Functional Programming in Scala

Javier Fuentes (@javifdev)
Habla Computing

What Is Functional Programming?

- Functional programming is programming with pure functions
- A pure function is just a mapping from input values to output values, that does nothing else
- A pure function doesn’t have side effects

Goals

- Know the design patterns that allow us to purify side-effectful programs
- Understand the reasoning behind functional structures
- Be able to build these patterns from scratch
- Use trending functional libraries that implements these patterns for us

Outline

- Purifying process walkthrough
- Sugaring process walkthrough
- Bonus track: Bundle up your library
- Functional libraries (Scalaz/Cats)
- Conclusions

The Functional Way: Interpreter Pattern

- Programs
  - Just pure immutable values
  - Describe our business logic
  - WHAT
- Interpreters
  - Give meaning to our programs
  - Execute effects
  - Multiple interpreters for same program
  - HOW

Purifying Process Walkthrough

- **Step 1: Ad Hoc programs**
- Step 2: Parameterize your programs
- Step 3: Sequence programs
- Step 4: Sequence context-dependent programs
- Step 5: Add pure computations

Purifying Process Walkthrough

- Step 1: Ad Hoc programs
- **Step 2: Parameterize your programs**
- Step 3: Sequence programs
- Step 4: Sequence context-dependent programs
- Step 5: Add pure computations

Purifying Process Walkthrough

- Step 1: Ad Hoc programs
- Step 2: Parameterize your programs
- **Step 3: Sequence programs**
- Step 4: Sequence context-dependent programs
- Step 5: Add pure computations

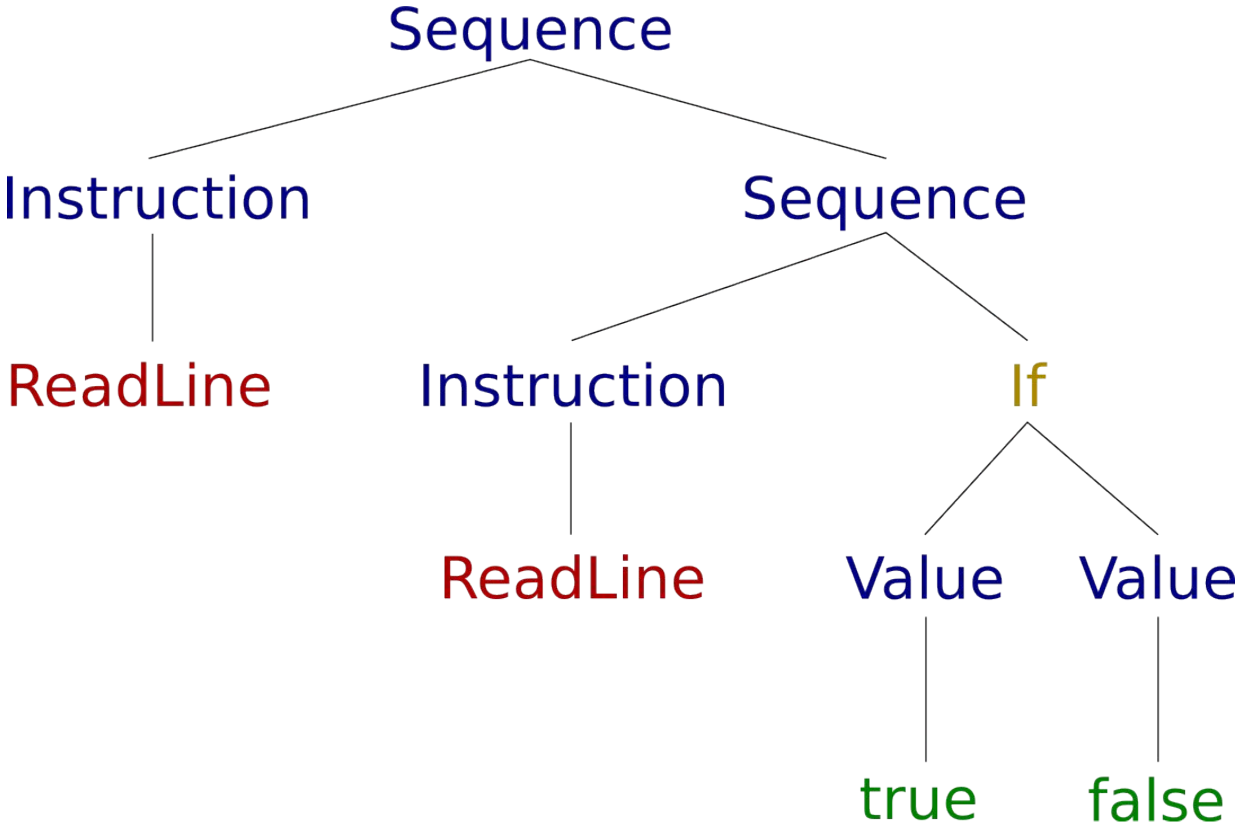
Purifying Process Walkthrough

- Step 1: Ad Hoc programs
- Step 2: Parameterize your programs
- Step 3: Sequence programs
- **Step 4: Sequence context-dependent programs**
- Step 5: Add pure computations

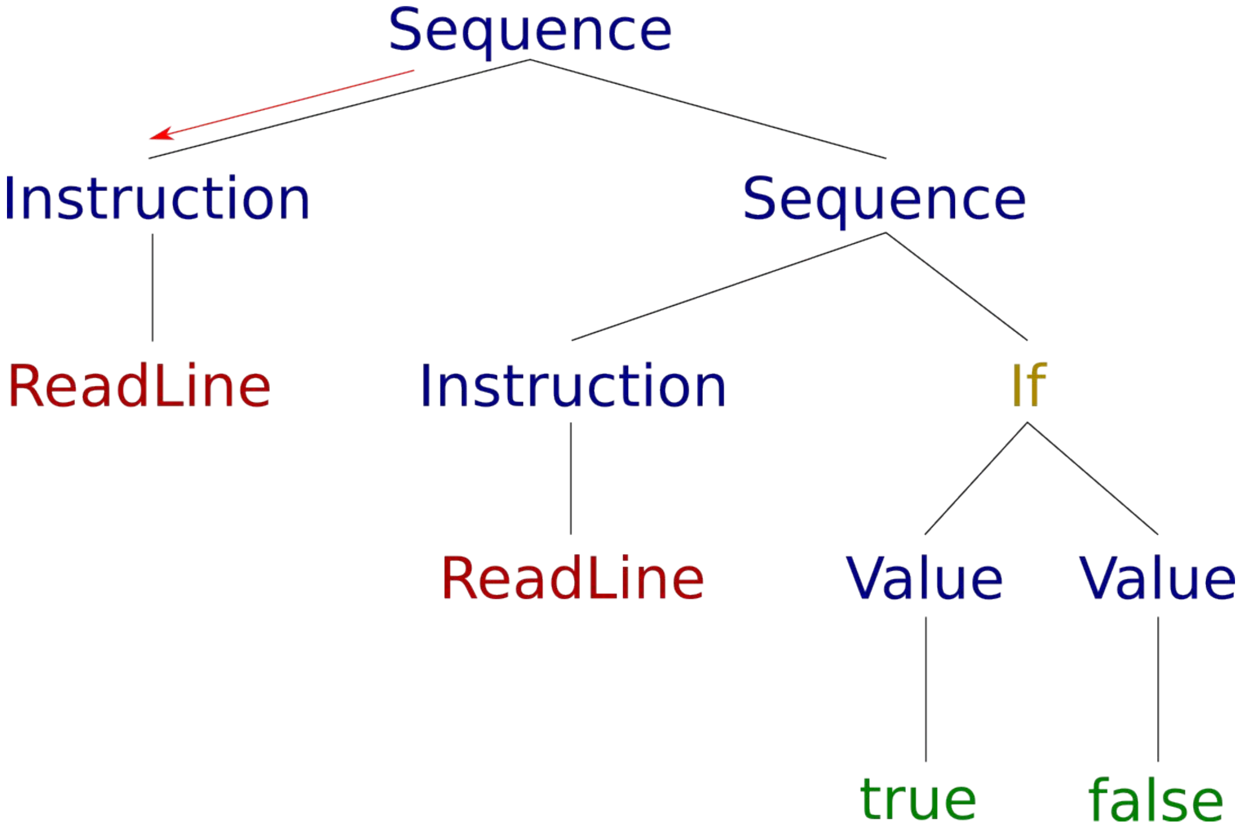
Purifying Process Walkthrough

- Step 1: Ad Hoc programs
- Step 2: Parameterize your programs
- Step 3: Sequence programs
- Step 4: Sequence context-dependent programs
- **Step 5: Add pure computations**

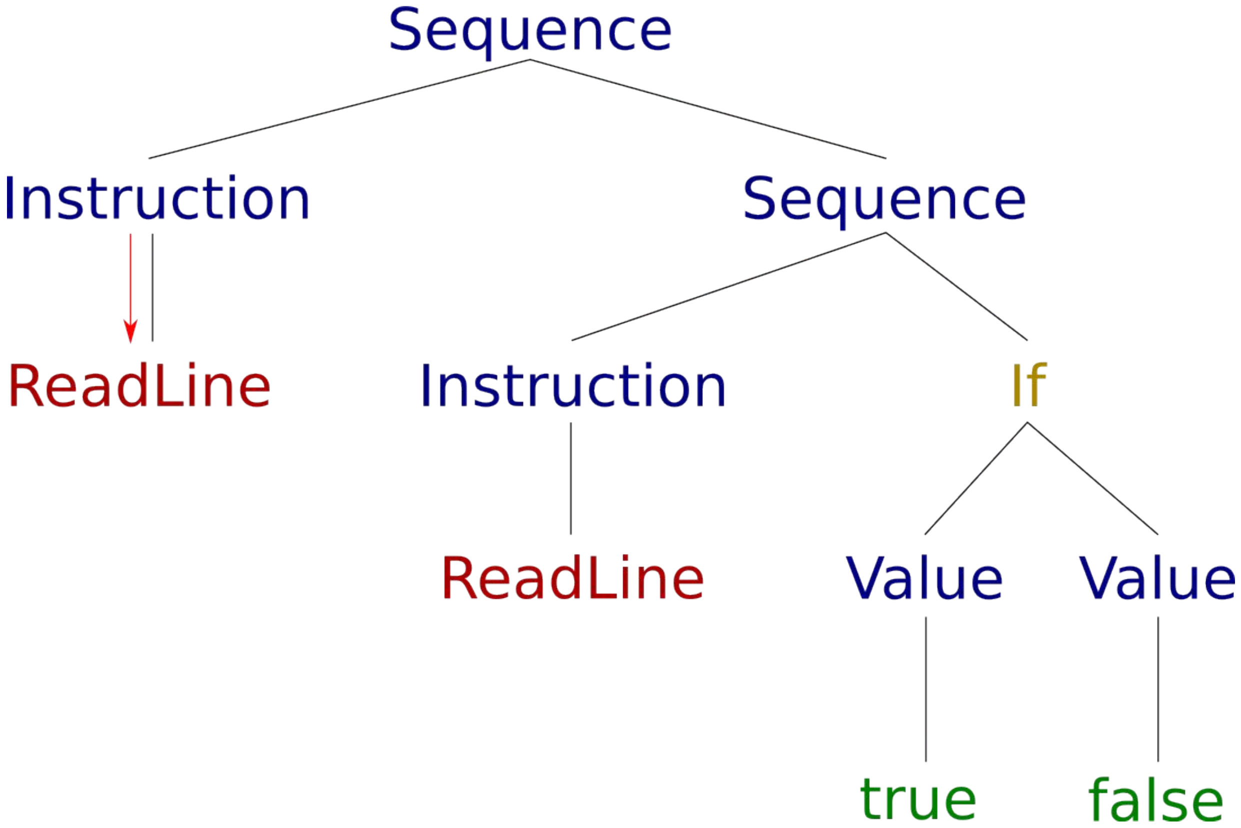
Interpreting a Program



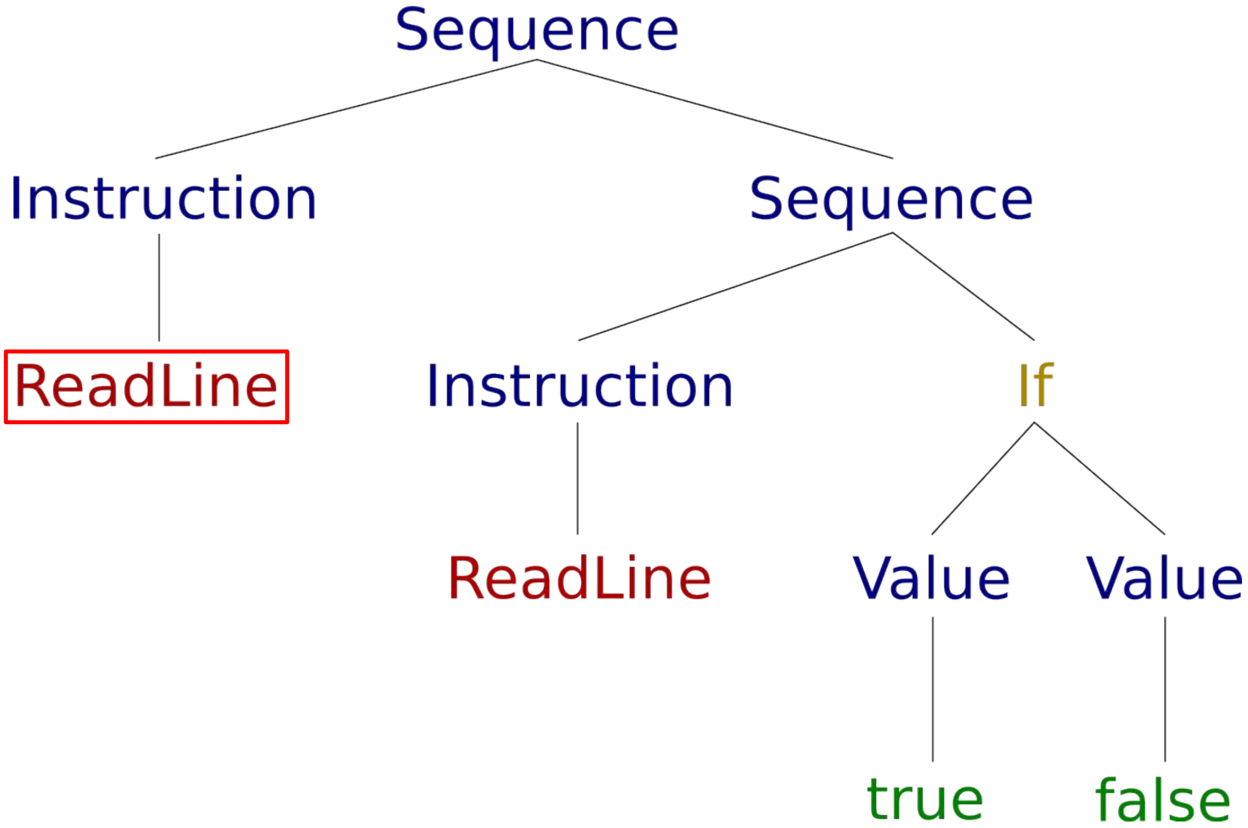
Interpreting a Program



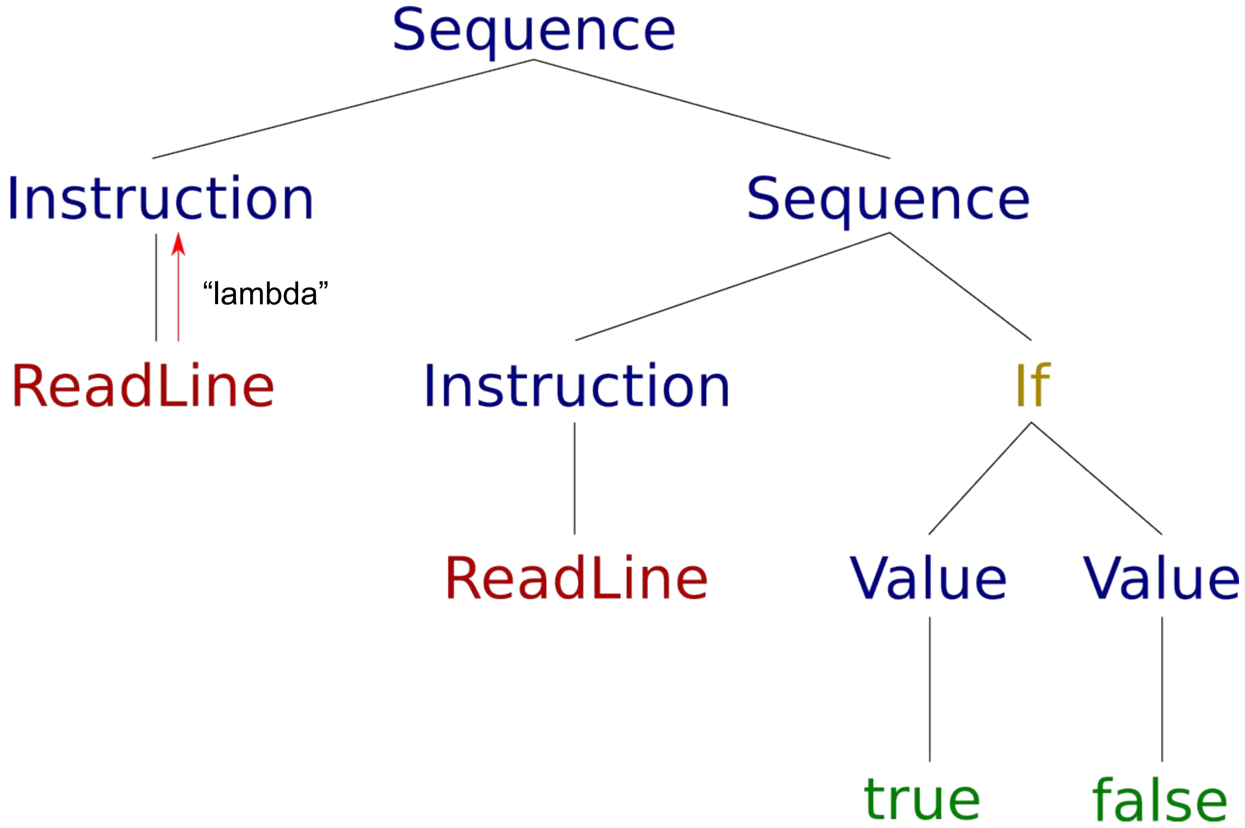
Interpreting a Program



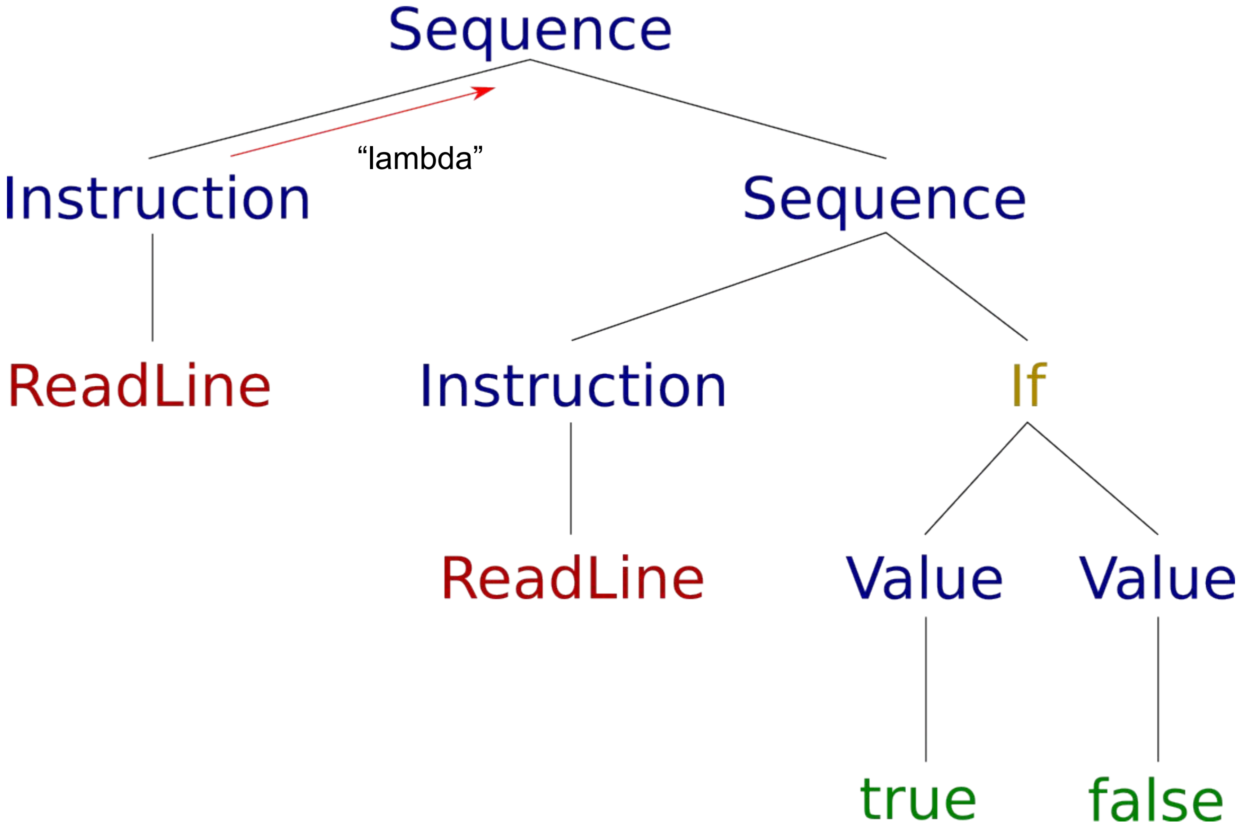
Interpreting a Program



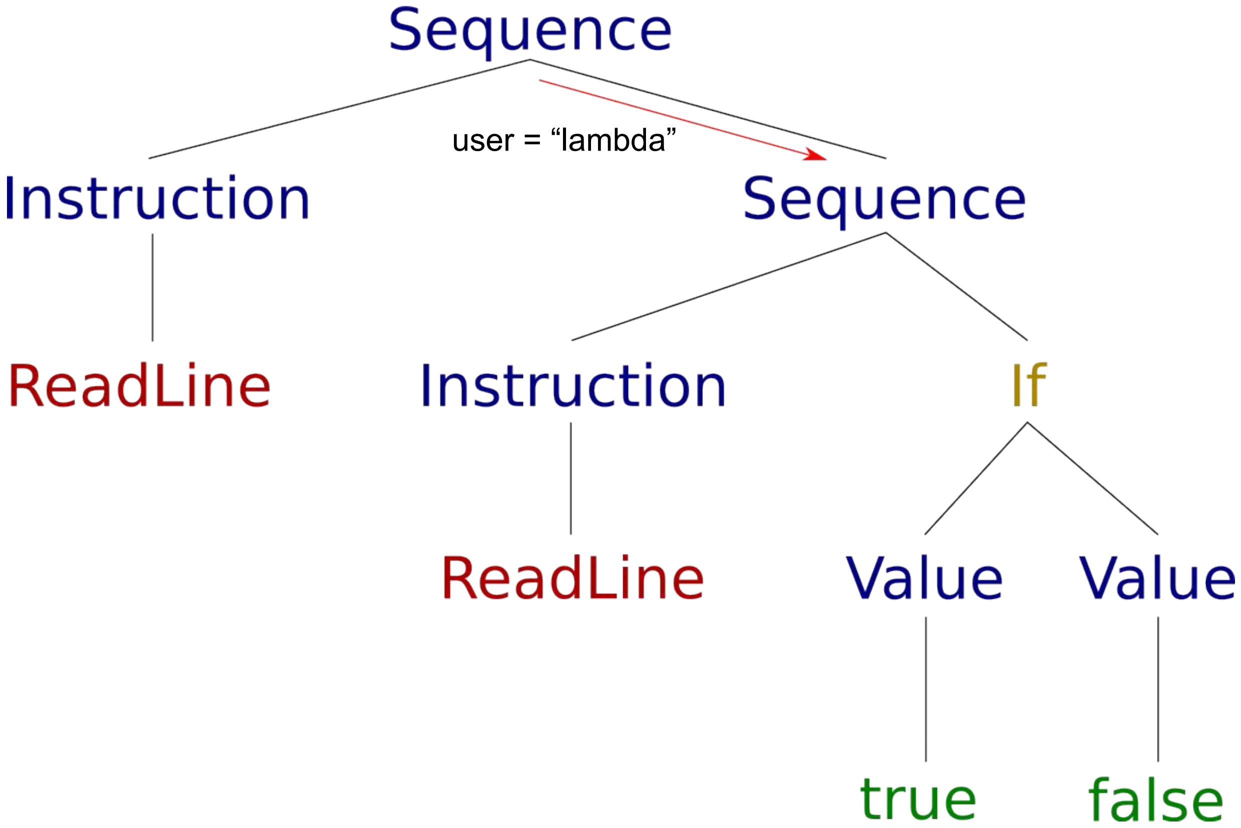
Interpreting a Program



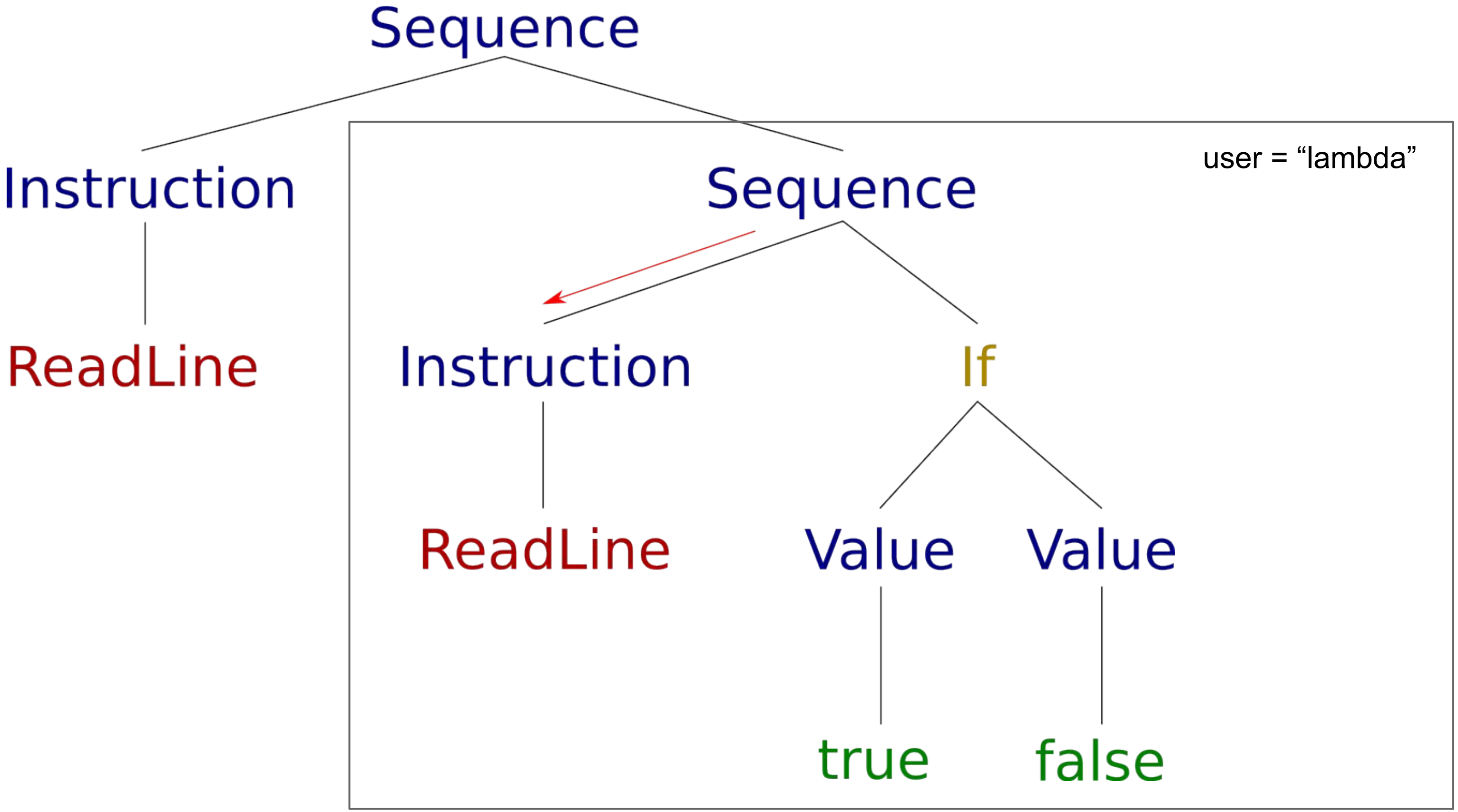
Interpreting a Program



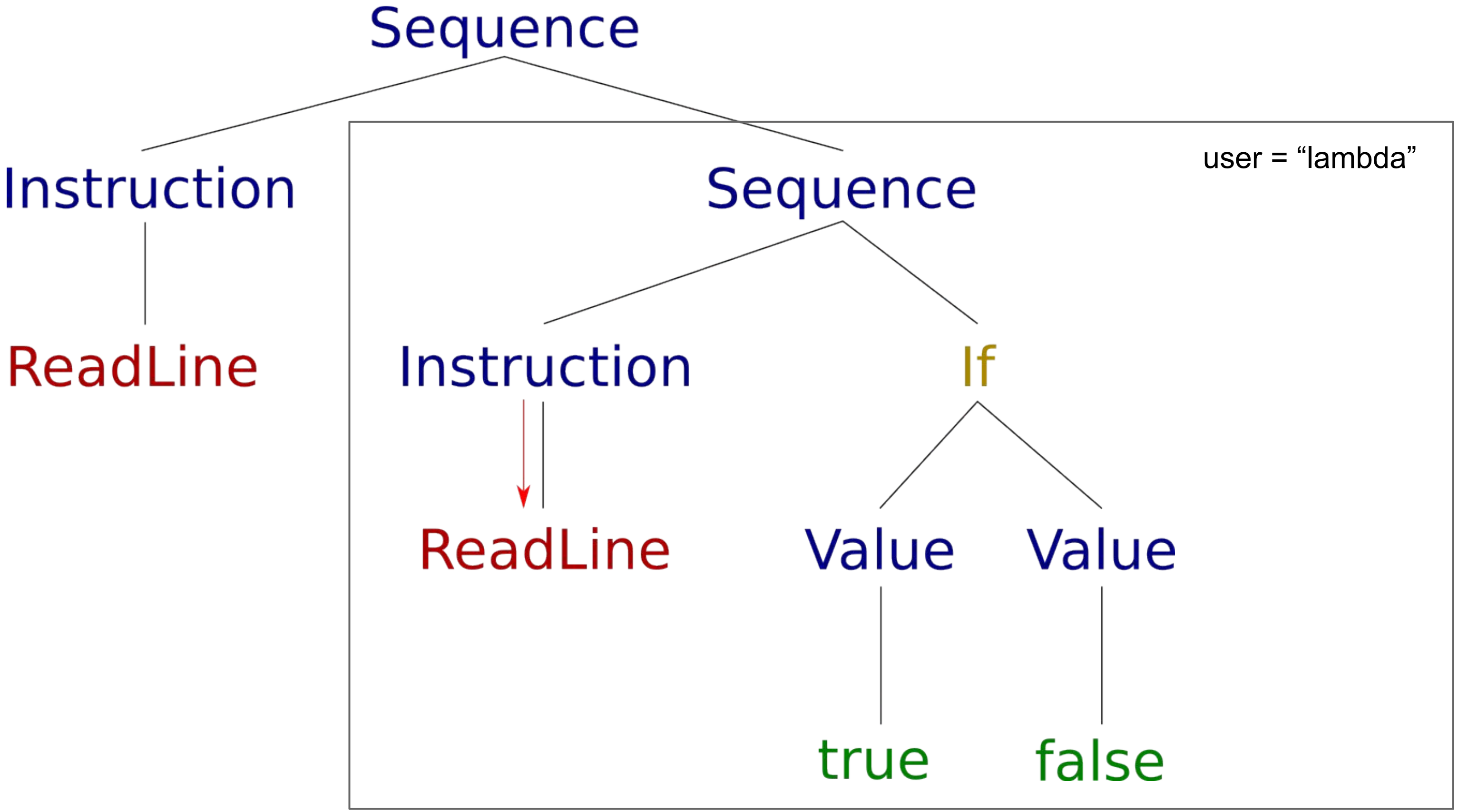
Interpreting a Program



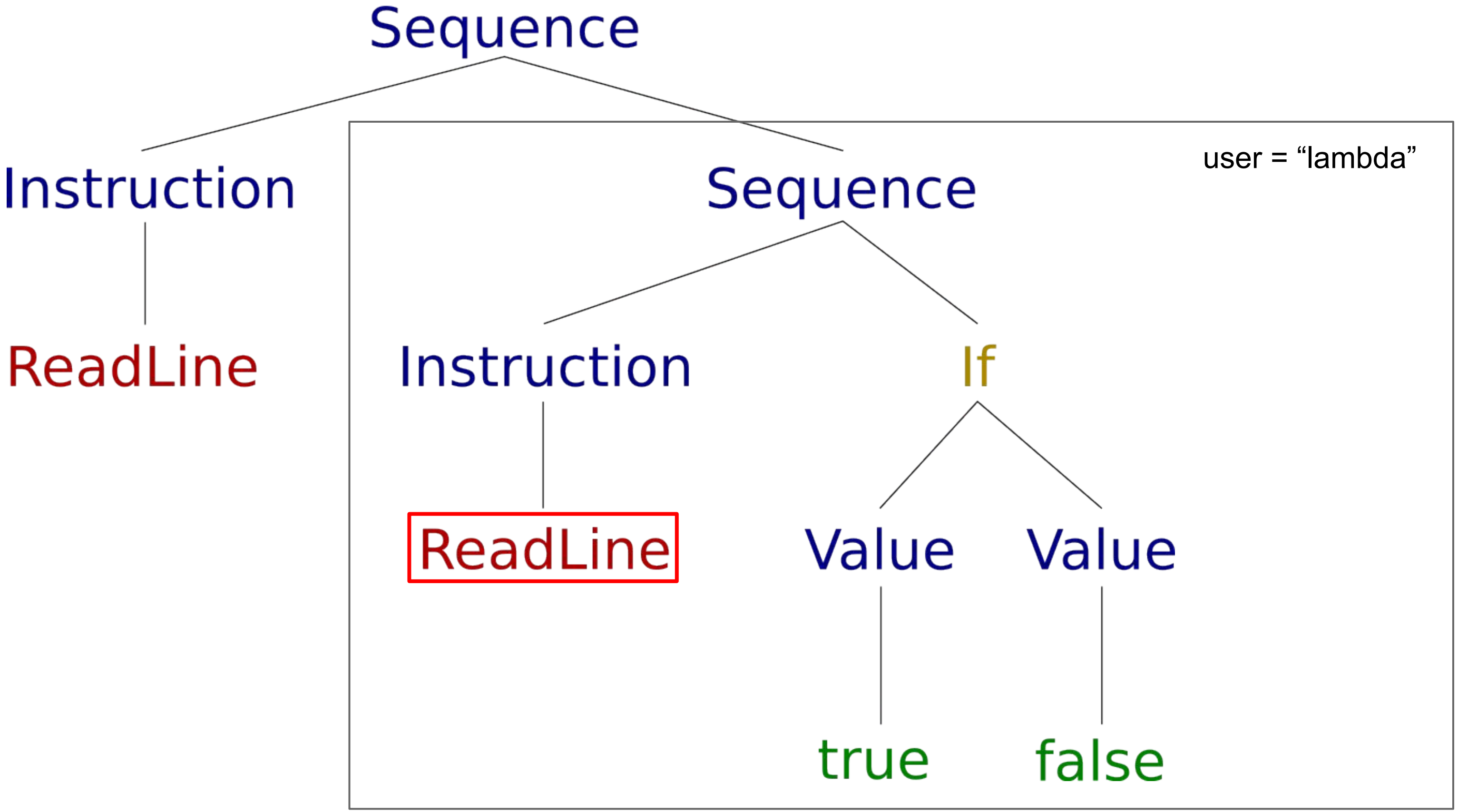
Interpreting a Program



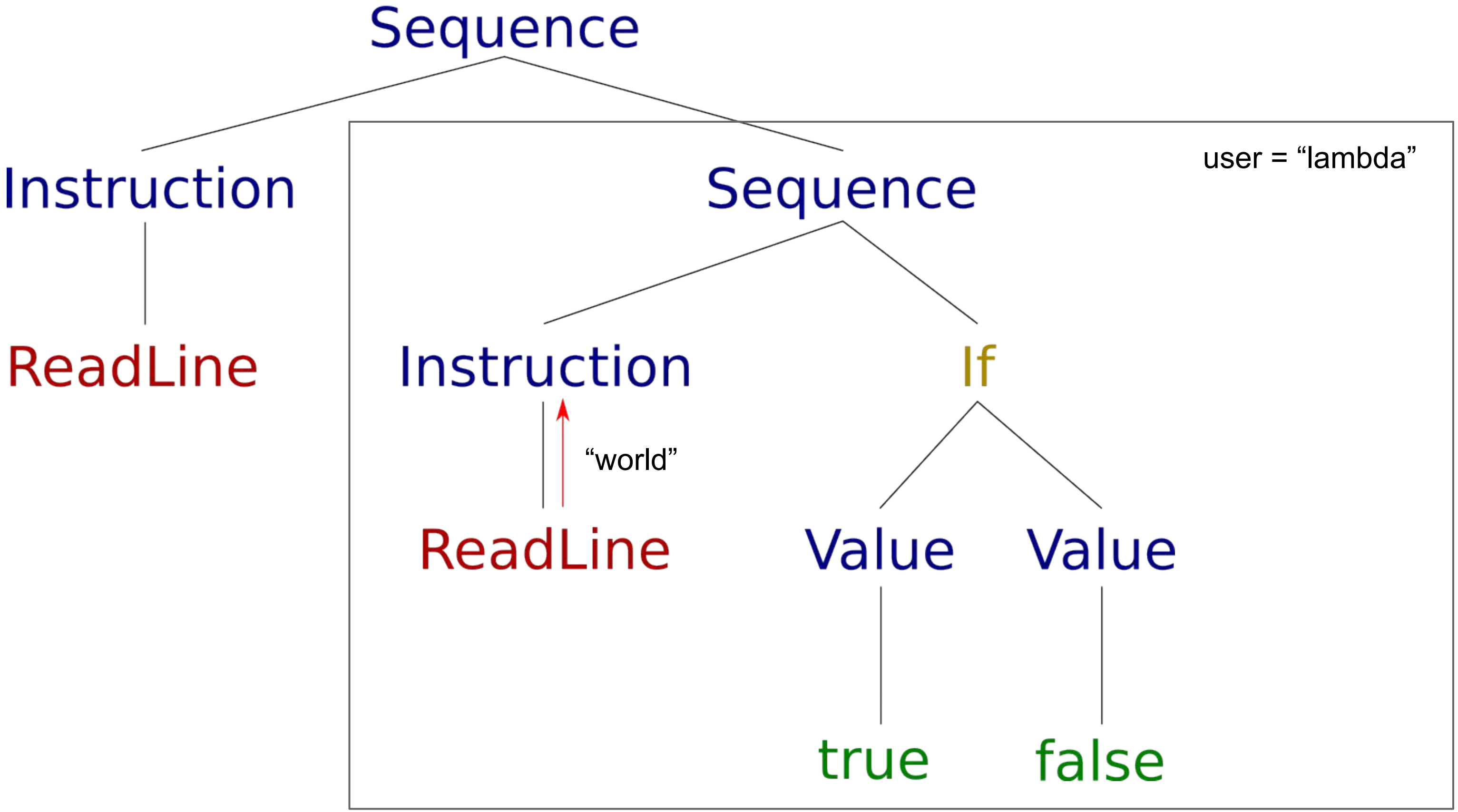
Interpreting a Program



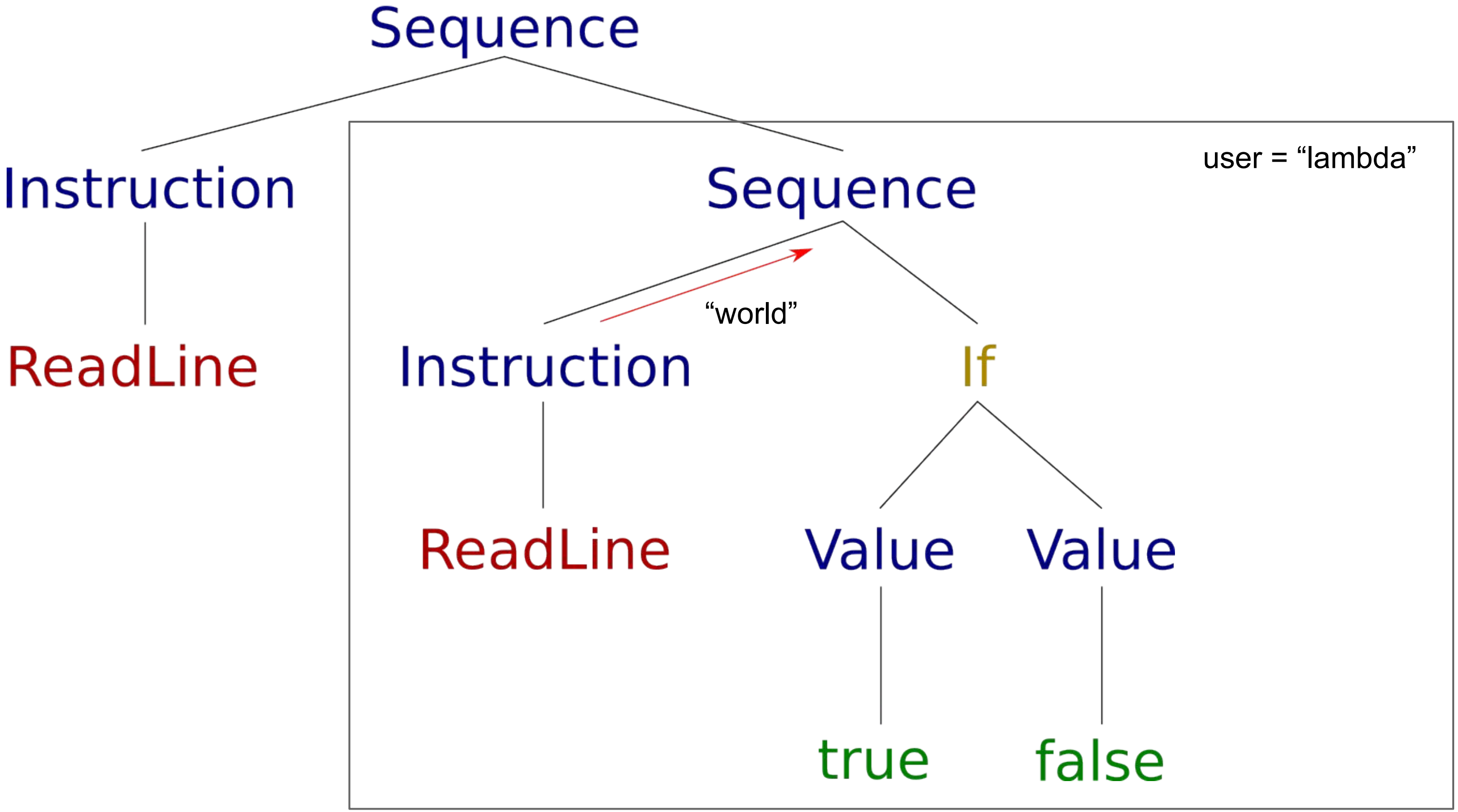
Interpreting a Program



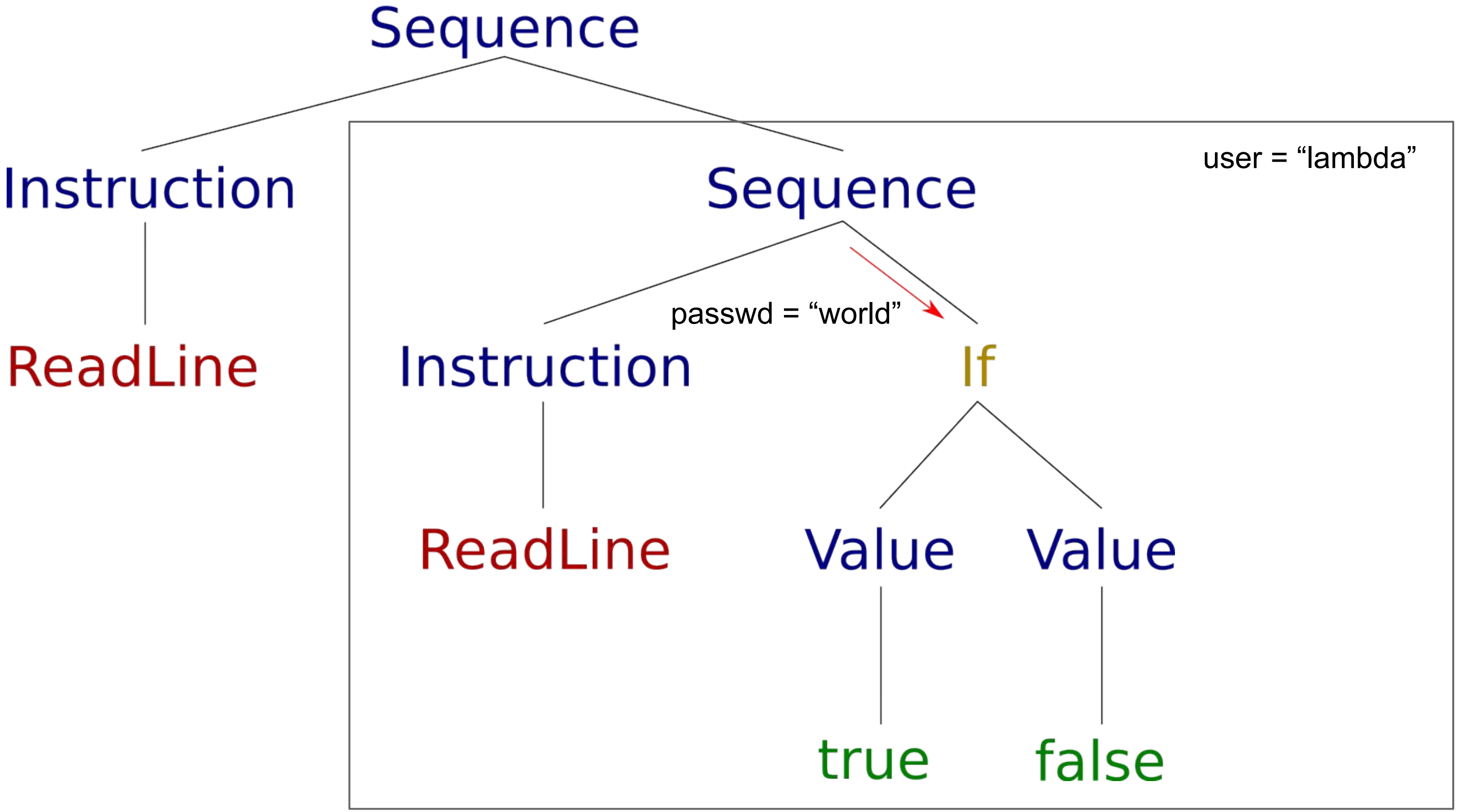
Interpreting a Program



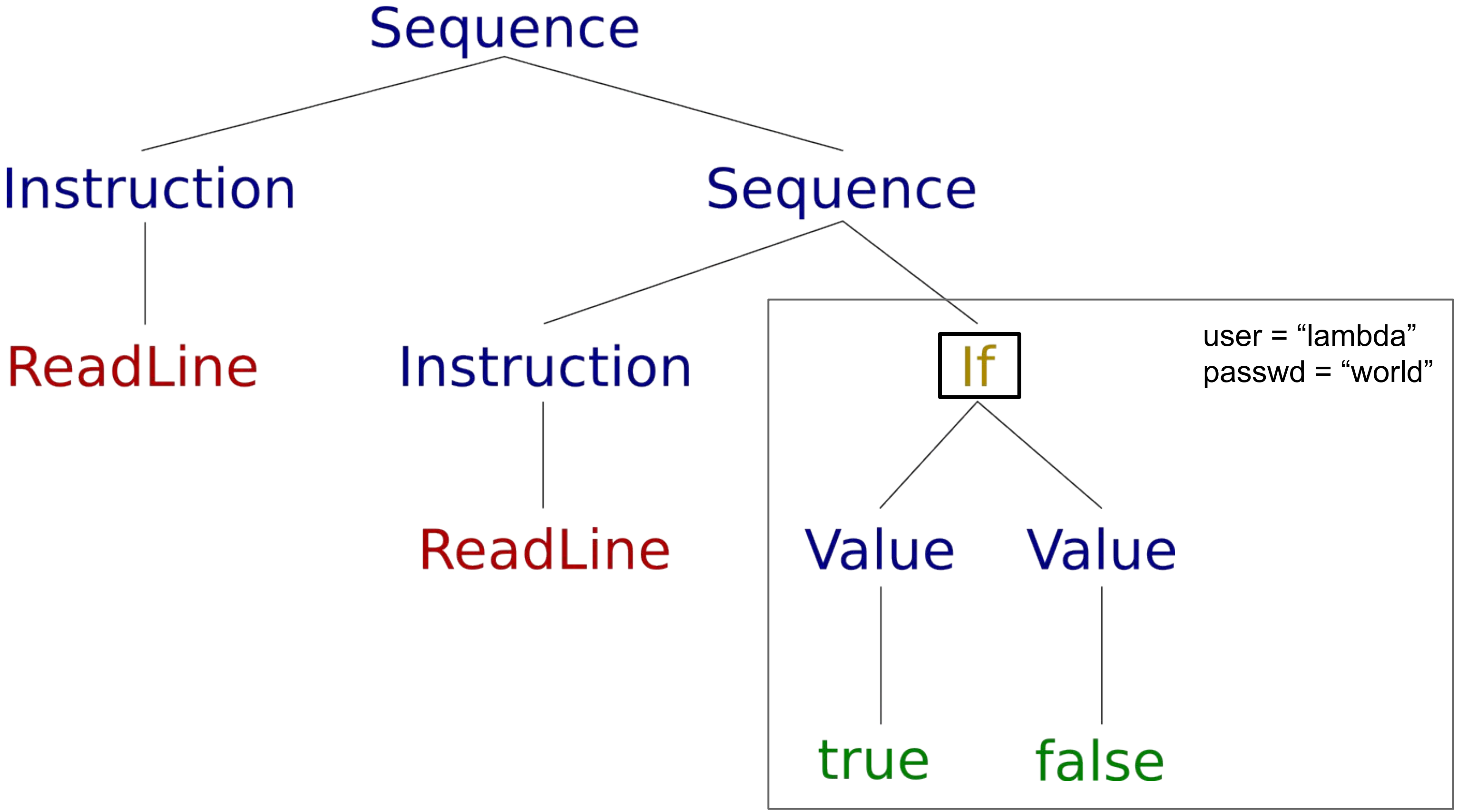
Interpreting a Program



Interpreting a Program

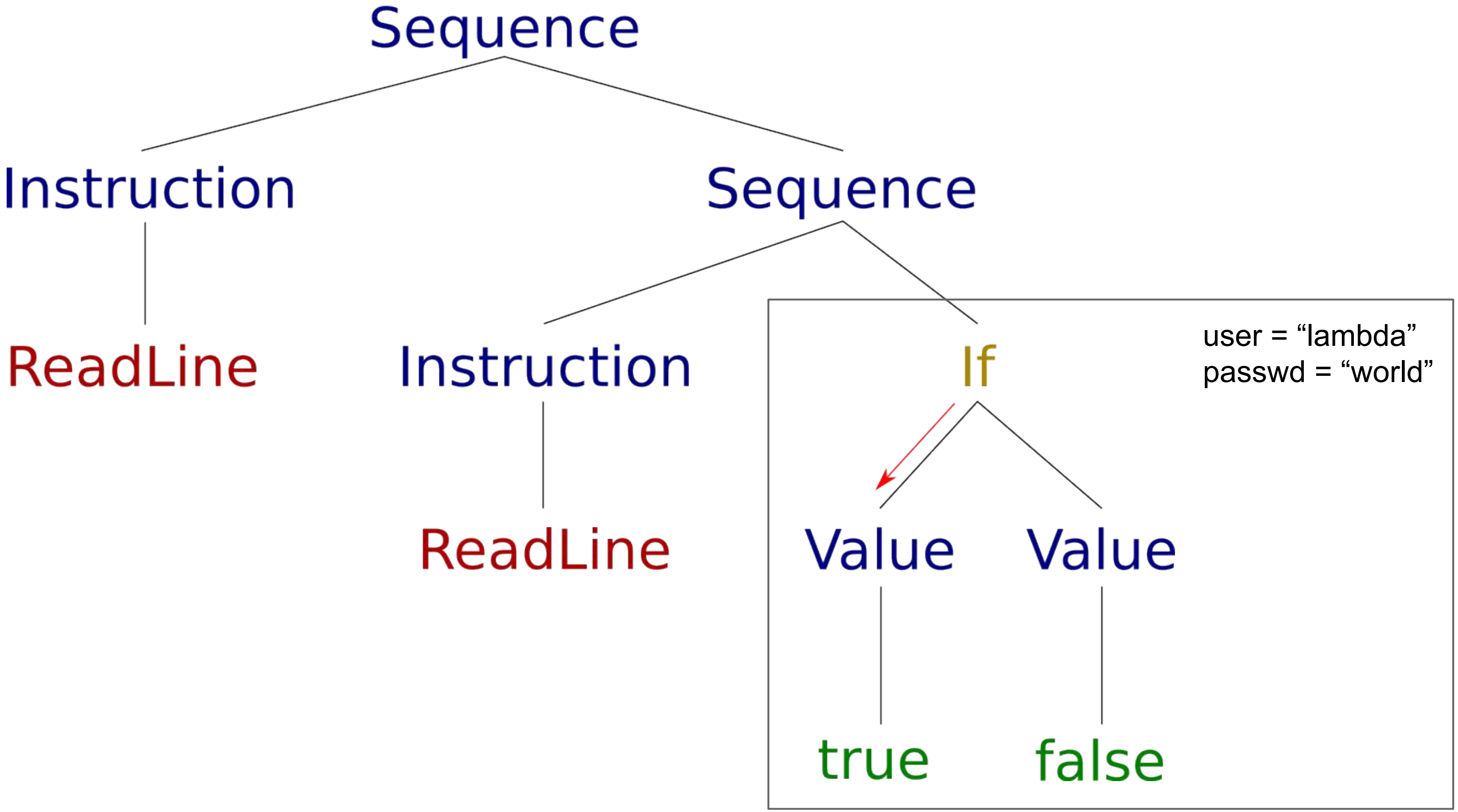


Interpreting a Program

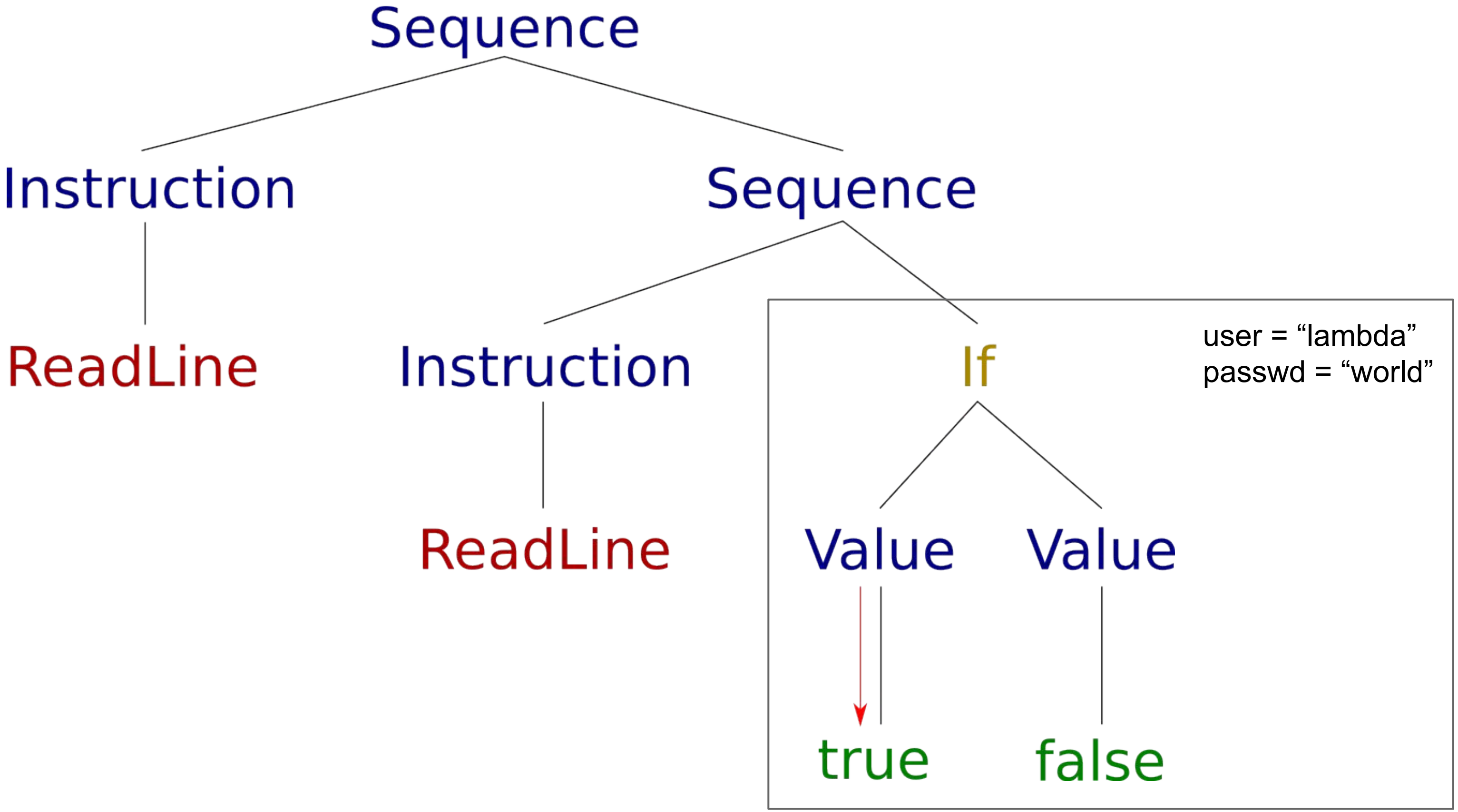


user = “lambda”
passwd = “world”

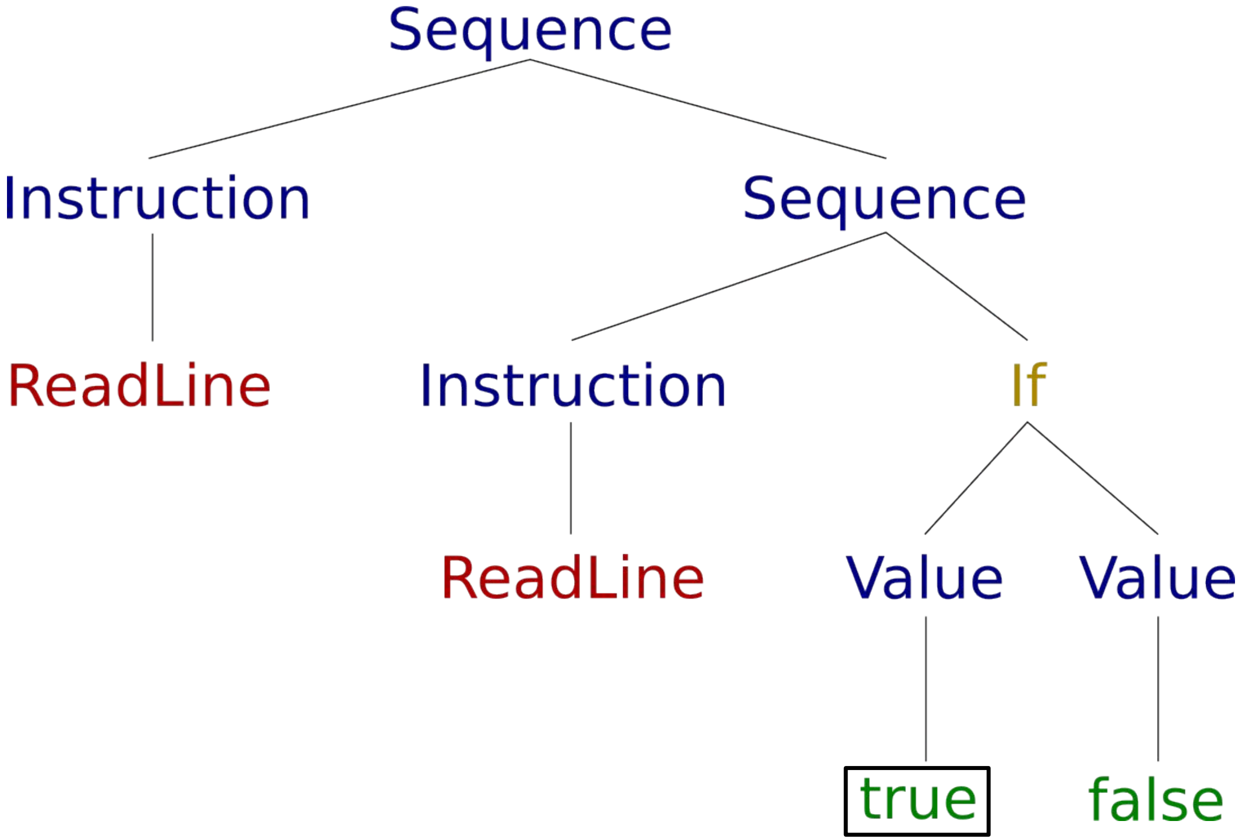
Interpreting a Program



Interpreting a Program



Interpreting a Program



Have We Finished Yet?

Nope.

Sugaring Process Walkthrough

- **Step 6: Smart constructors**
- Step 7: Infix notation
- Step 8: for-comprehension syntax

Sugaring Process Walkthrough

- Step 6: Smart constructors
- **Step 7: Infix notation**
- Step 8: for-comprehension syntax

Sugaring Process Walkthrough

- Step 6: Smart constructors
- Step 7: Infix notation
- **Step 8: for-comprehension syntax**

Bonus Track.
Bundle Up Your Library.

Final Step - Use Scalaz/Cats

- Our Program definition already exists out there
- It’s called Free Monad
- Scalaz/Cats implementation of Free is/has:
  - More efficient
  - Stack safe
  - Additional functionality

Conclusions

- We don’t lose expressiveness
- We don’t lose conciseness
- We gain flexibility
- Better decoupling
- Language + Interpreter > Interfaces

What's Next?

- Other free structures
- Combine your languages
  - Coproducts (Data types *“À la carte”*)
  - Monad transformers
- Kleislis
- ...

The End. Questions?