Lecture 23: Logic I

Marvin Zhang
08/01/2016
Announcements
Roadmap

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

- This week (Paradigms), the goals are:
This week (Paradigms), the goals are:

- To study examples of paradigms that are very different from what we have seen so far.
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
Today’s Example: Map Coloring
Today’s Example: Map Coloring

• Problem: Given a map divided into regions, is there a way to color each region red, blue, or green without using the same color for any neighboring regions?
Today’s Example: Map Coloring

• Problem: Given a map divided into regions, is there a way to color each region red, blue, or green without using the same color for any neighboring regions?
Today’s Example: Map Coloring

• Problem: Given a map divided into regions, is there a way to color each region red, blue, or green without using the same color for any neighboring regions?
Imperative Programming
Imperative Programming

• All of the programs we have seen so far are examples of imperative programming, i.e., they specify detailed instructions that the computer carries out.
Imperative Programming

• All of the programs we have seen so far are examples of *imperative programming*, i.e., they specify detailed instructions that the computer carries out

• In imperative programming, the programmer must first solve the problem, and then code that solution
Imperative Programming

• All of the programs we have seen so far are examples of imperative programming, i.e., they specify detailed instructions that the computer carries out
  • In imperative programming, the programmer must first solve the problem, and then code that solution
  • But what if we can’t solve the problem? Or what if we can’t code the solution?
Imperative Programming

• All of the programs we have seen so far are examples of imperative programming, i.e., they specify detailed instructions that the computer carries out
  • In imperative programming, the programmer must first solve the problem, and then code that solution
  • But what if we can’t solve the problem? Or what if we can’t code the solution?

  # Imperative map coloring
All of the programs we have seen so far are examples of imperative programming, i.e., they specify detailed instructions that the computer carries out.

In imperative programming, the programmer must first solve the problem, and then code that solution.

But what if we can’t solve the problem? Or what if we can’t code the solution?

```python
# Imperative map coloring
colors = ['red', 'blue', 'green']
```
Imperative Programming

• All of the programs we have seen so far are examples of \textit{imperative programming}, i.e., they specify detailed instructions that the computer carries out

• In imperative programming, the programmer must first solve the problem, and then code that solution

• But what if we can’t solve the problem? Or what if we can’t code the solution?

    \begin{verbatim}
    # Imperative map coloring
    colors = ['red', 'blue', 'green']
    for region in map:
    \end{verbatim}
Imperative Programming

- All of the programs we have seen so far are examples of *imperative programming*, i.e., they specify detailed instructions that the computer carries out.

- In imperative programming, the programmer must first solve the problem, and then code that solution.

- But what if we can’t solve the problem? Or what if we can’t code the solution?

```python
# Imperative map coloring
colors = ['red', 'blue', 'green']
for region in map:
    i = 0
```
Imperative Programming

• All of the programs we have seen so far are examples of *imperative programming*, i.e., they specify detailed instructions that the computer carries out

• In imperative programming, the programmer must first solve the problem, and then code that solution

• But what if we can’t solve the problem? Or what if we can’t code the solution?

```python
# Imperative map coloring
colors = ['red', 'blue', 'green']
for region in map:
    i = 0
    while not region.valid:
```
Imperative Programming

• All of the programs we have seen so far are examples of imperative programming, i.e., they specify detailed instructions that the computer carries out.

• In imperative programming, the programmer must first solve the problem, and then code that solution.

• But what if we can’t solve the problem? Or what if we can’t code the solution?

```
# Imperative map coloring
colors = ['red', 'blue', 'green']
for region in map:
    i = 0
    while not region.valid:
        region.color = colors[i]
```
Imperative Programming

• All of the programs we have seen so far are examples of imperative programming, i.e., they specify detailed instructions that the computer carries out.

• In imperative programming, the programmer must first solve the problem, and then code that solution.

• But what if we can’t solve the problem? Or what if we can’t code the solution?

```python
# Imperative map coloring
colors = ['red', 'blue', 'green']
for region in map:
    i = 0
    while not region.valid:
        region.color = colors[i]
        i += 1
```
Imperative Programming

• All of the programs we have seen so far are examples of imperative programming, i.e., they specify detailed instructions that the computer carries out

• In imperative programming, the programmer must first solve the problem, and then code that solution

• But what if we can’t solve the problem? Or what if we can’t code the solution?

```python
# Imperative map coloring
colors = ['red', 'blue', 'green']
for region in map:
    i = 0
    while not region.valid:
        region.color = colors[i]
        i += 1
    if i >= len(colors):
```
Imperative Programming

- All of the programs we have seen so far are examples of imperative programming, i.e., they specify detailed instructions that the computer carries out.
- In imperative programming, the programmer must first solve the problem, and then code that solution.
- But what if we can’t solve the problem? Or what if we can’t code the solution?

```python
# Imperative map coloring
colors = ['red', 'blue', 'green']
for region in map:
    i = 0
    while not region.valid:
        region.color = colors[i]
        i += 1
    if i >= len(colors):
        # ???
```
Declarative Programming
Declarative Programming

- In *declarative programming*, we specify the properties that a solution satisfies, instead of specifying the instructions to compute the solution.
Declarative Programming

• In *declarative programming*, we specify the properties that a solution satisfies, instead of specifying the instructions to compute the solution
  • We tell the computer *what the solution looks like*, instead of *how to get the solution*
Declarative Programming

- In *declarative programming*, we specify the properties that a solution satisfies, instead of specifying the instructions to compute the solution
  - We tell the computer *what the solution looks like*, instead of *how to get the solution*
  - This is simpler, more natural, and more intuitive for certain problems and domains
Declarative Programming

• In declarative programming, we specify the properties that a solution satisfies, instead of specifying the instructions to compute the solution
  • We tell the computer what the solution looks like, instead of how to get the solution
  • This is simpler, more natural, and more intuitive for certain problems and domains
  • We will write code that looks like this:
Declarative Programming

• In *declarative programming*, we specify the properties that a solution satisfies, instead of specifying the instructions to compute the solution
  
  • We tell the computer *what the solution looks like*, instead of *how to get the solution*
  
  • This is simpler, more natural, and more intuitive for certain problems and domains
  
  • We will write code that looks like this:

    ```
    # Declarative map coloring idea:
    Find a solution where:
    - All regions of the map are colored
    - No neighboring regions have the same color
    ```
Disclaimer
Disclaimer

- Declarative languages move the job of solving the problem over from the programmer to the interpreter
Disclaimer

- Declarative languages move the job of solving the problem over from the programmer to the interpreter.
- However, building a problem solver is hard! We don’t know how to build a universal problem solver.
Disclaimer

• Declarative languages move the job of solving the problem over from the programmer to the interpreter

• However, building a problem solver is hard! We don’t know how to build a universal problem solver

• As a result, declarative languages usually only handle some subset of problems
Disclaimer

- Declarative languages move the job of solving the problem over from the programmer to the interpreter.

- However, building a problem solver is hard! We don’t know how to build a universal problem solver.

- As a result, declarative languages usually only handle some subset of problems.

- Many problems will still require careful thought and a clever approach from the programmer.
Disclaimer

- Declarative languages move the job of solving the problem over from the programmer to the interpreter.
- However, building a problem solver is hard! We don’t know how to build a universal problem solver.
- As a result, declarative languages usually only handle some subset of problems.
- Many problems will still require careful thought and a clever approach from the programmer.
- Think declaratively, not imperatively.
Disclaimer

- Declarative languages move the job of solving the problem over from the programmer to the interpreter.
- However, building a problem solver is hard! We don’t know how to build a universal problem solver.
- As a result, declarative languages usually only handle some subset of problems.
- Many problems will still require careful thought and a clever approach from the programmer.
- Think declaratively, not imperatively.

Today’s Lecture
Disclaimer

- Declarative languages move the job of solving the problem over from the programmer to the interpreter
- However, building a problem solver is hard! We don’t know how to build a universal problem solver
- As a result, declarative languages usually only handle some subset of problems
- Many problems will still require careful thought and a clever approach from the programmer
- Think declaratively, not imperatively

Today’s Lecture

- Solve some cool problems
Disclaimer

- Declarative languages move the job of solving the problem over from the programmer to the interpreter.
- However, building a problem solver is hard! We don’t know how to build a universal problem solver.
- As a result, declarative languages usually only handle some subset of problems.
- Many problems will still require careful thought and a clever approach from the programmer.
- Think declaratively, not imperatively.

Today’s Lecture

- Solve some cool problems.
- As long as the problem is not too big.
Disclaimer

• Declarative languages move the job of solving the problem over from the programmer to the interpreter

• However, building a problem solver is hard! We don’t know how to build a universal problem solver

• As a result, declarative languages usually only handle some subset of problems

• Many problems will still require careful thought and a clever approach from the programmer

• Think declaratively, not imperatively

Today’s Lecture

• Solve some cool problems
• As long as the problem is not too big
• Requires cleverness from the programmer
Disclaimer

- Declarative languages move the job of solving the problem over from the programmer to the interpreter.

- However, building a problem solver is hard! We don’t know how to build a universal problem solver.

- As a result, declarative languages usually only handle some subset of problems.

- Many problems will still require careful thought and a clever approach from the programmer.

- Think declaratively, not imperatively.

Today’s Lecture

- Solve some cool problems
- As long as the problem is not too big
- Requires cleverness from the programmer

Most Declarative Programming
Disclaimer

• Declarative languages move the job of solving the problem over from the programmer to the interpreter.

• However, building a problem solver is hard! We don’t know how to build a universal problem solver.

• As a result, declarative languages usually only handle some subset of problems.

• Many problems will still require careful thought and a clever approach from the programmer.

• Think declaratively, not imperatively.

Today's Lecture

• Solve some cool problems
• As long as the problem is not too big
• Requires cleverness from the programmer

Most Declarative Programming

• Solve less cool problems
Disclaimer

• Declarative languages move the job of solving the problem over from the programmer to the interpreter

• However, building a problem solver is hard! We don’t know how to build a universal problem solver

• As a result, declarative languages usually only handle some subset of problems

• Many problems will still require careful thought and a clever approach from the programmer

• Think declaratively, not imperatively

Today’s Lecture

• Solve some cool problems
• As long as the problem is not too big
• Requires cleverness from the programmer

Most Declarative Programming

• Solve less cool problems
• But the problems can be much bigger
Disclaimer

- Declarative languages move the job of solving the problem over from the programmer to the interpreter.
- However, building a problem solver is hard! We don’t know how to build a universal problem solver.
- As a result, declarative languages usually only handle some subset of problems.
- Many problems will still require careful thought and a clever approach from the programmer.
- Think declaratively, not imperatively.

Today’s Lecture

- Solve some cool problems
- As long as the problem is not too big
- Requires cleverness from the programmer

Most Declarative Programming

- Solve less cool problems
- But the problems can be much bigger
- More standard approach for programmers
Logic

The programming language
Logic
Logic

• The Logic language was built for this course
Logic

• The Logic language was built for this course
• Borrows syntax from Scheme and semantics from Prolog (1972)
Logic

- The Logic language was built for this course
- Borrows syntax from Scheme and semantics from Prolog (1972)
- Programs consist of relations, which are lists of symbols
Logic

- The Logic language was built for this course
- Borrows syntax from Scheme and semantics from Prolog (1972)
- Programs consist of relations, which are lists of symbols
  - Logic is pure symbolic programming, no concept of numbers or arithmetic of any kind
Logic

• The Logic language was built for this course
• Borrows syntax from Scheme and semantics from Prolog (1972)
• Programs consist of relations, which are lists of symbols
  • Logic is pure symbolic programming, no concept of numbers or arithmetic of any kind
• There are two types of expressions:
Logic

- The Logic language was built for this course
- Borrows syntax from Scheme and semantics from Prolog (1972)
- Programs consist of relations, which are lists of symbols
  - Logic is pure symbolic programming, no concept of numbers or arithmetic of any kind
- There are two types of expressions:
  - Facts declare relations to be true
Logic

• The Logic language was built for this course
• Borrows syntax from Scheme and semantics from Prolog (1972)
• Programs consist of *relations*, which are lists of symbols
  • Logic is pure symbolic programming, no concept of numbers or arithmetic of any kind
• There are two types of expressions:
  • *Facts* declare relations to be true
    • All relations are false until declared true by a fact
Logic

• The Logic language was built for this course
• Borrows syntax from Scheme and semantics from Prolog (1972)
• Programs consist of relations, which are lists of symbols
  • Logic is pure symbolic programming, no concept of numbers or arithmetic of any kind
• There are two types of expressions:
  • Facts declare relations to be true
    • All relations are false until declared true by a fact
  • Queries ask whether relations are true, based on the facts that have been declared
Logic

- The Logic language was built for this course
- Borrows syntax from Scheme and semantics from Prolog (1972)
- Programs consist of relations, which are lists of symbols
  - Logic is pure symbolic programming, no concept of numbers or arithmetic of any kind
- There are two types of expressions:
  - Facts declare relations to be true
    - All relations are false until declared true by a fact
  - Queries ask whether relations are true, based on the facts that have been declared
    - It is the job of the interpreter to figure out if a query is true or false
The Logic language was built for this course. Borrows syntax from Scheme and semantics from Prolog (1972). Programs consist of relations, which are lists of symbols. Logic is pure symbolic programming, no concept of numbers or arithmetic of any kind.

There are two types of expressions:

- **Facts** declare relations to be true
  - All relations are false until declared true by a fact
- **Queries** ask whether relations are true, based on the facts that have been declared
  - It is the job of the interpreter to figure out if a query is true or false
Variables
Variables

• Relations can contain variables, which start with ?
Variables

• Relations can contain variables, which start with ?
• A variable can take on the value of a symbol
Variables

- Relations can contain variables, which start with ?
- A variable can take on the value of a symbol

```
logic> (fact (border NSW Q))
```
Variables

• Relations can contain variables, which start with ?
• A variable can take on the value of a symbol

logic> (fact (border NSW Q))
logic> (query (border NSW Q))
Variables

- Relations can contain variables, which start with ?
- A variable can take on the value of a symbol

```logic
logic> (fact (border NSW Q))
logic> (query (border NSW Q))
Success!
```
Variables

• Relations can contain variables, which start with `?`
• A variable can take on the value of a symbol

```
logic> (fact (border NSW Q))
logic> (query (border NSW Q))
Success!
logic> (query (border NSW NT))
```
Variables

- Relations can contain variables, which start with ?
- A variable can take on the value of a symbol

```
logic> (fact (border NSW Q))
logic> (query (border NSW Q))
Success!
logic> (query (border NSW NT))
Failed.
```
Variables

- Relations can contain variables, which start with `?`
- A variable can take on the value of a symbol

```logic
logic> (fact (border NSW Q))
logic> (query (border NSW Q))
Success!
logic> (query (border NSW NT))
Failed.
logic> (query (border NSW ?region))
```
Variables

- Relations can contain variables, which start with ?
- A variable can take on the value of a symbol

```
logic> (fact (border NSW Q))
logic> (query (border NSW Q))
Success!
logic> (query (border NSW NT))
Failed.
logic> (query (border NSW ?region))
```
Variables

- Relations can contain variables, which start with ?
- A variable can take on the value of a symbol

```logic
logic> (fact (border NSW Q))
Success!
logic> (query (border NSW Q))
Failed.
logic> (query (border NSW NT))
Success!
```

variable
Variables

- Relations can contain variables, which start with ?
- A variable can take on the value of a symbol

```
logic> (fact (border NSW Q))
logic> (query (border NSW Q))
Success!
logic> (query (border NSW NT))
Failed.
logic> (query (border NSW ?region))
Success!
region: q
```

variable
Variables

- Relations can contain variables, which start with ?
- A variable can take on the value of a symbol
  
  ```logic
  logic> (fact (border NSW Q))
  logic> (query (border NSW Q))
  Success!
  logic> (query (border NSW NT))
  Failed.
  logic> (query (border NSW ?region))
  Success!
  region: q
  ```

- Relations in facts can also contain variables
Variables

- Relations can contain variables, which start with `?`
- A variable can take on the value of a symbol
  ```logic
  logic> (fact (border NSW Q))
  logic> (query (border NSW Q))
  Success!
  logic> (query (border NSW NT))
  Failed.
  logic> (query (border NSW ?region))
  Success!
  variable
  region: q
  ```
- Relations in facts can also contain variables
  ```logic
  logic> (fact (equal ?x ?x))
  ```
Variables

- Relations can contain variables, which start with ?
- A variable can take on the value of a symbol
  
  ```
  logic> (fact (border NSW Q))
  logic> (query (border NSW Q))
  Success!
  logic> (query (border NSW NT))
  Failed.
  logic> (query (border NSW ?region))
  Success!
  region: q
  ```

- Relations in facts can also contain variables
  
  ```
  logic> (fact (equal ?x ?x))
  logic> (query (equal brian brian))
  ```
Variables

• Relations can contain variables, which start with \(?\).
• A variable can take on the value of a symbol

\[
\text{logic}\rangle (\text{fact } (\text{border NSW } Q))
\]
\[
\text{logic}\rangle (\text{query } (\text{border NSW } Q))
\]
Success!
\[
\text{logic}\rangle (\text{query } (\text{border NSW NT}))
\]
Failed.
\[
\text{logic}\rangle (\text{query } (\text{border NSW } ?\text{region}))
\]
Success!
region: q

• Relations in facts can also contain variables

\[
\text{logic}\rangle (\text{fact } (\text{equal } ?x ?x))
\]
\[
\text{logic}\rangle (\text{query } (\text{equal brian brian}))
\]
Success!
Variables

- Relations can contain variables, which start with ?
- A variable can take on the value of a symbol

```
logic> (fact (border NSW Q))
logic> (query (border NSW Q))
Success!
logic> (query (border NSW NT))
Failed.
logic> (query (border NSW ?region))
Success!
region: q
```

- Relations in facts can also contain variables

```
logic> (fact (equal ?x ?x))
logic> (query (equal brian brian))
Success!
logic> (query (equal brian marvin))
```
Variables

• Relations can contain variables, which start with ?

• A variable can take on the value of a symbol

  logic> (fact (border NSW Q))
  logic> (query (border NSW Q))
  Success!
  logic> (query (border NSW NT))
  Failed.
  logic> (query (border NSW ?region))
  Success!
  region: q

• Relations in facts can also contain variables

  logic> (fact (equal ?x ?x))
  logic> (query (equal brian brian))
  Success!
  logic> (query (equal brian marvin))
  Failed.
Negation
Negation

- What if we want to check if a relation is false, rather than if it is true?
Negation

• What if we want to check if a relation is false, rather than if it is true?
  • (not <relation>) is true if <relation> is false, and false if <relation> is true
Negation

- What if we want to check if a relation is false, rather than if it is true?
  - \((\text{not } \langle \text{relation} \rangle)\) is true if \langle \text{relation} \rangle\ is false, and false if \langle \text{relation} \rangle\ is true
- This is an idea known as negation as failure
Negation

• What if we want to check if a relation is false, rather than if it is true?
  • \((\text{not } \text{<relation>})\) is true if \(<\text{relation}>\) is false, and false if \(<\text{relation}>\) is true
  • This is an idea known as \textit{negation as failure}

\[
\text{logic> (query (not (border NSW NT)))}
\]
Negation

What if we want to check if a relation is false, rather than if it is true?

- (not <relation>) is true if <relation> is false, and false if <relation> is true

- This is an idea known as negation as failure

  logic> (query (not (border NSW NT)))
  Success!
Negation

• What if we want to check if a relation is false, rather than if it is true?
  • (not <relation>) is true if <relation> is false, and false if <relation> is true
  • This is an idea known as *negation as failure*

```lisp
logic> (query (not (border NSW NT)))
Success!
logic> (query (not (equal brian marvin)))
```
Negation

• What if we want to check if a relation is false, rather than if it is true?
  
  • \((\text{not } \text{relation})\) is true if \(\text{relation}\) is false, and false if \(\text{relation}\) is true

• This is an idea known as negation as failure

  \text{logic} > \text{(query} (\text{not } \text{(border NSW NT)}))
  \text{Success!}

  \text{logic} > \text{(query} (\text{not } \text{(equal brian marvin)}))
  \text{Success!}
Negation

• What if we want to check if a relation is false, rather than if it is true?

• (not <relation>) is true if <relation> is false, and false if <relation> is true

• This is an idea known as negation as failure

```
logic> (query (not (border NSW NT)))
Success!
logic> (query (not (equal brian marvin)))
Success!
logic> (query (not (equal brian brian)))
```
Negation

• What if we want to check if a relation is false, rather than if it is true?
  
  • \((\text{not } \text{<relation>})\) is true if \(<\text{relation}>\) is false, and false if \(<\text{relation}>\) is true

  • This is an idea known as *negation as failure*

    logic> (query (not (border NSW NT)))
    Success!
    logic> (query (not (equal brian marvin)))
    Success!
    logic> (query (not (equal brian brian)))
    Failed.
Negation

• What if we want to check if a relation is false, rather than if it is true?
  
  • (not <relation>) is true if <relation> is false, and false if <relation> is true

• This is an idea known as negation as failure

  logic> (query (not (border NSW NT)))
  Success!

  logic> (query (not (equal brian marvin)))
  Success!

  logic> (query (not (equal brian brian)))
  Failed.

• Sometimes, negation as failure does not work the same as logical negation
Negation

• What if we want to check if a relation is false, rather than if it is true?
  
  • \((\text{not } \text{<relation>})\) is true if \(<\text{relation}>\) is false, and false if \(<\text{relation}>\) is true

• This is an idea known as **negation as failure**

  ```
  logic> (query (not (border NSW NT)))
  Success!
  logic> (query (not (equal brian marvin)))
  Success!
  logic> (query (not (equal brian brian)))
  Failed.
  ```

• Sometimes, negation as failure does not work the same as logical negation

• It is useful to be able to understand the differences
Negation

• What if we want to check if a relation is false, rather than if it is true?
  
  • \((\text{not } \text{relation})\) is true if \(\text{relation}\) is false, and false if \(\text{relation}\) is true
  
  • This is an idea known as \textit{negation as failure}

```logic>
(query (not (border NSW NT)))
Success!
```

```logic>
(query (not (equal brian marvin)))
Success!
```

```logic>
(query (not (equal brian brian)))
Failed.
```

• Sometimes, negation as failure does not work the same as logical negation

• It is useful to be able to understand the differences
Negation

• What if we want to check if a relation is false, rather than if it is true?
  • (not <relation>) is true if <relation> is false, and false if <relation> is true
  • This is an idea known as negation as failure

logic> (query (not (border NSW NT)))
Success!
logic> (query (not (equal brian marvin)))
Success!
logic> (query (not (equal brian brian)))
Failed.

• Sometimes, negation as failure does not work the same as logical negation
  • It is useful to be able to understand the differences

logic> (query (not (equal brian ?who)))
Failed.
Compound Facts
Compound Facts

- Compound facts contain more than one relation
Compound Facts

- Compound facts contain more than one relation
- The first relation is the *conclusion* and the subsequent relations are *hypotheses*
Compound Facts

- Compound facts contain more than one relation
- The first relation is the **conclusion** and the subsequent relations are **hypotheses**
  
  (fact <conclusion> <hypothesis-1> ... <hypothesis-n>)
Compound Facts

- Compound facts contain more than one relation
- The first relation is the conclusion and the subsequent relations are hypotheses
  (fact <conclusion> <hypothesis-1> ... <hypothesis-n>)
- The conclusion is true if, and only if, all of the hypotheses are true
Compound Facts

- Compound facts contain more than one relation
- The first relation is the *conclusion* and the subsequent relations are *hypotheses*
  
  (fact <conclusion> <hypothesis-1> ... <hypothesis-n>)

- The conclusion is true if, and only if, all of the hypotheses are true
Compound Facts

• Compound facts contain more than one relation

• The first relation is the conclusion and the subsequent relations are hypotheses
  
  (fact <conclusion> <hypothesis-1> ... <hypothesis-n>)

• The conclusion is true if, and only if, all of the hypotheses are true

; declare all border relations first
logic> (fact (two-away ?r1 ?r2)  
  (border ?r1 ?mid)  
  (border ?mid ?r2)  
  (not (border ?r1 ?r2)))

logic> (query (two-away ?r1 ?r2))
Success!

r1: nsw  r2: wa
r1: nt  r2: v
r1: q  r2: wa
r1: q  r2: v
An Aside
An Aside

\texttt{logic> (query (border NSW Q))}
An Aside

logic> (query (border NSW Q))
Success!
An Aside

```
logic> (query (border NSW Q))
Success!
logic> (query (border Q NSW))
```
An Aside

```
logic> (query (border NSW Q))
Success!
logic> (query (border Q NSW))
Failed.
```
An Aside

• Relations are not symmetric, which is weird for borders

```logic>
(query (border NSW Q))
Success!
```
An Aside

• Relations are not symmetric, which is weird for borders
• We can fix this by declaring more facts for borders, but we won’t do that yet because doing so introduces cycles

```
logic> (query (border NSW Q))
Success!
logic> (query (border Q NSW))
Failed.
```
An Aside

• Relations are not symmetric, which is weird for borders
• We can fix this by declaring more facts for borders, but we won’t do that yet because doing so introduces cycles
• Handling cycles is hard (remember cyclic linked lists?), and makes the whole example a bit too complicated

```
logic> (query (border NSW Q))
Success!
logic> (query (border Q NSW))
Failed.
```
An Aside

- Relations are not symmetric, which is weird for borders
- We can fix this by declaring more facts for borders, but we won’t do that yet because doing so introduces cycles
- Handling cycles is hard (remember cyclic linked lists?), and makes the whole example a bit too complicated
- So we will leave it out for now

```
logic> (query (border NSW Q))
Success!
logic> (query (border Q NSW))
Failed.
```
An Aside

- Relations are not symmetric, which is weird for borders
- We can fix this by declaring more facts for borders, but we won’t do that yet because doing so introduces cycles
- Handling cycles is hard (remember cyclic linked lists?), and makes the whole example a bit too complicated
  - So we will leave it out for now
- But the basic idea is that, if we have cycles, we have to keep track of what regions we have already seen, to make sure we don’t look through the same regions forever

logic> (query (border NSW Q))
Success!
logic> (query (border Q NSW))
Failed.
Compound Queries
Compound Queries

• Compound queries contain more than one relation
Compound Queries

- Compound queries contain more than one relation
  
  (query <relation-1> ... <relation-n>)
Compound Queries

- Compound queries contain more than one relation
  
  \[(\text{query } <\text{relation-1}> \ldots <\text{relation-n}>)\]

- The query succeeds if, and only if, all of the relations are true
Compound Queries

- Compound queries contain more than one relation
  (query <relation-1> ... <relation-n>)
- The query succeeds if, and only if, all of the relations are true
Compound Queries

- Compound queries contain more than one relation
  
  \[(\text{query } <\text{relation-1}> \ldots <\text{relation-n}>)\]

- The query succeeds if, and only if, all of the relations are true

\text{logic} \ (query \ (\text{two-away NSW } ?\text{region}) \ (\text{two-away Q } ?\text{region}))
Compound Queries

- Compound queries contain more than one relation
  \[(\text{query } \langle \text{relation-1} \rangle \ldots \langle \text{relation-n} \rangle)\]
- The query succeeds if, and only if, all of the relations are true

\text{logic}\ (\text{query}\ \langle\text{two-away NSW ?region}\rangle\n
\langle\text{two-away Q ?region}\rangle)\n
\text{Success!}
Compound Queries

• Compound queries contain more than one relation
  \[(query \ <relation-1> \ ... \ <relation-n>)\]
• The query succeeds if, and only if, all of the relations are true

\[
\text{logic} \geq (query \ (\text{two-away NSW } ?\text{region}) \ \ (\text{two-away Q } ?\text{region}))
\]

Success!
region: wa
Compound Queries

- Compound queries contain more than one relation
  
  \[(\text{query}\ <\text{relation-1}>\ ...\ <\text{relation-n}>)\]

- The query succeeds if, and only if, all of the relations are true

\begin{verbatim}
logic> (query (two-away NSW ?region)
              (two-away Q ?region))
Success!
region: wa
\end{verbatim}

\begin{verbatim}
logic> (query (two-away ?r1 ?r2)
              (border NT ?r2))
\end{verbatim}
Compound Queries

- Compound queries contain more than one relation
  
  (query <relation-1> ... <relation-n>)

- The query succeeds if, and only if, all of the relations are true

```plaintext
logic> (query (two-away NSW ?region) (two-away Q ?region))
Success!
region: wa
logic> (query (two-away ?r1 ?r2) (border NT ?r2))
Success!
```
Compound Queries

- Compound queries contain more than one relation
  
  \[(\text{query } <\text{relation-1}> \ldots <\text{relation-n}>)\]

- The query succeeds if, and only if, all of the relations are true

```
logic> (query (two-away NSW ?region) 
             (two-away Q ?region))
Success!
region: wa
logic> (query (two-away ?r1 ?r2) 
             (border NT ?r2))
Success!
r1: nsw   r2: wa
```
Compound Queries

- Compound queries contain more than one relation
  
  \[(\text{query} \ <\text{relation-1}> \ \ldots \ <\text{relation-n}>)\]

- The query succeeds if, and only if, all of the relations are true

```
logic> (query (two-away NSW ?region) (two-away Q ?region))
Success!
region: wa
logic> (query (two-away ?r1 ?r2) (border NT ?r2))
Success!
r1: nsw  r2: wa
r1: q    r2: wa
```
Recursive facts

Also, hierarchical facts
Recursive Facts
Recursive Facts

- A recursive fact uses the same relation in the conclusion and one or more hypotheses
Recursive Facts

• A recursive fact uses the same relation in the conclusion and one or more hypotheses

• Just like in imperative programming, we need a base fact that stops the recursion
Recursive Facts

• A recursive fact uses the same relation in the conclusion and one or more hypotheses.

• Just like in imperative programming, we need a base fact that stops the recursion.
**Recursive Facts**

- A *recursive fact* uses the same relation in the conclusion and one or more hypotheses.
- Just like in imperative programming, we need a *base fact* that stops the recursion.
Recursive Facts

- A recursive fact uses the same relation in the conclusion and one or more hypotheses.
- Just like in imperative programming, we need a base fact that stops the recursion.

```
logic> (fact (connected ?r1 ?r2) (border ?r1 ?r2))
logic> (fact (connected ?r1 ?r2) (border ?r1 ?next) (connected ?next ?r2))
```
Recursive Facts

- A recursive fact uses the same relation in the conclusion and one or more hypotheses.
- Just like in imperative programming, we need a base fact that stops the recursion.

\[
\begin{align*}
\text{logic}> & \quad \text{(fact (connected \ ?r1 \ ?r2))} \\
\text{logic}> & \quad \text{(border \ ?r1 \ ?r2))}
\end{align*}
\]

\[
\begin{align*}
\text{logic}> & \quad \text{(fact (connected \ ?r1 \ ?r2))} \\
& \quad \text{(border \ ?r1 \ ?next)} \\
& \quad \text{(connected \ ?next \ ?r2))}
\end{align*}
\]
Recursive Facts

- A **recursive fact** uses the same relation in the conclusion and one or more hypotheses.
- Just like in imperative programming, we need a **base fact** that stops the recursion.

```
logic> (fact (connected ?r1 ?r2)
            (border ?r1 ?r2))
logic> (fact (connected ?r1 ?r2)
            (border ?r1 ?next)
            (connected ?next ?r2))
```
Recursive Facts

- A **recursive fact** uses the same relation in the conclusion and one or more hypotheses.
- Just like in imperative programming, we need a **base fact** that stops the recursion.

```
logic> (fact (connected ?r1 ?r2)
  (border ?r1 ?r2))
```

```
logic> (fact (connected ?r1 ?r2)
  (border ?r1 ?next)
  (connected ?next ?r2))
```
Recursive Facts

- A **recursive fact** uses the same relation in the conclusion and one or more hypotheses.

- Just like in imperative programming, we need a **base fact** that stops the recursion.

```logic
(fact (connected ?r1 ?r2)
     (border ?r1 ?r2))
```

```logic
(fact (connected ?r1 ?r2)
     (border ?r1 ?next)
     (connected ?next ?r2))
```

```logic
(fact (border V T))
```
Recursive Facts

- A recursive fact uses the same relation in the conclusion and one or more hypotheses.
- Just like in imperative programming, we need a base fact that stops the recursion.

```logic
(fact (connected ?r1 ?r2) (border ?r1 ?r2))
(fact (connected ?r1 ?r2) (border ?r1 ?next) (connected ?next ?r2))
(fact (border V T))
(query (two-away NT T))
```
Recursive Facts

- A **recursive fact** uses the same relation in the conclusion and one or more hypotheses.

- Just like in imperative programming, we need a **base fact** that stops the recursion.

```
logic> (fact (connected ?r1 ?r2) 
  (border ?r1 ?r2))
logic> (fact (connected ?r1 ?r2) 
  (border ?r1 ?next) 
  (connected ?next ?r2))
logic> (fact (border V T))
logic> (query (two-away NT T))
```

Failed.
Recursive Facts

- A **recursive fact** uses the same relation in the conclusion and one or more hypotheses.
- Just like in imperative programming, we need a **base fact** that stops the recursion.

```logic
(fact (connected ?r1 ?r2)
     (border ?r1 ?r2))
```

```logic
(fact (connected ?r1 ?r2)
     (border ?r1 ?next)
     (connected ?next ?r2))
```

```logic
(fact (border V T))
```

```logic
(query (two-away NT T))
```

Failed.

```logic
(query (connected NT T))
```
Recursive Facts

- A **recursive fact** uses the same relation in the conclusion and one or more hypotheses.
- Just like in imperative programming, we need a **base fact** that stops the recursion.

```
logic> (fact (connected ?r1 ?r2)
  (border ?r1 ?r2))
logic> (fact (connected ?r1 ?r2)
  (border ?r1 ?next)
  (connected ?next ?r2))
logic> (fact (border V T))
logic> (query (two-away NT T))
Failed.
logic> (query (connected NT T))
Success!
```
Recursive Facts
Recursive Facts

- The Logic interpreter performs a search in the space of relations for each query to find satisfying assignments.
Recursive Facts

• The Logic interpreter performs a search in the space of relations for each query to find satisfying assignments
Recursive Facts

- The Logic interpreter performs a search in the space of relations for each query to find satisfying assignments

```
logic> (fact (connected ?r1 ?r2)
(border ?r1 ?r2))
logic> (fact (connected ?r1 ?r2)
(border ?r1 ?next)
(connected ?next ?r2))
```
Recursive Facts

- The Logic interpreter performs a search in the space of relations for each query to find satisfying assignments.

\[
\text{logic}> \ (\text{fact} \ (\text{connected} \ ?r1 \ ?r2) \\
\quad \ (\text{border} \ ?r1 \ ?r2))
\]

\[
\text{logic}> \ (\text{fact} \ (\text{connected} \ ?r1 \ ?r2) \\
\quad \ (\text{border} \ ?r1 \ ?\text{next}) \\
\quad \ (\text{connected} \ ?\text{next} \ ?r2))
\]

\[
\text{logic}> \ (\text{query} \ (\text{connected} \ \text{NT} \ \text{T}))
\]

Success!
Recursive Facts

• The Logic interpreter performs a search in the space of relations for each query to find satisfying assignments

```
logic> (fact (connected ?r1 ?r2)
             (border ?r1 ?r2))
logic> (fact (connected ?r1 ?r2)
             (border ?r1 ?next)
             (connected ?next ?r2))
logic> (query (connected NT T))
Success!
```

(border NT SA)
Recursive Facts

- The Logic interpreter performs a search in the space of relations for each query to find satisfying assignments.

```logic
logic> (fact (connected ?r1 ?r2)
           (border ?r1 ?r2))
logic> (fact (connected ?r1 ?r2)
           (border ?r1 ?next)
           (connected ?next ?r2))
logic> (query (connected NT T))
Success!

(border NT SA)
(border SA V)
```
Recursive Facts

- The Logic interpreter performs a search in the space of relations for each query to find satisfying assignments.

\[
\text{logic> (fact (connected ?r1 ?r2))}
\]

\[
\text{logic> (fact (connected ?r1 ?r2) (border ?r1 ?next) (connected ?next ?r2))}
\]

\[
\text{logic> (query (connected NT T))}
\]

Success!

\[
\text{(border NT SA)}
\]

\[
\text{(border SA V)}
\]

\[
\text{(border V T)}
\]
Recursive Facts

- The Logic interpreter performs a search in the space of relations for each query to find satisfying assignments.

\[\text{logic}> (\text{fact} \ (\text{connected} \ ?r1 \ ?r2) \ (\text{border} \ ?r1 \ ?r2))\]

\[\text{logic}> (\text{fact} \ (\text{connected} \ ?r1 \ ?r2) \ (\text{border} \ ?r1 \ ?\text{next}) \ (\text{connected} \ ?\text{next} \ ?r2))\]

\[\text{logic}> (\text{query} \ (\text{connected} \ \text{NT} \ \text{T}))\]

Success!

\(\text{(border NT SA)}\)

\(\text{(border SA V)}\)

\(\text{(border V T) } \Rightarrow \text{(connected V T)}\)
Recursive Facts

- The Logic interpreter performs a search in the space of relations for each query to find satisfying assignments

```
logic> (fact (connected ?r1 ?r2) (border ?r1 ?r2))
logic> (fact (connected ?r1 ?r2) (border ?r1 ?next) (connected ?next ?r2))
logic> (query (connected NT T))
Success!
```

```
(border SA V) => (connected SA T)
(border V T) => (connected V T)
```
Recursive Facts

- The Logic interpreter performs a search in the space of relations for each query to find satisfying assignments

```logic
logic> (fact (connected ?r1 ?r2) (border ?r1 ?r2))
logic> (fact (connected ?r1 ?r2) (border ?r1 ?next) (connected ?next ?r2))
logic> (query (connected NT T))
Success!
```

\[
\begin{align*}
(border \ SA \ V) & \Rightarrow (connected \ SA \ T) \\
(border \ V \ T) & \Rightarrow (connected \ V \ T)
\end{align*}
\]
Hierarchical Facts
Hierarchical Facts

• Relations can also contain lists in addition to symbols
Hierarchical Facts

- Relations can also contain lists in addition to symbols
  \[\text{fact (australia (NSW NT Q SA T WA V))}\]
Hierarchical Facts

• Relations can also contain lists in addition to symbols
  
  \[
  \text{fact (australia (NSW NT Q SA T WA V))}
  \]
  
symbol
Hierarchical Facts

- Relations can also contain lists in addition to symbols

\[
\text{(fact (australia (NSW NT Q SA T WA V)))}
\]

- symbol
- list of symbols
Hierarchical Facts

• Relations can also contain lists in addition to symbols
  (fact (australia (NSW NT Q SA T WA V)))

• The fancy name for this is hierarchy, but it's not a fancy or complex idea
Hierarchical Facts

- Relations can also contain lists in addition to symbols
  \[(\text{fact (australia (NSW NT Q SA T WA V))})\]
  symbol \text{symbol} \text{list of symbols}
- The fancy name for this is \textit{hierarchy}, but it’s not a fancy or complex idea
- Variables can refer to either symbols or lists of symbols
Hierarchical Facts

- Relations can also contain lists in addition to symbols
  \[\text{fact (australia (NSW NT Q SA T WA V))}\]
  symbol list of symbols

- The fancy name for this is *hierarchy*, but it’s not a fancy or complex idea

- Variables can refer to either symbols or lists of symbols
  \[\text{logic> (query (australia ?regions))}\]
Hierarchical Facts

• Relations can also contain lists in addition to symbols

  (fact (australia (NSW NT Q SA T WA V)))

  symbol ____________ list of symbols

• The fancy name for this is hierarchy, but it’s not a fancy or complex idea

• Variables can refer to either symbols or lists of symbols

  logic> (query (australia ?regions))

  Success!
Hierarchical Facts

- Relations can also contain lists in addition to symbols
  
  ```lisp
  (fact (australia (NSW NT Q SA T WA V)))
  ```

- The fancy name for this is hierarchy, but it’s not a fancy or complex idea

- Variables can refer to either symbols or lists of symbols
  
  ```lisp
  logic> (query (australia ?regions))
  Success!
  regions: (nsw nt q sa t wa v)
  ```
Hierarchical Facts

- Relations can also contain lists in addition to symbols
  
  \[
  \text{fact (australia (NSW NT Q SA T WA V)))}
  \]

- The fancy name for this is hierarchy, but it's not a fancy or complex idea

- Variables can refer to either symbols or lists of symbols

```logic>
(query (australia ?regions))
Success!
regions: (nsw nt q sa t wa v)
logic> (query (australia (?first . ?rest)))
```
Hierarchical Facts

- Relations can also contain lists in addition to symbols
  
  \[ \text{fact (australia (NSW NT Q SA T WA V))} \]

  symbol \underline{list of symbols}

- The fancy name for this is hierarchy, but it’s not a fancy or complex idea

- Variables can refer to either symbols or lists of symbols

  logic> (query (australia ?regions))
  Success!
  regions: (nsw nt q sa t wa v)

  logic> (query (australia (?first . ?rest)))
  Success!
Hierarchical Facts

• Relations can also contain lists in addition to symbols
  \[ \text{(fact (australia (NSW NT Q SA T WA V)))} \]
  symbol \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
  list of symbols

• The fancy name for this is hierarchy, but it’s not a fancy or complex idea

• Variables can refer to either symbols or lists of symbols
  \[
  \text{logic> (query (australia ?regions))} \\
  \text{Success!} \\
  \text{regions: (nsw nt q sa t wa v)} \\
  \text{logic> (query (australia (?first . ?rest)))} \\
  \text{Success!} \\
  \text{first: nsw rest: (nt q sa t wa v)}
  \]
Hierarchical Facts

- Relations can also contain lists in addition to symbols
  \[(\text{fact } (\text{australia } (\text{NSW NT Q SA T WA V})))\]
  symbol list of symbols

- The fancy name for this is \textit{hierarchy}, but it's not a fancy or complex idea

- Variables can refer to either symbols or lists of symbols
  \begin{verbatim}
  logic> (query (australia ?regions))
  Success!
  regions: (nsw nt q sa t wa v)
  logic> (query (australia (?first . ?rest)))
  Success!
  first: nsw  rest: (nt q sa t wa v)
  \end{verbatim}

- Why the dot? Because we are using Scheme lists, \((\text{nsw nt q sa t wa v})\) is the same as \((\text{nsw . (nt q sa t wa v)})\)
Hierarchical Facts

• Relations can also contain lists in addition to symbols
  \[(\text{fact } (\text{australia} (\text{NSW NT Q SA T WA V})))\]
  symbol \quad list of symbols

• The fancy name for this is \textit{hierarchy}, but it’s not a fancy or complex idea

• Variables can refer to either symbols or lists of symbols
  \begin{verbatim}
  logic> (query (australia ?regions))
  Success!
  regions: (nsw nt q sa t wa v)
  logic> (query (australia (?first . ?rest)))
  Success!
  first: nsw rest: (nt q sa t wa v)
  \end{verbatim}

• Why the dot? Because we are using Scheme lists,
  \[(\text{nsw nt q sa t wa v})\text{ is the same as } (\text{nsw } . (\text{nt q sa t wa v}))\]
  first
Hierarchical Facts

- Relations can also contain lists in addition to symbols

  ```scheme
  (fact (australia (NSW NT Q SA T WA V)))
  ```

- The fancy name for this is hierarchy, but it's not a fancy or complex idea

- Variables can refer to either symbols or lists of symbols

  ```scheme
  logic> (query (australia ?regions))
  Success!
  regions: (nsw nt q sa t wa v)
  logic> (query (australia (?first . ?rest)))
  Success!
  first: nsw  rest: (nt q sa t wa v)
  ```

- Why the dot? Because we are using Scheme lists, (nsw nt q sa t wa v) is the same as (nsw . (nt q sa t wa v))
Example: Membership
Example: Membership

• Recursive and hierarchical facts allow us to solve some interesting problems in Logic
Example: Membership

• Recursive and hierarchical facts allow us to solve some interesting problems in Logic

• As a first example, let’s declare facts for membership of an element in a list
Example: Membership

- Recursive and hierarchical facts allow us to solve some interesting problems in Logic
- As a first example, let’s declare facts for membership of an element in a list

```
logic> (query (in 1 (1 2 3 4)))
```
Example: Membership

- Recursive and hierarchical facts allow us to solve some interesting problems in Logic.
- As a first example, let’s declare facts for membership of an element in a list.

```
logic> (query (in 1 (1 2 3 4)))
Success!
```
Example: Membership

- Recursive and hierarchical facts allow us to solve some interesting problems in Logic
- As a first example, let’s declare facts for membership of an element in a list

```
logic> (query (in 1 (1 2 3 4)))
Success!
logic> (query (in 5 (1 2 3 4)))
```
Example: Membership

- Recursive and hierarchical facts allow us to solve some interesting problems in Logic

- As a first example, let’s declare facts for membership of an element in a list

```logic
logic> (query (in 1 (1 2 3 4)))
Success!
logic> (query (in 5 (1 2 3 4)))
Failed.
```
Example: Membership

- Recursive and hierarchical facts allow us to solve some interesting problems in Logic
- As a first example, let’s declare facts for membership of an element in a list

```
logic> (query (in 1 (1 2 3 4)))
Success!
logic> (query (in 5 (1 2 3 4)))
Failed.
```
Example: Membership

- Recursive and hierarchical facts allow us to solve some interesting problems in Logic
- As a first example, let’s declare facts for membership of an element in a list

  \[
  \text{logic}> \ (\text{fact} \ (\text{in} \ ?\text{elem} \ (?\text{elem} \ . \ ?\text{rest})))
  \]

  \[
  \text{logic}> \ (\text{fact} \ (\text{in} \ ?\text{elem} \ (?\text{first} \ . \ ?\text{rest}))
  \]

  \[
  (\text{in} \ ?\text{elem} \ ?\text{rest})
  \]

  \[
  \text{logic}> \ (\text{query} \ (\text{in} \ 1 \ (1 \ 2 \ 3 \ 4)))
  \]

  Success!

  \[
  \text{logic}> \ (\text{query} \ (\text{in} \ 5 \ (1 \ 2 \ 3 \ 4)))
  \]

  Failed.
Recursive and hierarchical facts allow us to solve some interesting problems in Logic.

As a first example, let’s declare facts for membership of an element in a list:

```
logic> (fact (in ?elem (?first . ?rest)))
logic> (fact (in ?elem (?first . ?rest))
            (in ?elem ?rest))
logic> (query (in 1 (1 2 3 4)))
Success!
logic> (query (in 5 (1 2 3 4)))
Failed.
logic> (query (in ?x (1 2 3 4)))
Success!
```

x: 1
x: 2
x: 3
x: 4
Example: Appending Lists
Example: Appending Lists

• Let’s declare facts for appending two lists together to form a third list
Example: Appending Lists

- Let’s declare facts for appending two lists together to form a third list

```
logic> (query (append (1 2) (3 4) (1 2 3 4)))
```
Example: Appending Lists

Let’s declare facts for appending two lists together to form a third list

```
logic> (query (append (1 2) (3 4) (1 2 3 4)))
Success!
```
Example: Appending Lists

- Let’s declare facts for appending two lists together to form a third list

    logic> (query (append (1 2) (3 4) (1 2 3 4)))
    Success!
    logic> (query (append (1 2) (3 4 5) (1 2 3 4)))
Example: Appending Lists

Let’s declare facts for appending two lists together to form a third list

```
logic> (query (append (1 2) (3 4) (1 2 3 4)))
Success!
logic> (query (append (1 2) (3 4 5) (1 2 3 4)))
Failed.
```
Example: Appending Lists

- Let’s declare facts for appending two lists together to form a third list

```
logic> (query (append (1 2) (3 4) (1 2 3 4)))
Success!
logic> (query (append (1 2) (3 4 5) (1 2 3 4)))
Failed.
```
Example: Appending Lists

• Let’s declare facts for appending two lists together to form a third list

```
logic> (fact (append () ?lst ?lst))
logic> (fact (append (?first . ?rest) ?lst (?first . ?rest+1st))
        (append ?rest ?lst ?rest+1st))
logic> (query (append (1 2) (3 4) (1 2 3 4)))
Success!
logic> (query (append (1 2) (3 4 5) (1 2 3 4)))
Failed.
```
Example: Appending Lists  

- Let’s declare facts for appending two lists together to form a third list

```
logic> (fact (append () ?lst1 ?lst2))
logic> (fact (append (?first . ?rest) ?lst (?first . ?rest+lst))
  (append ?rest ?lst ?rest+lst))
logic> (query (append (1 2) (3 4) (1 2 3 4)))
Success!
logic> (query (append (1 2) (3 4 5) (1 2 3 4)))
Failed.
logic> (query (append ?lst1 ?lst2 (1 2 3 4)))
Success!
lst1: ()    lst2: (1 2 3 4)
lst1: (1)   lst2: (2 3 4)
lst1: (1 2) lst2: (3 4)
lst1: (1 2 3) lst2: (4)
lst1: (1 2 3 4) lst2: ()
```
Let’s Color Australia

In two different ways
Map Coloring Way #1
Map Coloring Way #1

• Idea: Create a variable for the color of each region
Map Coloring Way #1

- Idea: Create a variable for the color of each region
  - We have to make sure each variable is assigned to one of the symbols red, green, or blue
Map Coloring Way #1

• Idea: Create a variable for the color of each region
  • We have to make sure each variable is assigned to one of the symbols red, green, or blue
  • Then, we have to make sure the variables for bordering regions are not equal
Map Coloring Way #1

• Idea: Create a variable for the color of each region
  • We have to make sure each variable is assigned to one of the symbols red, green, or blue
  • Then, we have to make sure the variables for bordering regions are not equal

• We can pretty closely follow what we wrote at the beginning of lecture:
Map Coloring Way #1

- Idea: Create a variable for the color of each region
  - We have to make sure each variable is assigned to one of the symbols red, green, or blue
  - Then, we have to make sure the variables for bordering regions are not equal

- We can pretty closely follow what we wrote at the beginning of lecture:

```python
# Declarative map coloring idea:
Find a solution where:
- All regions of the map are colored
- No neighboring regions have the same color
```
Map Coloring Way #1

- Idea: Create a variable for the color of each region
  - We have to make sure each variable is assigned to one of the symbols red, green, or blue
  - Then, we have to make sure the variables for bordering regions are not equal

- We can pretty closely follow what we wrote at the beginning of lecture:

  # Declarative map coloring idea:
  Find a solution where:
  - All regions of the map are colored
  - No neighboring regions have the same color
Map Coloring Way #1
Map Coloring Way #1

Find a solution where:
- All regions of the map are colored
- No neighboring regions have the same color
Map Coloring Way #1

Find a solution where:
- All regions of the map are colored
- No neighboring regions have the same color

logic> (query (in ?NSW (red green blue))
  (in ?NT (red green blue))
  (in ?Q (red green blue))
  (in ?SA (red green blue))
  (in ?T (red green blue))
  (in ?V (red green blue))
  (in ?WA (red green blue))
  (not (equal ?NSW ?Q))
  (not (equal ?NSW ?SA))
  (not (equal ?NSW ?V))
  (not (equal ?NT ?Q))
  (not (equal ?NT ?SA))
  (not (equal ?NT ?WA))
  (not (equal ?Q ?SA))
  (not (equal ?SA ?WA))
  (not (equal ?SA ?V))))
Map Coloring Way #1

Find a solution where:
- All regions of the map are colored
- No neighboring regions have the same color

```
logic> (query
  (in ?NSW (red green blue))
  (in ?NT (red green blue))
  (in ?Q (red green blue))
  (in ?SA (red green blue))
  (in ?T (red green blue))
  (in ?V (red green blue))
  (in ?WA (red green blue))
  (not (equal ?NSW ?Q))
  (not (equal ?NSW ?SA))
  (not (equal ?NSW ?V))
  (not (equal ?NT ?Q))
  (not (equal ?NT ?SA))
  (not (equal ?NT ?WA))
  (not (equal ?Q ?SA))
  (not (equal ?SA ?WA))
  (not (equal ?SA ?V)))
```
Map Coloring Way #1

Find a solution where:
- All regions of the map are colored
- No neighboring regions have the same color

logic> (query
(in ?NSW (red green blue))
(in ?NT (red green blue))
(in ?Q (red green blue))
(in ?Q (red green blue))
(in ?SA (red green blue))
(in ?T (red green blue))
(in ?V (red green blue))
(in ?WA (red green blue))
(not (equal ?NSW ?Q))
(not (equal ?NSW ?SA))
(not (equal ?NSW ?V))
(not (equal ?NT ?Q))
(not (equal ?NT ?SA))
(not (equal ?NT ?WA))
(not (equal ?Q ?SA))
(not (equal ?SA ?WA))
(not (equal ?SA ?V)))
Map Coloring Way #2
Map Coloring Way #2

- Solution #1 was simple and allowed us to directly follow our original idea
Map Coloring Way #2

• Solution #1 was simple and allowed us to directly follow our original idea
• However, it wasn’t an elegant or efficient solution
Map Coloring Way #2

- Solution #1 was simple and allowed us to directly follow our original idea
- However, it wasn’t an elegant or efficient solution
  - Lots of repetition
Map Coloring Way #2

• Solution #1 was simple and allowed us to directly follow our original idea
• However, it wasn’t an elegant or efficient solution
  • Lots of repetition
  • No separation between *data and program*
Map Coloring Way #2

• Solution #1 was simple and allowed us to directly follow our original idea

• However, it wasn’t an elegant or efficient solution
  • Lots of repetition
  • No separation between data and program
  • As a result, only works for this specific map
Map Coloring Way #2

- Solution #1 was simple and allowed us to directly follow our original idea
- However, it wasn’t an elegant or efficient solution
  - Lots of repetition
  - No separation between data and program
  - As a result, only works for this specific map
- Let’s look at a more complicated and clever solution
Map Coloring Way #2

- Solution #1 was simple and allowed us to directly follow our original idea
- However, it wasn’t an elegant or efficient solution
  - Lots of repetition
  - No separation between *data and program*
  - As a result, only works for this specific map
- Let’s look at a more complicated and clever solution
  - We will first declare our data, which is our map
Map Coloring Way #2

- Solution #1 was simple and allowed us to directly follow our original idea
- However, it wasn’t an elegant or efficient solution
  - Lots of repetition
  - No separation between data and program
  - As a result, only works for this specific map
- Let’s look at a more complicated and clever solution
  - We will first declare our data, which is our map
  - We will then try and find assignments with no conflicts
Map Coloring Way #2

- Solution #1 was simple and allowed us to directly follow our original idea
- However, it wasn’t an elegant or efficient solution
  - Lots of repetition
  - No separation between data and program
  - As a result, only works for this specific map
- Let’s look at a more complicated and clever solution
  - We will first declare our data, which is our map
  - We will then try and find assignments with no conflicts
  - This way, our program does not repeat itself, and will be general to any map!
• Solution #1 was simple and allowed us to directly follow our original idea
• However, it wasn’t an elegant or efficient solution
  • Lots of repetition
  • No separation between data and program
  • As a result, only works for this specific map
• Let’s look at a more complicated and clever solution
  • We will first declare our data, which is our map
  • We will then try and find assignments with no conflicts
  • This way, our program does not repeat itself, and will be general to any map!
Summary
Summary

• We learned about **declarative programming** today
Summary

• We learned about *declarative programming* today
  • A completely different programming paradigm where our programs specify what properties solutions should satisfy rather than how to find a solution
Summary

• We learned about declarative programming today
  • A completely different programming paradigm where our programs specify what properties solutions should satisfy rather than how to find a solution
  • This allows us to solve some problems in an easier and more intuitive manner
Summary

• We learned about declarative programming today
  • A completely different programming paradigm where our programs specify what properties solutions should satisfy rather than how to find a solution
  • This allows us to solve some problems in an easier and more intuitive manner
• We learned Logic, a declarative language
Summary

• We learned about *declarative programming* today
  • A completely different programming paradigm where our programs specify what properties solutions should satisfy rather than how to find a solution
  • This allows us to solve some problems in an easier and more intuitive manner
• We learned Logic, a declarative language
  • Logic consists of facts, which declare relations that are true, and queries, which ask if relations are true
Summary

• We learned about **declarative programming** today
  • A completely different programming paradigm where our programs specify what properties solutions should satisfy rather than how to find a solution
  • This allows us to solve some problems in an easier and more intuitive manner
• We learned **Logic**, a declarative language
  • Logic consists of facts, which declare relations that are true, and queries, which ask if relations are true
  • Recursive and hierarchical facts allow us to solve many interesting problems
Summary

• We learned about *declarative programming* today
  • A completely different programming paradigm where our programs specify what properties solutions should satisfy rather than how to find a solution
  • This allows us to solve some problems in an easier and more intuitive manner
• We learned Logic, a declarative language
  • Logic consists of facts, which declare relations that are true, and queries, which ask if relations are true
  • Recursive and hierarchical facts allow us to solve many interesting problems
• This is very different idea, so you’ll have to practice!