Lecture 24: Logic II

Brian Hou
August 2, 2016

Announcements

Announcements

- Project 4 is due Friday (8/5)

Announcements

- Project 4 is due Friday (8/5)
  - Finish through Part II today for 1 EC point

Announcements

- Project 4 is due Friday (8/5)
  - Finish through Part II today for 1 EC point
- Homework 9 is due Wednesday (8/3)

Announcements

- Project 4 is due Friday (8/5)
  - Finish through Part II today for 1 EC point
- Homework 9 is due Wednesday (8/3)
- Quiz 9 on Thursday (8/4) at the beginning of lecture

Announcements

- Project 4 is due Friday (8/5)
  - Finish through Part II today for 1 EC point
- Homework 9 is due Wednesday (8/3)
- Quiz 9 on Thursday (8/4) at the beginning of lecture
  - Will cover Logic

Announcements

- Project 4 is due Friday (8/5)
  - Finish through Part II today for 1 EC point
- Homework 9 is due Wednesday (8/3)
- Quiz 9 on Thursday (8/4) at the beginning of lecture
  - Will cover Logic
- Final Review on Friday (8/5) from 11–12:30pm in 2050 VLSB

Announcements

- Project 4 is due Friday (8/5)
  - Finish through Part II today for 1 EC point
- Homework 9 is due Wednesday (8/3)
- Quiz 9 on Thursday (8/4) at the beginning of lecture
  - Will cover Logic
- Final Review on Friday (8/5) from 11–12:30pm in 2050 VLSB
- Ants composition revisions due Saturday (8/6)

Announcements

- Project 4 is due Friday (8/5)
  - Finish through Part II today for 1 EC point
- Homework 9 is due Wednesday (8/3)
- Quiz 9 on Thursday (8/4) at the beginning of lecture
  - Will cover Logic
- Final Review on Friday (8/5) from 11–12:30pm in 2050 VLSB
- Ants composition revisions due Saturday (8/6)
- Scheme Recursive Art Contest is open! Submissions due 8/9

Roadmap

Introduction

Functions

Data

Mutability

Objects

Interpretation

Paradigms

Applications

Roadmap

Introduction

Functions

Data

Mutability

Objects

Interpretation

Paradigms

Applications

- This week (Paradigms), the goals are:

Roadmap

- Introduction
- Functions
- Data
- Mutability
- Objects
- Interpretation
- Paradigms
- Applications

- This week (Paradigms), the goals are:
  - To study examples of paradigms that are very different from what we have seen so far

Roadmap

- Introduction
- Functions
- Data
- Mutability
- Objects
- Interpretation
- Paradigms
- Applications

- This week (Paradigms), the goals are:
  - To study examples of paradigms that are very different from what we have seen so far
  - To expand our definition of what counts as programming

Anagram

Did you mean: *nag a ram*?

Anagrams

Anagrams

cat

Anagrams

cat at

Anagrams

at

cat at

ta

Anagrams

cat

at

cat

at

ta

Anagrams

cat

at

ta

at

cat

a**c**t

Anagrams

cat

act

at

atc

cat at

ta

Anagrams

cat

at a**c**t

at**c**

cat at

cta

ta

Anagrams

cat

at

at

cat

act

atc

ta

cta

tca

Anagrams

			cat
		at	a**c**t
			at**c**
cat	at		
			cta
		ta	t**c**a
			ta**c**

Imperative Anagrams

Imperative Anagrams

```
def anagram(s):
```

Imperative Anagrams

```
def anagram(s):
    if len(s) == 0:
```

Imperative Anagrams

```
def anagram(s):
    if len(s) == 0:
        return [[]]
```

Imperative Anagrams

```
def anagram(s):
    if len(s) == 0:
        return [[]]
    result = []
```

Imperative Anagrams

```
def anagram(s):
    if len(s) == 0:
        return [[]]
    result = []
    anagrams = anagram(s[1:])
```

Imperative Anagrams

```
def anagram(s):
    if len(s) == 0:
        return [[]]
    result = []
    anagrams = anagram(s[1:])
    for x in anagrams:
```

Imperative Anagrams

```
def anagram(s):
    if len(s) == 0:
        return [[]]
    result = []
    anagrams = anagram(s[1:])
    for x in anagrams:
        for i in range(0, len(x) + 1):
```

Imperative Anagrams

```python
def anagram(s):
    if len(s) == 0:
        return [[]]
    result = []
    anagrams = anagram(s[1:])
    for x in anagrams:
        for i in range(0, len(x) + 1):
            new_anagram = x[:i] + [s[0]] + x[i:]
```

Imperative Anagrams

```
def anagram(s):
    if len(s) == 0:
        return [[]]
    result = []
    anagrams = anagram(s[1:])
    for x in anagrams:
        for i in range(0, len(x) + 1):
            new_anagram = x[:i] + [s[0]] + x[i:]
            result.append(new_anagram)
```

Imperative Anagrams

```
def anagram(s):
    if len(s) == 0:
        return [[]]
    result = []
    anagrams = anagram(s[1:])
    for x in anagrams:
        for i in range(0, len(x) + 1):
            new_anagram = x[:i] + [s[0]] + x[i:]
            result.append(new_anagram)
    return result
```

Imperative Anagrams (demo)

```
def anagram(s):
    if len(s) == 0:
        return [[]]
    result = []
    anagrams = anagram(s[1:])
    for x in anagrams:
        for i in range(0, len(x) + 1):
            new_anagram = x[:i] + [s[0]] + x[i:]
            result.append(new_anagram)
    return result
```

Declarative Anagrams

Declarative Anagrams

```
logic> (fact (insert ?a ?r (?a . ?r)))
```

Declarative Anagrams

```
logic> (fact (insert ?a ?r (?a . ?r)))
logic> (fact (insert ?a (?b . ?r) (?b . ?s)))
```

Declarative Anagrams

```
logic> (fact (insert ?a ?r (?a . ?r)))
logic> (fact (insert ?a (?b . ?r) (?b . ?s))
             (insert ?a ?r ?s))
```

Declarative Anagrams (demo)

```
logic> (fact (insert ?a ?r (?a . ?r)))
logic> (fact (insert ?a (?b . ?r) (?b . ?s))
             (insert ?a ?r ?s))
```

Declarative Anagrams (demo)

```
logic> (fact (insert ?a ?r (?a . ?r)))
logic> (fact (insert ?a (?b . ?r) (?b . ?s))
             (insert ?a ?r ?s))

logic> (fact (anagram () ()))
```

Declarative Anagrams (demo)

```
logic> (fact (insert ?a ?r (?a . ?r)))
logic> (fact (insert ?a (?b . ?r) (?b . ?s))
             (insert ?a ?r ?s))

logic> (fact (anagram () ()))
logic> (fact (anagram (?a . ?r) ?b)
```

Declarative Anagrams (demo)

```
logic> (fact (insert ?a ?r (?a . ?r)))
logic> (fact (insert ?a (?b . ?r) (?b . ?s))
             (insert ?a ?r ?s))

logic> (fact (anagram () ()))
logic> (fact (anagram (?a . ?r) ?b)
             (insert ?a ?s ?b))
```

Declarative Anagrams (demo)

```
logic> (fact (insert ?a ?r (?a . ?r)))
logic> (fact (insert ?a (?b . ?r) (?b . ?s))
             (insert ?a ?r ?s))

logic> (fact (anagram () ()))
logic> (fact (anagram (?a . ?r) ?b)
             (insert ?a ?s ?b)
             (anagram ?r ?s))
```

Declarative Anagrams (demo)

```
logic> (fact (insert ?a ?r (?a . ?r)))
logic> (fact (insert ?a (?b . ?r) (?b . ?s))
             (insert ?a ?r ?s))

logic> (fact (anagram () ()))
logic> (fact (anagram (?a . ?r) ?b)
             (insert ?a ?s ?b)
             (anagram ?r ?s))

logic> (query (anagram ?s (s t a r)))
```

Palindromes

Palindromes

Palindromes

- A palindrome is a sequence that is the same when read backward and forward

Palindromes

- A palindrome is a sequence that is the same when read backward and forward
  - Examples: "racecar"

Palindromes

- A palindrome is a sequence that is the same when read backward and forward
  - Examples: "racecar", "senile felines"

Palindromes

- A palindrome is a sequence that is the same when read backward and forward
  - Examples: "racecar", "senile felines", "too hot to hoot"

Palindromes

- A palindrome is a sequence that is the same when read backward and forward
  - Examples: "racecar", "senile felines", "too hot to hoot"

```
logic> (fact (palindrome ?s)
```

Palindromes

- A palindrome is a sequence that is the same when read backward and forward
  - Examples: "racecar", "senile felines", "too hot to hoot"

```
logic> (fact (palindrome ?s)
             (reverse ?s ?s))
```

Palindromes

- A palindrome is a sequence that is the same when read backward and forward
  - Examples: "racecar", "senile felines", "too hot to hoot"

```
logic> (fact (palindrome ?s)
             (reverse ?s ?s))
logic> (fact (reverse () ()))
```

Palindromes

- A palindrome is a sequence that is the same when read backward and forward
  - Examples: "racecar", "senile felines", "too hot to hoot"

```
logic> (fact (palindrome ?s)
             (reverse ?s ?s))
logic> (fact (reverse () ()))
logic> (fact (reverse (?first . ?rest) ?rev)
```

Palindromes

- A palindrome is a sequence that is the same when read backward and forward
  - Examples: "racecar", "senile felines", "too hot to hoot"

```
logic> (fact (palindrome ?s)
             (reverse ?s ?s))
logic> (fact (reverse () ()))
logic> (fact (reverse (?first . ?rest) ?rev)
             (reverse ?rest ?rest-rev)
```

Palindromes

- A palindrome is a sequence that is the same when read backward and forward
  - Examples: "racecar", "senile felines", "too hot to hoot"

```
logic> (fact (palindrome ?s)
             (reverse ?s ?s))
logic> (fact (reverse () ()))
logic> (fact (reverse (?first . ?rest) ?rev)
             (reverse ?rest ?rest-rev)
             (append ?rest-rev (?first) ?rev))
```

Palindromes (demo)

- A palindrome is a sequence that is the same when read backward and forward
  - Examples: "racecar", "senile felines", "too hot to hoot"

```
logic> (fact (palindrome ?s)
             (reverse ?s ?s))
logic> (fact (reverse () ()))
logic> (fact (reverse (?first . ?rest) ?rev)
             (reverse ?rest ?rest-rev)
             (append ?rest-rev (?first) ?rev))
```

Declarative Programming

Declarative Programming

- In declarative programming, we tell the computer what a solution looks like, rather than how to get the solution

Declarative Programming

- In declarative programming, we tell the computer what a solution looks like, rather than how to get the solution
- If we describe a solution in two different ways, will the computer take the same amount of time to compute a solution?

Declarative Programming

- In declarative programming, we tell the computer what a solution looks like, rather than how to get the solution
- If we describe a solution in two different ways, will the computer take the same amount of time to compute a solution?
  - Probably not...

Reverse

Reverse

```
logic> (fact (reverse () ()))
logic> (fact (reverse (?first . ?rest) ?rev)
             (reverse ?rest ?rest-rev)
             (append ?rest-rev (?first) ?rev))
```

Reverse

```
logic> (fact (reverse () ()))
logic> (fact (reverse (?first . ?rest) ?rev)
             (reverse ?rest ?rest-rev)
             (append ?rest-rev (?first) ?rev))


logic> (fact (accrev (?first . ?rest) ?acc ?rev)
```

Reverse

```
logic> (fact (reverse () ()))
logic> (fact (reverse (?first . ?rest) ?rev)
             (reverse ?rest ?rest-rev)
             (append ?rest-rev (?first) ?rev))


logic> (fact (accrev (?first . ?rest) ?acc ?rev)
             (accrev ?rest (?first . ?acc) ?rev))
```

Reverse

```
logic> (fact (reverse () ()))
logic> (fact (reverse (?first . ?rest) ?rev)
             (reverse ?rest ?rest-rev)
             (append ?rest-rev (?first) ?rev))


logic> (fact (accrev (?first . ?rest) ?acc ?rev)
             (accrev ?rest (?first . ?acc) ?rev))
logic> (fact (accrev () ?acc ?acc))
```

Reverse

```
logic> (fact (reverse () ()))
logic> (fact (reverse (?first . ?rest) ?rev)
             (reverse ?rest ?rest-rev)
             (append ?rest-rev (?first) ?rev))


logic> (fact (accrev (?first . ?rest) ?acc ?rev)
             (accrev ?rest (?first . ?acc) ?rev))
logic> (fact (accrev () ?acc ?acc))
logic> (fact (accrev ?s ?rev)
```

Reverse

```
logic> (fact (reverse () ()))
logic> (fact (reverse (?first . ?rest) ?rev)
             (reverse ?rest ?rest-rev)
             (append ?rest-rev (?first) ?rev))


logic> (fact (accrev (?first . ?rest) ?acc ?rev)
             (accrev ?rest (?first . ?acc) ?rev))
logic> (fact (accrev () ?acc ?acc))
logic> (fact (accrev ?s ?rev)
             (accrev ?s () ?rev))
```

Reverse (demo)

```
logic> (fact (reverse () ()))
logic> (fact (reverse (?first . ?rest) ?rev)
             (reverse ?rest ?rest-rev)
             (append ?rest-rev (?first) ?rev))


logic> (fact (accrev (?first . ?rest) ?acc ?rev)
             (accrev ?rest (?first . ?acc) ?rev))
logic> (fact (accrev () ?acc ?acc))
logic> (fact (accrev ?s ?rev)
             (accrev ?s () ?rev))
```

Break!

Arithmetic

Number Representation

Number Representation

- Logic does not have numbers, but does have Scheme lists

Number Representation

- Logic does not have numbers, but does have Scheme lists
- Let's create our own number representation!

Number Representation

- Logic does not have numbers, but does have Scheme lists
- Let's create our own number representation!
    - We'll limit ourselves to non-negative integers

Number Representation

- Logic does not have numbers, but does have Scheme lists
- Let's create our own number representation!
  - We'll limit ourselves to non-negative integers
- We can represent the numbers

Number Representation

- Logic does not have numbers, but does have Scheme lists
- Let's create our own number representation!
  - We'll limit ourselves to non-negative integers
- We can represent the numbers
  - 0, 1, 2, 3, ... as

Number Representation

- Logic does not have numbers, but does have Scheme lists
- Let's create our own number representation!
  - We'll limit ourselves to non-negative integers
- We can represent the numbers
  - 0, 1, 2, 3, ... as
  - 0, (+ 1 0), (+ 1 (+ 1 0)), (+ 1 (+ 1 (+ 1 0))), ...

Number Representation

- Logic does not have numbers, but does have Scheme lists
- Let's create our own number representation!
  - We'll limit ourselves to non-negative integers
- We can represent the numbers
  - 0, 1, 2, 3, ... as
  - 0, (+ 1 0), (+ 1 (+ 1 0)), (+ 1 (+ 1 (+ 1 0))), ...
- This is still a **symbolic** representation! Logic doesn't know that these are Scheme expressions that would evaluate to that number

Addition

Addition

- Mathematical facts:

Addition

- Mathematical facts:
  - 0 + n = n

Addition

- Mathematical facts:
  - 0 + n = n

```
logic> (fact (+ 0 ?n ?n))
```

Addition

- Mathematical facts:
  - 0 + n = n
  - In order for (x + 1) + y = (z + 1) to be true, x + y = z

```
logic> (fact (+ 0 ?n ?n))
```

Addition

- Mathematical facts:
  - 0 + n = n
  - In order for (x + 1) + y = (z + 1) to be true, x + y = z

```
logic> (fact (+ 0 ?n ?n))
logic> (fact (+ (+ 1 ?x) ?y (+ 1 ?z)))
```

Addition

- Mathematical facts:
  - 0 + n = n
  - In order for (x + 1) + y = (z + 1) to be true, x + y = z

```
logic> (fact (+ 0 ?n ?n))
logic> (fact (+ (+ 1 ?x) ?y (+ 1 ?z))
             (+ ?x ?y ?z))
```

Addition (demo)

- Mathematical facts:
  - 0 + n = n
  - In order for (x + 1) + y = (z + 1) to be true, x + y = z

```
logic> (fact (+ 0 ?n ?n))
logic> (fact (+ (+ 1 ?x) ?y (+ 1 ?z))
             (+ ?x ?y ?z))
```

Addition (demo)

- Mathematical facts:
  - 0 + n = n
  - In order for (x + 1) + y = (z + 1) to be true, x + y = z

```
logic> (fact (+ 0 ?n ?n))
logic> (fact (+ (+ 1 ?x) ?y (+ 1 ?z))
             (+ ?x ?y ?z))
logic> (query (+
```

Addition (demo)

- Mathematical facts:
  - 0 + n = n
  - In order for (x + 1) + y = (z + 1) to be true, x + y = z

```
logic> (fact (+ 0 ?n ?n))
logic> (fact (+ (+ 1 ?x) ?y (+ 1 ?z))
             (+ ?x ?y ?z))
logic> (query (+
                (+ 1 (+ 1 (+ 1 0)))
```

Addition (demo)

- Mathematical facts:
  - 0 + n = n
  - In order for (x + 1) + y = (z + 1) to be true, x + y = z

```
logic> (fact (+ 0 ?n ?n))
logic> (fact (+ (+ 1 ?x) ?y (+ 1 ?z))
             (+ ?x ?y ?z))
logic> (query (+
                (+ 1 (+ 1 (+ 1 0)))
                (+ 1 (+ 1 0))
```

Addition (demo)

- Mathematical facts:
  - 0 + n = n
  - In order for (x + 1) + y = (z + 1) to be true, x + y = z

```
logic> (fact (+ 0 ?n ?n))
logic> (fact (+ (+ 1 ?x) ?y (+ 1 ?z))
             (+ ?x ?y ?z))
logic> (query (+
                (+ 1 (+ 1 (+ 1 0)))
                (+ 1 (+ 1 0))
                ?z))
```

Multiplication

Multiplication

- Mathematical facts:

Multiplication

- Mathematical facts:
  - $0 * n = 0$

Multiplication

- Mathematical facts:
  - 0 * n = 0

```
logic> (fact (* 0 ?n 0))
```

Multiplication

- Mathematical facts:
  - 0 * n = 0
  - In order for (x + 1) * y = z to be true, x * y + y = z

```
logic> (fact (* 0 ?n 0))
```

Multiplication

- Mathematical facts:
  - 0 * n = 0
  - In order for (x + 1) * y = z to be true, x * y + y = z

```
logic> (fact (* 0 ?n 0))
logic> (fact (* (+ 1 ?x) ?y ?z)
```

Multiplication

- Mathematical facts:
  - 0 * n = 0
  - In order for (x + 1) * y = z to be true, x * y + y = z

```
logic> (fact (* 0 ?n 0))
logic> (fact (* (+ 1 ?x) ?y ?z)
             (+ ?xy ?y ?z))
```

Multiplication

- Mathematical facts:
  - 0 ∗ n = 0
  - In order for (x + 1) ∗ y = z to be true, x ∗ y + y = z

```
logic> (fact (* 0 ?n 0))
logic> (fact (* (+ 1 ?x) ?y ?z)
             (+ ?xy ?y ?z)
             (* ?x ?y ?xy))
```

Multiplication (demo)

- Mathematical facts:
  - 0 * n = 0
  - In order for (x + 1) * y = z to be true, x * y + y = z

```
logic> (fact (* 0 ?n 0))
logic> (fact (* (+ 1 ?x) ?y ?z)
             (+ ?xy ?y ?z)
             (* ?x ?y ?xy))
```

Multiplication (demo)

- Mathematical facts:
  - 0 * n = 0
  - In order for (x + 1) * y = z to be true, x * y + y = z

```
logic> (fact (* 0 ?n 0))
logic> (fact (* (+ 1 ?x) ?y ?z)
             (+ ?xy ?y ?z)
             (* ?x ?y ?xy))
logic> (query (*
```

Multiplication (demo)

- Mathematical facts:
  - 0 * n = 0
  - In order for (x + 1) * y = z to be true, x * y + y = z

```
logic> (fact (* 0 ?n 0))
logic> (fact (* (+ 1 ?x) ?y ?z)
             (+ ?xy ?y ?z)
             (* ?x ?y ?xy))
logic> (query (*
                 (+ 1 (+ 1 (+ 1 0)))
```

Multiplication (demo)

- Mathematical facts:
  - 0 ∗ n = 0
  - In order for (x + 1) ∗ y = z to be true, x ∗ y + y = z

```
logic> (fact (* 0 ?n 0))
logic> (fact (* (+ 1 ?x) ?y ?z)
             (+ ?xy ?y ?z)
             (* ?x ?y ?xy))
logic> (query (*
                (+ 1 (+ 1 (+ 1 0)))
                (+ 1 (+ 1 0))
```

Multiplication (demo)

- Mathematical facts:
  - 0 * n = 0
  - In order for (x + 1) * y = z to be true, x * y + y = z

```
logic> (fact (* 0 ?n 0))
logic> (fact (* (+ 1 ?x) ?y ?z)
             (+ ?xy ?y ?z)
             (* ?x ?y ?xy))
logic> (query (*
                (+ 1 (+ 1 (+ 1 0)))
                (+ 1 (+ 1 0))
                ?z))
```

Subtraction and Division

Subtraction and Division

- Mathematical facts:

Subtraction and Division

- Mathematical facts:
  - Subtraction is the inverse of addition

Subtraction and Division

- Mathematical facts:
  - Subtraction is the inverse of addition
    - In order for $x - y = z$, $y + z = x$

Subtraction and Division

- Mathematical facts:
  - Subtraction is the inverse of addition
    - In order for x − y = z, y + z = x

```
logic> (fact (- ?x ?y ?z)
```

Subtraction and Division

- Mathematical facts:
  - Subtraction is the inverse of addition
    - In order for $x - y = z$, $y + z = x$

```
logic> (fact (- ?x ?y ?z)
             (+ ?y ?z ?x))
```

Subtraction and Division

- Mathematical facts:
  - Subtraction is the inverse of addition
    - In order for $x - y = z$, $y + z = x$
  - Division is the inverse of multiplication

```
logic> (fact (- ?x ?y ?z)
             (+ ?y ?z ?x))
```

Subtraction and Division

- Mathematical facts:
  - Subtraction is the inverse of addition
    - In order for x − y = z, y + z = x
  - Division is the inverse of multiplication
    - In order for x / y = z, y * z = x (assuming x is divisible by y)

```
logic> (fact (- ?x ?y ?z)
             (+ ?y ?z ?x))
```

Subtraction and Division

- Mathematical facts:
  - Subtraction is the inverse of addition
    - In order for x − y = z, y + z = x
  - Division is the inverse of multiplication
    - In order for x / y = z, y * z = x (assuming x is divisible by y)

```
logic> (fact (- ?x ?y ?z)
             (+ ?y ?z ?x))
logic> (fact (/ ?x ?y ?z)
```

Subtraction and Division

- Mathematical facts:
  - Subtraction is the inverse of addition
    - In order for x − y = z, y + z = x
  - Division is the inverse of multiplication
    - In order for x / y = z, y * z = x (assuming x is divisible by y)

```
logic> (fact (- ?x ?y ?z)
             (+ ?y ?z ?x))
logic> (fact (/ ?x ?y ?z)
             (* ?y ?z ?x))
```

Subtraction and Division (demo)

- Mathematical facts:
  - Subtraction is the inverse of addition
    - In order for x - y = z, y + z = x
  - Division is the inverse of multiplication
    - In order for x / y = z, y * z = x (assuming x is divisible by y)

```
logic> (fact (- ?x ?y ?z)
             (+ ?y ?z ?x))
logic> (fact (/ ?x ?y ?z)
             (* ?y ?z ?x))
```

Arithmetic

Arithmetic

- We've implemented the four basic arithmetic operations!

Arithmetic (demo)

- We've implemented the four basic arithmetic operations!

Arithmetic (demo)

- We've implemented the four basic arithmetic operations!
- We can now ask Logic about all the different ways to compute the number 6

Arithmetic (demo)

- We've implemented the four basic arithmetic operations!
- We can now ask Logic about all the different ways to compute the number 6

```
logic> (query (?op ?arg1 ?arg2
              (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 0))))))))
```

Summary

Summary

- Some problems can be solved more easily or concisely with declarative programming than imperative programming

Summary

- Some problems can be solved more easily or concisely with declarative programming than imperative programming
- However, just because the computer is the one solving the problem doesn't mean that we can write any declarative program and it will "just work"

Summary

- Some problems can be solved more easily or concisely with declarative programming than imperative programming
- However, just because the computer is the one solving the problem doesn't mean that we can write any declarative program and it will "just work"
- As declarative programmers, we (eventually) should understand how the underlying problem solver works

Summary

- Some problems can be solved more easily or concisely with declarative programming than imperative programming
- However, just because the computer is the one solving the problem doesn't mean that we can write any declarative program and it will "just work"
- As declarative programmers, we (eventually) should understand how the underlying problem solver works
- This semester, just focus on writing declarative programs; no need to worry about the underlying solver yet!