3 things For today's sample synthesizer, let's pick...

How does the user express what they want the program to do?

Input-Output examples

What space of programs is the synthesizer allowed to use?

Anything in a Domain-Specific Language (DSL) of our choice

What algorithm will the synthesizer use to search that space?



Enumeration

Which is to say...generating programs until we find one that works

Input-Output Examples

- Any work here?
- Nah, this is going to be pretty straightforward.

- Example:

$(\{ "x" \rightarrow 3, "y" \rightarrow 7 \}, 23)$

$(\{ "x" \rightarrow 4, "y" \rightarrow 4 \}, 19)$

$(\{ "x" \rightarrow 2, "y" \rightarrow 12 \}, 31)$

**Can you guess it?? Did you already
synthesize this in your head?**

Domain-Specific Language

- This one's a classic, but for another domain we might design something more customized

$$\begin{array}{l} \textit{expr} := \mathcal{N} \\ \quad | \textit{v} \\ \quad | (\textit{expr} + \textit{expr}) \\ \quad | (\textit{expr} - \textit{expr}) \\ \quad | (\textit{expr} * \textit{expr}) \end{array}$$

Enumeration

Spec:

({ "x" → 3, "y" → 7 } , 23)
({ "x" → 4, "y" → 4 } , 19)
({ "x" → 2, "y" → 12 } , 31)

Space of programs:

$expr := \mathcal{N}$
| v
| $(expr + expr)$
| $(expr - expr)$
| $(expr * expr)$

level 0:

[0, 1, 2, 3, 4, y, x]

count: 7

level 1 :

[0, 1, 2, 3, 4, y, x, (0+0), (0*0), (0-0), (0+1), (0*1), (0-1), (0+2), (0*2), (0-2), (0+3), (0*3), (0-3), (0+4), (0*4), (0-4), (0+y), (0*y), (0-y), (0+x), (0*x), (0-x), (1+0), (1*0), (1-0), (1+1), (1*1), (1-1), (1+2), (1*2), (1-2), (1+3), (1*3), (1-3), (1+4), (1*4), (1-4), (1+y), (1*y), (1-y), (1+x), (1*x), (1-x), (2+0), (2*0), (2-0), (2+1), (2*1), (2-1), (2+2), (2*2), (2-2), (2+3), (2*3), (2-3), (2+4), (2*4), (2-4), (2+y), (2*y), (2-y), (2+x), (2*x), (2-x), (3+0), (3*0), (3-0), (3+1), (3*1), (3-1), (3+2), (3*2), (3-2), (3+3), (3*3), (3-3), (3+4), (3*4), (3-4), (3+y), (3*y), (3-y), (3+x), (3*x), (3-x), (4+0), (4*0), (4-0), (4+1), (4*1), (4-1), (4+2), (4*2), (4-2), (4+3), (4*3), (4-3), (4+4), (4*4), (4-4), (4+y), (4*y), (4-y), (4+x), (4*x), (4-x), (y+0), (y*0), (y-0), (y+1), (y*1), (y-1), (y+2), (y*2), (y-2), (y+3), (y*3), (y-3), (y+4), (y*4), (y-4), (y+y), (y*y), (y-y), (y+x), (y*x), (y-x), (x+0), (x*0), (x-0), (x+1), (x*1), (x-1), (x+2), (x*2), (x-2), (x+3), (x*3), (x-3), (x+4), (x*4), (x-4), (x+y), (x*y), (x-y), (x+x), (x*x), (x-x)]

count: 154

level 2 :

[0, 1, 2, 3, 4, y, x, (0+0), (0*0), (0-0), (0+1), (0*1), (0-1), (0+2), (0*2), (0-2), (0+3), (0*3), (0-3), (0+4), (0*4), (0-4), (0+y), (0*y), (0-y), (0+x), (0*x), (0-x), (1+0), (1*0), (1-0), (1+1), (1*1), (1-1), (1+2), (1*2), (1-2), (1+3), (1*3), (1-3), (1+4), (1*4), (1-4), (1+y), (1*y), (1-y), (1+x), (1*x), (1-x), (2+0), (2*0), (2-0), (2+1), (2*1), (2-1), (2+2), (2*2), (2-2), (2+3), (2*3), (2-3), (2+4), (2*4), (2-4), (2+y), (2*y), (2-y), (2+x), (2*x), (2-x), (3+0), (3*0), (3-0), (3+1), (3*1), (3-1), (3+2), (3*2), (3-2), (3+3), (3*3), (3-3), (3+4), (3*4), (3-4), (3+y), (3*y), (3-y), (3+x), (3*x), (3-x), (4+0), (4*0), (4-0), (4+1), (4*1), (4-1), (4+2), (4*2), (4-2), (4+3), (4*3), (4-3), (4+4), (4*4), (4-4), (4+y), (4*y), (4-y), (4+x), (4*x), (4-x), (y+0), (y*0), (y-0), (y+1), (y*1), (y-1), (y+2), (y*2), (y-2), (y+3), (y*3), (y-3), (y+4), (y*4), (y-4), (y+y), (y*y), (y-y), (y+x), (y*x), (y-x), (x+0), (x*0), (x-0), (x+1), (x*1), (x-1), (x+2), (x*2), (x-2), (x+3), (x*3), (x-3), (x+4), (x*4), (x-4), (x+y), (x*y), (x-y), (x+x), (x*x), (x-x), (0+0), (0*0), (0-0), (0+1), (0*1), (0-1), (0+2), (0*2), (0-2), (0+3), (0*3), (0-3), (0+4), (0*4), (0-4), (0+y), (0*y), (0-y), (0+x), (0*x), (0-x), (1+0), (1*0), (1-0), (1+1), (1*1), (1-1), (1+2), (1*2), (1-2), (1+3), (1*3), (1-3), (1+4), (1*4), (1-4), (1+y), (1*y), (1-y), (1+x), (1*x), (1-x), (2+0), (2*0), (2-0), (2+1), (2*1), (2-1), (2+2), (2*2), (2-2), (2+3), (2*3), (2-3), (2+4), (2*4), (2-4), (2+y), (2*y), (2-y), (2+x), (2*x), (2-x), (3+0), (3*0), (3-0), (3+1), (3*1), (3-1), (3+2), (3*2), (3-2), (3+3), (3*3), (3-3), (3+4), (3*4), (3-4), (3+y), (3*y), (3-y), (3+x), (3*x), (3-x), (4+0), (4*0), (4-0), (4+1), (4*1), (4-1), (4+2), (4*2), (4-2), (4+3), (4*3), (4-3), (4+4), (4*4), (4-4), (4+y), (4*y), (4-y), (4+x), (4*x), (4-x), (y+0), (y*0), (y-0), (y+1), (y*1), (y-1), (y+2), (y*2), (y-2), (y+3), (y*3), (y-3), (y+4), (y*4), (y-4), (y+y), (y*y), (y-y), (y+x), (y*x), (y-x), (x+0), (x*0), (x-0), (x+1), (x*1), (x-1), (x+2), (x*2), (x-2), (x+3), (x*3), (x-3), (x+4), (x*4), (x-4), (x+y), (x*y), (x-y), (x+x), (x*x), (x-x)]

count: 71,302

Ok, no luck so far. Let's just mash these together! In every possible combination!

Hm, still no luck. Keep mashing.

Enumeration...pruned with Operational Equivalence

←Which is the fancy program synthesis way of saying "they do the same thing on the inputs we care about."

Spec:

({ "x" → 3, "y" → 7 } , 23)
({ "x" → 4, "y" → 4 } , 19)
({ "x" → 2, "y" → 12 } , 31)

Space of programs:

$expr := \mathcal{N}$
| v
| $(expr + expr)$
| $(expr - expr)$
| $(expr * expr)$

Ok, these are all just 0...which we already have. Why'd you give me these???

level 0:
[0, 1, 2, 3, 4, y, x]
count: 7

level 1 :

[0, 1, 2, 3, 4, y, x, (0+0), (0*0), (0-0), (0+1), (0*1), (0-1), (0+2), (0*2), (0-2), (0+3), (0*3), (0-3), (0+4), (0*4), (0-4), (0+y), (0*y), (0-y), (0+x), (0*x), (0-x), (1+0), (1*0), (1-0), (1+1), (1*1), (1-1), (1+2), (1*2), (1-2), (1+3), (1*3), (1-3), (1+4), (1*4), (1-4), (1+y), (1*y), (1-y), (1+x), (1*x), (1-x), (2+0), (2*0), (2-0), (2+1), (2*1), (2-1), (2+2), (2*2), (2-2), (2+3), (2*3), (2-3), (2+4), (2*4), (2-4), (2+y), (2*y), (2-y), (2+x), (2*x), (2-x), (3+0), (3*0), (3-0), (3+1), (3*1), (3-1), (3+2), (3*2), (3-2), (3+3), (3*3), (3-3), (3+4), (3*4), (3-4), (3+y), (3*y), (3-y), (3+x), (3*x), (3-x), (4+0), (4*0), (4-0), (4+1), (4*1), (4-1), (4+2), (4*2), (4-2), (4+3), (4*3), (4-3), (4+4), (4*4), (4-4), (4+y), (4*y), (4-y), (4+x), (4*x), (4-x), (y+0), (y*0), (y-0), (y+1), (y*1), (y-1), (y+2), (y*2), (y-2), (y+3), (y*3), (y-3), (y+4), (y*4), (y-4), (y+y), (y*y), (y-y), (y+x), (y*x), (y-x), (x+0), (x*0), (x-0), (x+1), (x*1), (x-1), (x+2), (x*2), (x-2), (x+3), (x*3), (x-3), (x+4), (x*4), (x-4), (x+y), (x*y), (x-y), (x+x), (x*x), (x-x)]

count: 154

And these are the same on all inputs.

And eventually we'll find some that aren't the same on *all* inputs, but are the same on {"x" → 3, "y" → 7}, {"x" → 4, "y" → 4}, and {"x" → 2, "y" → 12}

This is exactly as simple as it looks. Seriously, you can write this synthesizer in vanilla Python in one page. Let's see it!


```

1 import itertools
2 class Op:
3     ops = {"+": lambda a,b: a+b, "-": lambda a,b: a-b, "*": lambda a,b: a*b}
4     def __init__(self, a, op, b):
5         self.a = a; self.op = op; self.b = b
6     def __repr__(self):
7         return "(" + str(self.a) + self.op + str(self.b) + ")"
8     def interpret(self, argDict):
9         return Op.ops[self.op](self.a.interpret(argDict), self.b.interpret(argDict))
10 class Val:
11     def __init__(self, v):
12         self.v = v
13     def __repr__(self):
14         return str(self.v)
15     def interpret(self, argDict):
16         return self.v
17 class Var:
18     def __init__(self, n):
19         self.n = n
20     def __repr__(self):
21         return self.n
22     def interpret(self, argDict):
23         return argDict[self.n]
24
25 spec = [({"x": 3, "y": 7}, 23), # our input-output pairs
26         ({"x": 4, "y": 4}, 19),
27         ({"x": 2, "y": 12}, 31)]
28 expected_outputs = [output for inputDict, output in spec] # for convenience, the outputs we expect for our i-o pairs
29 def test_against_spec(expr):
30     outputs = [expr.interpret(inputDict) for inputDict, output in spec] # run the expression on our target inputs
31     if (outputs == expected_outputs):
32         print "found it!", expr
33         exit()
34
35 exprs = [Val(x) for x in range(5)] + [Var(x) for x in spec[0][0].keys()] # the starting set is 0 through 5 and our args
36 print "level 0:\n", exprs, "\ncount:", len(exprs)
37 for expr in exprs:
38     test_against_spec(expr) # let's just see if any of our starting exprs do the trick...
39
40 ops = Op.ops.keys() # what operators are we allowed to use?
41 level = 0
42 while(True):
43     level += 1
44     print "level", level, ":"
45     for pair in itertools.product(exprs, exprs): #let's make bigger expressions!
46         for op in ops:
47             new_expr = Op(pair[0], op, pair[1])
48             test_against_spec(new_expr)
49             exprs.append(new_expr)
50 print exprs, "\ncount:", len(exprs)

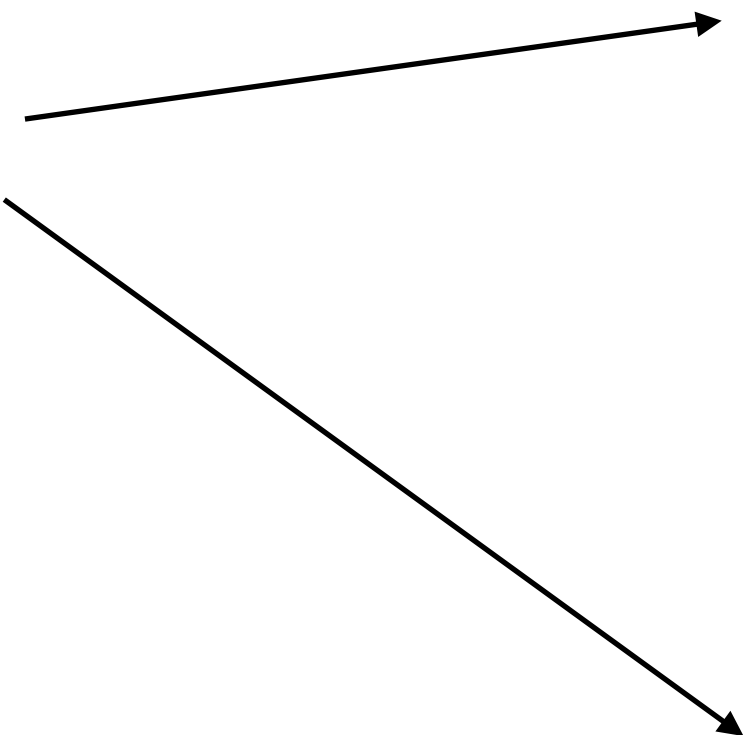
```

This one isn't pruning at all.
What do we do to prune with OE?

Just an extra 6 lines!

Pruning based on Operational
Equivalence can cut down our
search space dramatically!

And this is just at level 2!



```
[Sarahs-MBP:othermaterials schasins$ python onePageSynthesizer.py
level 0:
[0, 1, 2, 3, 4, y, x]
count: 7
level 1 :
count: 154
level 2 :
count: 71302
level 3 :
found it! (3+(2*(y+x)))
[Sarahs-MBP:othermaterials schasins$ python onePageSynthesizer0E.py
level 0:
[0, 1, 2, 3, 4, y, x]
count: 7
level 1 :
count: 63
level 2 :
count: 2051
level 3 :
found it! (3+(2*(y+x)))
```


So if you're ever watching a synthesis talk and get confused...just remember enumeration. At a sufficiently high level of abstraction, it's just going through programs until it finds one that works.

We can make enumeration smarter

- Doesn't have to be just start with the smallest program, then list all the programs in order of size until you find one that works
- We can have heuristics or language models that let us explore better/likelier programs first instead of smaller programs first
- There are other ways of pruning (other than Operational Equivalence) that let us cut out much more of the space
- We can make smart choices about what constants to include
- This was the easy-to-write version, but there are many ways to make it more effective
 - For a long time, the winner of the SyGuS competition (the primary competition for people who write synthesizers) was an enumerative solver!
 - This is a real technique!

Quick brainstorm. What would
you like to synthesize?

Synthesis is like a buffet



- This is not one technique that either applies or doesn't apply to your problem
- It's a whole family of techniques
- Tackling a new problem, you'll probably be looking through a host of existing approaches and tools...
 - If you read synth literature, you'll see very different domains formalized in very different ways. This isn't accidental!
- ...and maybe inventing your own. Custom synthesizers are still common

To think about for Thursday's reading

- The issue of ambiguous specs. As designers of usable tools, do we want to prevent ambiguous specs? If yes, how? Do we want to allow them? If yes, how does this affect our synthesizer?
- What constrains the design of a our target languages for synthesis?
- What's the tradeoff between designing for making the synthesizer's task easier vs. designing for the user of the tool?

Please install before next class

https://docs.racket-lang.org/rosette-guide/ch_getting-started.html#%28part._sec~3aget%29



ABOUT DOWNLOAD DOCS APPS COURSES PAPERS

About Rosette

Rosette is a solver-aided programming language that extends [Racket](#) with language constructs for program synthesis, verification, and more. To verify or synthesize code, Rosette compiles it to logical constraints solved with off-the-shelf [SMT](#) solvers. By combining virtualized access to solvers with Racket's metaprogramming, Rosette makes it easy to develop synthesis and verification tools for new languages. You simply

A brilliant language from
Emina Torlak