Lab: Evaluating Boolean Expressions
CSC/MAT 208: Discrete Structures
Department of Computer Science
Department of Mathematics and Statistics
Grinnell College

Pairs

Here are the programmer pairs for today’s lab. Drivers are on the left.

- will take the place of anyone who is absent.

Exercises

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.

Constructing a truth-table

**Exercise 0:** Using pencil and paper, construct a truth-table for the Boolean expression ‘((¬p ≡ ¬q) ≡ ¬r)’. Note that you’ll need eight rows (one for every combination of values for the variables ‘p’, ‘q’, and ‘r’) and eight columns (three for the variables, three for their negations, one for ‘(¬p ≡ ¬q)’, and one for the entire expression).

**Exercise 1:** State concisely the conditions under which the Boolean expression in exercise 0 is true.

Data Structures for Boolean Expressions

In the /home/stone/discrete/code/discrete directory on MathLAN, you will find four Scheme libraries that can be used to parse and evaluate Boolean expressions:
• logical-characters.sls,
• propositional-scanner.sls,
• propositional-parser.sls,
• propositional-evaluator.sls.

Copy these libraries into the discrete subdirectory of your collections directory.

Now launch DrRacket and open the propositional-parser.sls 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.

Look now at the propositional-parser.sls file. After the export and import lists, you’ll notice that the body of the library begins with the line

(define-record-type literal-expression (fields polarity))

and continues with six more definitions of the same sort, 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.

Exercise 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

Exercise 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 does the waiter in the Interaction window represent values of this type?

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.

**Exercise 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 an “unparser” — a procedure called `expression->string` that 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 eight 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.

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

The `expression->string` procedure will work on any correctly structured record of the appropriate type.

**Exercise 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’. (Note that the argument to the constructor procedure `make-variable-expression` should be a single character, not a string.) Apply the `expression->string` procedure to the value you construct to make sure that it “unparses” correctly.

### Running the Evaluator

Now return to the window that DrRacket set up initially, before you opened the `propositional-parser.sls` file. Import the `(rnrs base)`, `(discrete propositional-parser)`, and `(discrete propositional-evaluator)` libraries. Click Run to make the Interactions window usable.

Since you’ve imported the `propositional-parser` library, the `parse` and `expression->string` procedures and all of the constructors, classifi-
cation 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 [new-term assignment] as a second argument.

We represent an assignment as an association list. Each pair in such a list consists of a character (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.

Exercise 7: In the Interactions window, invoke parse and evaluate to determine the value of ‘((¬p ≡ ¬q) ≡ ¬r)’ under the assignment just given. Check whether the result is consistent with the corresponding entry in the truth-table you constructed at the beginning of today’s lab. (Which row of your 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.

Exercise 8: In what ways is the structure of that procedure similar to the structure of the expression->string procedure we examined earlier?

Exercise 9: 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?

Exercise 10: Some presentations of the propositional calculus provide an additional connective ‘¬∧’, pronounced “nand,” such that \(\phi \bar{\wedge} \psi\) is false if, and only if, both \(\phi\) and \(\psi\) are true. If we had such a connective, what would the cond-clause for it in the definition of the evaluate procedure be?
Generating Possible Assignments

Return at this point to the original DrRacket window.

The last exercise in the “Truth tables” reading invited readers to define a Scheme procedure that takes a class of variables (characters) as its argument and returns the class of all possible assignments to those variables. This is an exercise in list recursion (the first step, as in many procedures that operate on classes, is to convert the argument from a class to a list), and it’s pretty straightforward if you break it down into two subproblems:

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

2. 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 class of all possible assignments to the variables in the cdr. Once we have that class, how can we transform it into the class 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?

Exercise 11: Define and test this all-assignments procedure.

Exercise 12: 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 in exercise 0 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 thorough.

The only remaining problem is to get the class of variables that we need to supply as the argument to all-assignments.
Exercise 13: Define and test a procedure called `occurring-variables` that takes a Boolean expression (the record, not the string) as its argument and returns a class 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.)

Exercise 14: 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.

Exercise 15: 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.

Exercise 16: The inference from a class 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.

Exercise 17: Collect the definitions you created in exercises 11–16 together with any helper procedures on which they depend, and structure them as a new library, called `(discrete propositional-semantics)`. Add an export list above the import list, exporting the names of the procedures that might conceivably be useful in an application program, but not the names of the helper procedures that are useful only inside the library. Save the contents of the Definitions window, as text, creating a new file `propositional-semantics.sls` and placing it in the `discrete` subdirectory of your collections directory, alongside `propositional-scanner.sls`, `propositional-parser.sls`, and `propositional-evaluator.sls`. 
Open-ended Optional Exercise: Tweaking the Propositional Calculus

We haven’t needed to look at the (discrete propositional-scanner) library. It’s present because the (discrete propositional-parser) library imports it and depends on it. Its purpose is to break a string into individual characters, to discard any spaces, tabs, and newlines that it encounters within the strings, and to group the characters if necessary into significant tokens, which it passes along to the parser. Specifically, the scanner procedure converts a given string into a Scheme procedure that disgorges the next token each time the parser asks for one.

Read through the propositional-scanner.sls file and the definition of the parse-expression procedure in propositional-parser.sls to get an overall idea how the parser uses the token-source procedure that scanner provides.

Revise the libraries to add the connectives ‘\(^\wedge\)' (‘nor’), and ‘\(^\not\equiv\)' (‘exclusive or’) to our implementation of the propositional calculus. \(\neg (\phi \lor \psi)\) is true if, and only if, both \(\phi\) and \(\psi\) are false; \(\neg (\phi \not\equiv \psi)\) is true if, and only if, \(\phi\) and \(\psi\) have different Boolean values.

If you work carefully, the only change that you will need to make in the (discrete propositional-semantics) library is the addition of two more cond-clauses in the definition of the occurring-variables procedure.

Copyright © 2013, 2014, 2016  John David Stone

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit

http://creativecommons.org/licenses/by-sa/4.0/deed.en_US

or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.