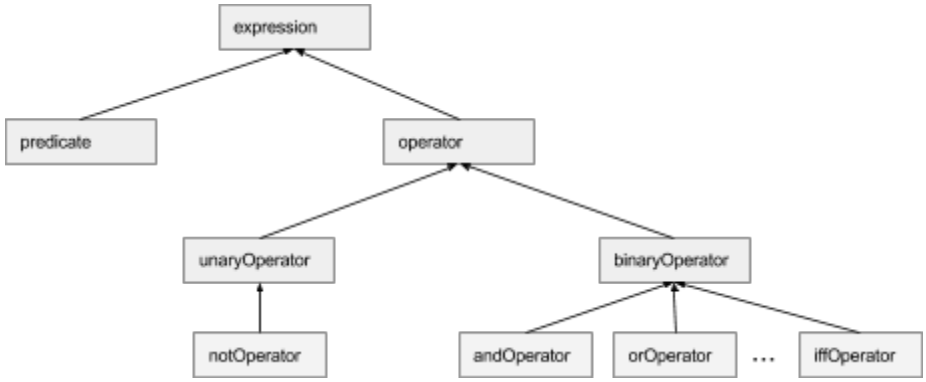
Team Name: *Grace*-ful Logic
Cameron White
David Opp
Sasha Fahrenkopf

Architecture

The logic module we created for the Grace language is built with a shallow hierarchy of logic expressions, predicates and operators. At the top level of the hierarchy are expressions: objects that respond to requests of logic operators and, or, not, implies, etc. and requests of logical queries like isContradiction and simplification. Predicates inherit from expression and represent the logical units (atoms) of an expression. Operators also inherit from expression and provide evaluation methods for the various logic operators. The hierarchy looks like this:



The equivalence module supplies semantics to the operators of the logic library. For example

```
def a = predicate("A")
def b = ~a
```

will request the notOperator extending expression which evaluates with a request to equivalence.not(). In this way we were able to treat our hierarchy as one similar to an abstract syntax tree in a compiler and produce semantics to the logical operators' syntax. Together with the logic hierarchy, this represents the entire architecture of our library. There are some details in why we chose to implement the objects as we did.

First, each level of the hierarchy carries information about an object's state and type, allowing for the top level expressions to have a robust set of expression evaluation methods. The use of predicate states is an important component of our design. Each predicate object has a public state of true or false. The predicate states allow for the library to cross product all the possible states for a given list of predicates. An expression's results method will make use of the states, iterating over each composite state and evaluating the expression with that set of true/false values. Similarly, the truthTable method uses states to print out each state combination and the resulting true/false value of the expression.

Second, you will find a copy method throughout the object methods. Copy is used throughout the the library because it was easier to deal with expressions as though they are immutable. For the scenario when the simplification needs to be at the top of the tree of operators and predicates, the expression needs

to be able to update itself. Copying allows us to ensure that the individual atoms are not affected when update operations are done.

Challenges

You will notice in the logic.grace module a number of utility and check functions for the current state and type of the object. These “isA” functions are used in almost every method of our code. This is one of our biggest infractions of object-oriented principles, as we are forcing the object to check the validity of it’s argument when it operates on it. This code is a result of not having verified type checking in the Grace language at our disposal. However, considering the situation in which the type checking was stable in Grace, using it would have still violated the principle of separating objects’ responsibilities. For the purpose of performing operations on expressions, the method must have the knowledge of what type of expression it is dealing with, be it an operator or a predicate. This challenge was both a challenge in not having language support but also in how to design a logic system that would operate *without* some kind of type checking in it’s evaluation methods.

Likewise, throughout the implementation we ran into different challenges constructing our abstractions in the ways we wanted because of an inability in Grace to support our desired form of inheritance. We wanted to nest factory methods inside our expression classes but were unable to because the scope of the keyword outer was unpredictable. Likewise, we wanted to implement the states, containedPredicates and results methods are definitions so they wouldn’t be recalculated on each reference but were unable to because the use of definitions broke the inheritance scope. This challenge was the biggest blocker because we spent a significant trying to debug what was broken with our model instead of making progress with the library itself.

Proud Of & Embarrassed About

Foremost, we are proud of creating a functioning logic library for the Grace language. We liked being able to contribute functionality that may come in handy for learning developers. We are also proud of how we created this library as a team. This project allowed us to further refine how we design code so it is reusable for even ourselves. Although in the end there are many things that we know need refactoring and could have been done in a more Object Oriented way, we are confident that we created a foundation for further development into this functionality.

If we would have had more time to devote, there are a few areas that we would immediately focus our intention because of their smelliness. First, our code lacks an overall modularity and simplicity. This is because of the constraints we had (in our and Grace’s ability) in dealing with inheritance and type checking. The result is one long file with lots of long methods, a source of smell because it hints that all of the objects know a lot about one another and work tightly together. Second, we would want this library to be able to parse logic strings instead of a user going through the motions of creating predicate and expression objects. The overall goal was to let the user create some arbitrary string and have the library create the structure for parsing, but we were unable to complete that goal in the time. Finally, our library lacks defined error checking. We rely on Grace’s standards, but offer no real support to the users for understanding where things failed (an error message indicating an illegal operator, for instance).