In this lab, we’ll gradually build up a Scheme program that can construct truth-tables automatically, test whether a given formula of the propositional calculus is satisfiable or whether it is a tautology, and check the validity of inferences.

Data Structures for Boolean Expressions

In the `/home/reseda/discrete-structures/code/discrete` directory on MathLAN, you will find four Scheme libraries that can be used to parse and evaluate Boolean expressions:

- (discrete logical-characters)
- (discrete propositional-scanner)
- (discrete propositional-parser)
- (discrete propositional-evaluator)

1. Launch DrRacket and open the `propositional-parser.ss` file. We’re not going to modify this file, but we’ll need to look it over briefly. The main purpose of this library is to define and export a procedure called `parse`, which starts with a string representation of a Boolean expression, such as "(((P ↔ Q) → ¬R) ↔ (¬(P ↔ Q) ∨ ¬R))", and returns an internal representation of that expression that includes useful information about its structure. This internal representation uses Scheme records.

   The body of the (discrete propositional-parser) library begins with seven record type definitions, each corresponding to one of the clauses in the recursive definition of ‘Boolean expression’ from the handout “The Propositional Calculus.” Read through the list of record type definitions for the various kinds of expressions.

   2. How many fields will each type of record have? What is likely to be stored in the fields of records of the types named after Boolean connectives?

The Structure of Expressions

3. Click the Run button to get an Interactions window. Invoke the `parse` procedure, giving it the argument "P"; `parse` will return a value of a record type. How are values of this type represented in the Interactions window?

   In experimenting with `parse`, you’ll probably need to write strings containing the exotic characters for the Boolean literals and connectives. Since it’s not obvious how to type them in from the keyboard, I’ve provided a library, (discrete logical-characters), that (a) contains an occurrence of each of these characters and (b) defines a name for each one, in case this turns out to be handy when constructing strings and formulas.

   4. Using invocations of `parse` and the relevant accessor methods, explore the result of parsing "(((P ↔ Q) → ¬R) ↔ (¬(P ↔ Q) ∨ ¬R))". What is the structure of the record that `parse` returns?

Structural Recursion

The propositional-parser library also includes a procedure called `expression->string` that acts as an “unparser”—it takes a record of any of these expression types and converts it to string format.
Scroll down to the definition of the `expression->string` procedure, which takes a record of any of the seven types just introduced and produces a string representation of it. Notice the locations of the recursive calls to the `expression->string` procedure in the body of its own definition.

5. What is the purpose of those recursive calls? How do they reflect the structure of the records that represent complicated Boolean expressions?

6. In the Interactions window for the `propositional-parser` library, use the constructor procedures `make-variable-expression` and `make-negation-expression` to build your own record for the negation of the variable ‘R’. Apply the `expression->string` procedure to the value you construct to make sure that it “unparses” correctly.

Running the Evaluator

7. Return to the window that DrRacket set up initially, before you opened the `propositional-parser.ss` file. Import the `(scheme base)`, `(discrete propositional-parser)`, and `(discrete propositional-evaluator)` libraries. Click Run to make the Interactions window usable.

8. Since you’ve imported the `propositional-parser` library, the `expression->string` and `parse` procedures, and all of the constructors, classification predicates, and field accessors from that library are now available in the Interactions. Parse some string to confirm that you have access to them.

The `evaluate` procedure in the `propositional-evaluator` library determines the value of a given Boolean expression—given not as a string, but as a record of one of the expression types, which is why you’ll need `parse` as well. However, the `evaluate` procedure cannot perform its calculation without knowing the values of the variables that occur inside the Boolean expression. In other words, it needs an assignment as a second argument.

We represent an assignment as an association list. Each pair in such a list consists of a string containing a single upper-case letter (as the car of the pair) and a Boolean value (as the cdr). So, for instance, `((("P" . #t) ("Q" . #t) ("R" . #f)))` is an assignment that gives ‘P’ and ‘Q’ the true Boolean value and ‘R’ the false one.

9. In the Interactions window, invoke `parse` and `evaluate` to determine the value of ‘((¬P ↔ ¬Q) ↔ ¬R)’ under the assignment just given. Build a truth table for this expression and determine whether the result is consistent with the corresponding entry in the truth table. (Which row of the truth-table corresponds to this assignment?)

Evaluation by Structural Recursion

Open the file containing the `(discrete propositional-evaluator)` library and scroll down to the definition of the `evaluate` procedure.

10. In what ways is the structure of that procedure similar to the structure of the unparsing procedure `expression->string` that we examined earlier?

11. The `cond`-clauses that deal with expressions containing connectives also reflect the semantics of those connectives. The clauses for case of negation, conjunction, disjunction, and equivalence use the Scheme procedures and control structures `not`, `and`, `or`, and `eq?` in an obvious way. How is the truth-table for the connective ‘→’ reflected in the `cond`-clause for implications?

Generating Possible Assignments

Return at this point to the original DrRacket window. Add the `(discrete sets)` library to your import list.

12. Define a Scheme procedure `all-assignments` that takes a set of variables as its argument and returns the set of all possible assignments to those variables. This is an exercise in list recursion.
(the first step, as in many procedures that operate on sets, is to convert the argument from a set to a list), and it’s pretty straightforward if you break it down into two subproblems:

(a) In the base case, the list of variables is empty, and we want to return a set containing only an empty assignment. How can we construct an empty assignment? How can we construct a set containing only that empty assignment?

(b) In the recursive case, we can issue a recursive call to the very procedure that we’re defining, giving it the cdr of the current list and getting back the set of all possible assignments to the variables in the cdr. Once we have that set, how can we transform it into the set of all possible assignments to all of the variables in the current list, including the car of the current list? How can we code that transformation in Scheme?

13. Using the all-assignments procedure, determine the value of ‘((¬P ↔ ¬Q) ↔ ¬R)’ under every possible assignment of Boolean values to ‘P’, ‘Q’, and ‘R’. Reconcile the results, if necessary, with the truth-table you constructed above.

Satisfiability

Recall from the reading on truth-tables that a Boolean expression is satisfiable if there is some assignment that satisfies it, that is, some assignment under which it is true. We’d like to automate the process of testing the satisfiability of a given formula, and at this point we have most of the necessary pieces: The parse procedure gives us the Boolean expression as a record structure, evaluate can check whether such an expression is true under a given assignment, and all-assignments can provide us with a list of the assignments that evaluate will need to go through.

The only remaining problem is to get the set of variables that we need to supply as the argument to all-assignments.

14. Define and test a procedure called occurring-variables that takes a Boolean expression (the syntax tree representation, not the string representation) as its argument and returns a set containing the single-character names of the variables occurring in that expression. (Hint: Use structural recursion. The definition of occurring-variables should resemble expression->string and evaluate in form.)

15. Define and test the satisfiable? predicate, which takes a string representing a Boolean expression and determines whether that expression is satisfiable. Use your procedure to determine whether ‘((¬P ↔ ¬Q) ↔ ¬R)’ is satisfiable.

Tautology and Inference Validity

As explained in the reading, a Boolean expression is a tautology if, and only if, it is true under every possible assignment of Boolean values to the variables that occur in it.

16. Define and test a predicate tautology? that takes a string representing a Boolean expression and determines whether that expression is a tautology. For example, determine whether ‘((¬P ↔ ¬Q) ↔ ¬R)’ is a tautology.

17. The inference from a set Γ of premisses to a conclusion φ is valid if, and only if, every assignment that satisfies all of the premisses also satisfies the conclusion. (This is the relationship expressed in the reading as “Γ |= φ.”) Define and test a predicate valid-inference? that takes two arguments, a list of premisses (strings representing Boolean expressions) and a conclusion (a string representing a Boolean expression) and determines whether the inference from the premisses to the conclusion is valid, in this sense.