# 1 CSE 341 — Autumn 2005 — Assignment 6

(version of Nov 22)

Due Dec 1, 10pm.

Write and test a set of Scheme functions to perform simplification of symbolic expressions involving sets of integers. These Scheme functions must all be written in a pure functional style (no side effects).

A set constant should be represented as a list consisting of the atom `set` followed by the integers in the set. The order of the integers in the list isn't significant, but there shouldn't be any duplicates. The empty set is denoted by `(set)`.

A set expression is either:

- a set constant
- the symbol `U`, which denotes the universe (in this case the set of all integers)
- some other symbol, which denotes a set-valued variable
- an expression consisting of `union` followed by two set expressions
- an expression consisting of `intersect` followed by two set expressions

First, write and test a predicate `good-set-expr?` that tests whether its argument is a properly represented set expression, i.e. it is a list consisting of `(set)` followed by 0 or more integers, with no duplicates; a symbol, an expression consisting of `union` followed by two set expressions, etc. For example:

```
(good-set-expr? '(set 1 5 2))  => #t
(good-set-expr? '(set))  => #t
(good-set-expr? '(set 1 5 5))  => #f
(good-set-expr? '(set 1 (5) 2))  => #f
(good-set-expr? 'U)  => #t
(good-set-expr? '(union (union x y) (intersect (set 3 4 5) z))  => #t
```

You can run `good-set-expr?` on sample input, and all your other functions can assume that set expressions are properly represented.

Then write and test functions to perform set union and intersection on set constants. For example

```
(union '(1 5 2) '(5 10))  => (1 2 5 10)
(intersect '(1 5 2) '(5 10))  => (5)
(intersect '(1 5 2) '())  => ()
```

Or using the `set` macro (described a bit later), these expressions could be better written:

```
(union (set 1 5 2) (set 5 10))  => (1 2 5 10)
(intersect (set 1 5 2) (set 5 10))  => (5)
(intersect (set 1 5 2) (set))  => ()
```

Next, write and test a function `set-simplify` that does simplification of symbolic expressions involving sets. Your function should perform the following simplifications. In the rules below, $x$, $y$, and $z$ are arbitrary expressions

involving the set operations $\cup$ and $\cap$ (set union and intersection respectively). These rules are given in standard set notation, but your Scheme functions should manipulate expressions using Scheme's expression syntax.

$$\begin{aligned} x \cup \emptyset &\Rightarrow x \\ x \cap \emptyset &\Rightarrow \emptyset \\ x \cup x &\Rightarrow x \\ x \cap x &\Rightarrow x \\ x \cup U &\Rightarrow U \\ x \cap U &\Rightarrow x \end{aligned}$$

Also include the commutative version of the rules (e.g. a rule for $\emptyset \cup x$).

Finally, if you have an expression involving set constants, simplify the expression by evaluating the expression. (Use the union and intersect functions you defined earlier.) For example:

$$\{1, 5, 2\} \cup \{5, 10\} \Rightarrow \{1, 2, 5, 10\}$$

Here are some example calls to `set-simplify`:

```
(set-simplify '(union x x))  =>  x
(set-simplify '(intersect x (set)))  =>  (set)
(set-simplify '(union (intersect U x) (union (set) x))) => x
(set-simplify '(union (set 1 5 2) (set 5 10)))  => (set 1 5 2 10)
(set-simplify '(union (set 1 5 2) x))  => (union (set 1 5 2) x)
(set-simplify '(set 1 5 2))  => (set 1 5 2)
(set-simplify '(intersect (intersect x y)
                          (intersect (set 1 3) (set 5 8 10))))  => (set)
```

Note that `x` in the examples is a symbolic variable, while `(set 1 5 2)` represents a constant, namely the set $\{1, 5, 2\}$. U is the set of all integers. Notice also that in the final two cases, no simplification was possible.

In standard mathematical notation, the above calls are equivalent to:

$$\begin{aligned} x \cup x &\Rightarrow x \\ x \cap \emptyset &\Rightarrow \emptyset \\ (U \cap x) \cup (\emptyset \cup x) &\Rightarrow x \\ \{1, 5, 2\} \cup \{5, 10\} &\Rightarrow \{1, 5, 2, 10\} \\ \{1, 5, 2\} \cup x &\Rightarrow \{1, 5, 2\} \cup x \\ \{1, 5, 2\} &\Rightarrow \{1, 5, 2\} \\ x \cap y \cap \{1, 3\} \cap \{5, 8, 10\} &\Rightarrow \emptyset \end{aligned}$$

## 2 Evaluating Set Expressions

You should also be able to give values to all variables used in a set expression, and then evaluate the result using `eval`. You should also be able to use `eval` on the result returned by `set-simplify` — it should be the same.

To make this work, you should define `set` as a macro that expands (for example) `(set 3 4 5)` into `'(3 4 5)`, and `(set)` into `'()`.

For example:

```
(define x (set 10 15 20))
;; with the macro for set, this is the same as writing
;; (define x '(10 15 20))
;; but for consistency we'll always write set constants using the macro

(set-simplify '(union x x))  =>  x
;; the result of evaluating the expression and the simplified expression
;; are the same
(eval '(union x x))  =>  (10 15 20)
(eval (set-simplify '(union x x)))  =>  (10 15 20)

(set-simplify '(intersect x (set)))  =>  (set)
(eval '(intersect x (set)))  =>  ()
(eval (set-simplify '(intersect x (set))))  =>  ()
```

Make sure you understand why this works. Something that may be confusing are the multiple layers of quoting and evaluating. First, here is an easy case.

```
'(union x x)
```

evaluates to a list with 3 elements, the symbols `union`, `x`, and `x`. If we evaluate this in turn, we get `(10 15 20)`, which represents the set $\{10, 15, 20\}$. For the second set of examples,

```
(set-simplify '(intersect x (set)))
```

evaluates to the list `(set)`, in other words, a list with a single element, the atom `set`. If we evaluate `(set)`, we get `()`.

## 3 Testing Your Functions

You should systematically test your top-level functions (`good-set-expr?`, `set-simplify`, `union`, and `intersect`, and show the output from the tests. For example, for `set-simplify`, run it on the test cases shown above, and any others you think are necessary. (For example, you should simplify the empty set also.) Similarly, test `union` and `intersect` with empty and non-empty sets, and also U. In addition, include some tests using `eval`, as above.

For lists representing set constants, it doesn't matter in what order the elements occur.

It's convenient to write a `test-sets` function that will run tests on all your functions, so that it's easy to perform the tests repeatedly. The `test-sets` function should print out some informative messages about each test, and then return the final result (`#t` if all the functions passed all of the tests, and `#f` otherwise).

# 4 Extra Credit

1. Add a set difference operator. (Remember this has to handle U as well as the other sorts of sets.)

2. Add additional simplification rules. The above rules are exhaustive if you just look at expressions involving two variables, or a variable and a constant, or two constants, but there are other simplifications if you consider more complex expressions. For example:

$$\begin{aligned} (x \cup y) \cap x &\Rightarrow x \\ w \cup x \cup y \cup z \cup w &\Rightarrow w \cup x \cup y \cup z \end{aligned}$$

For full extra credit, rather than just adding a grab-bag of additional simplification rules, devise a comprehensive approach to the issue — what are the classes of simplification rules you are considering? How did you pick the ones you included? You might want to look at a textbook on set theory or finite mathematics for ideas.