Spring 1997 **Programming Project #4: Sets** CMSC 420
Due: 4/29 David Traum

1 Introduction

Sets are a generalization of lists, in which only the elements themselves are important, and not the sequence in which they appear. The basic operation is a test to see whether an item is a **member** of a set. An important derived property is that, unlike a list, there is no notion of an item appearing multiple times. For instance, as lists, `(1 3)`, `(3 1)`, and `(3 1 3)` are all different, but as sets, they are all the same set {`1,3`}. Other interesting properties of sets are the **cardinality**, which is the number of (distinct) items in the set, and a number of comparisons with other sets, such as **equality**, which holds if two sets have the same elements, **subset**, if all the members of one set are in another. In addition there are also constructor functions which build up sets from other sets. These include **intersection**, **union**, and **set-difference** of two sets. Sometimes it is also useful to specify the range of possible set values. In this case, we have a special set **U**, the Universe, which contains all possible elements. Now we can also define the **complement** of a set, which is all members of the universe *not* in that set.

Sometimes, sets themselves can be elements of other sets. This can lead to problems in what is called *naive-set theory*, particularly: can a set be a member of itself? In order to avoid this question, some set-theories only allow as elements sets with a lower *level* than themselves. An atom (or non-set element) is of level zero. A set containing only atoms is of level 1. A set containing only atoms and sets of level 1 is of level 2, etc. Thus no set is allowed to contain itself (or other sets of the same or higher level than itself). Since it can be an expensive procedure to determine the level of a set (e.g., comparing all the levels of its elements), it may be useful to keep track of this property as an explicit part of the representation of the set. We can also talk about the **powerset** of a set. This is a set of all subsets of the given set. Note that this set will be one level higher than the original set, since it contains as elements sets of the elements of the original set. All level 1 sets are subsets of the Universe **U**, moreover, all level 2 sets are subsets of $\mathbf{U^1}$, the powerset of U, unioned with U itself, and all level 3 sets are subsets of $\mathbf{U^2}$, the powerset $\mathbf{U^1}$, unioned with $\mathbf{U^1}$.

For representing sets, it is often useful to *order* (i.e., sort) them. This cuts down in the search time needed for a number of operations, including testing for membership (i.e., searching for an item), as well as computing cardinality (since one must not count duplicates), and set construction functions. To order a set, one must have some principle of comparison to see if an item comes before another item. One possibility, if an enumeration of the universe is given, is to use the order that the items are present in that enumeration. E.g., if the universe is {`e, 3, we`}, then `e` comes before `3` which in turn comes before `we`. If no such enumeration is given, one could revert to a default ordering, such as lexicographic order. One must also determine a policy for how elements which are of different levels are to be compared.

2 Assignment

Your assignment is to implement a package in lisp for multi-level (finite) sets. You must write the following functions, as well as any support functions necessary. You may use any lisp functions discussed in class or in your handouts.

Part 1: Discussion

You must think about how best to implement the set package: what basic data structures would be most convenient, given the operations that need to be performed. How do these data structures support efficient implementations of the algorithms you choose to implement your functions. Give a description of the set data structure you choose, and why it is good and bad for representing sets. Discuss why it is better than other structures you could choose, and what drawbacks it has in relation to those others. Justify your particular choice. Also, describe the algorithms you will use to construct and manipulate your data structure, and explain why you chose these over other possibilities. Write no more than two (12point, typewritten) pages.

Part 2: Simple Sets

Write the following lisp functions to implement a set package:

Constructors

1. `set-universe` *list*
   set the universe to be the items listed in *list*. If given the empty list, then the universe is not considered finite, and any atom can be a valid element. Make sure you know whether the universe is finite for other operations, including making sure that a set is valid (contains only elements in the universe).

2. `make-set` *list*
   construct a set from a list of items. You will probably want to sort the items, and remove duplicates, as well as any other operations to construct your set data structure.

3. `set-intersection` *set1 set2*
   returns a new set which has as elements only objects that are elements of both *set1* and *set2*.

4. `set-union` *set1 set2*
   returns a new set which has as elements all objects that are elements of either *set1* or *set2* (or both).

5. `set-diff` *set1 set2*
   returns a new set which has as elements objects that are elements of *set1* but not *set2*.

6. `complement` *set*
   Returns a new set including all elements of the universe which are not elements of *set*. I.e., (`set-difference U` *set*).

7. `add-element` *object set*
   returns a new set which has *object* as well as all the elements of *set* as members.

8. `rem-element` *object set*
   returns a new set which has all the elements of *set* except *object* as members.

Accessors

9. `cardinality` *set*
   Returns the number of elements of the set.

10. `elements` *set*
    returns a list of all the elements of *set*.

11. `Universe`
    Returns a list of all the elements in the universe (returns `*error*` if the universe is not finite).

Predicates

12. `nullset` *object*
    true if *object* is the null set. The null set contains no elements. E.g., it is the complement of the set containing all objects in the Universe.

13. `finiteU`
    true if the universe of possible elements has been specified.

14. `Setp` *object*
    true if *object* is a valid set (according to your definition of the data structure).

15. `individualp` *object*
    true if *object* is a member of the universe (and not a set).

16. `setequal` *set1 set2*
    true if *set1* and *set2* are equivalent sets (i.e., have the same elements).

17. `subset` *set1 set2*
    true if *set1* is a subset of *set2*

18. `setmember` *object set*
    True if *object* is an element of *set*.

Part 3: Higher-Level Sets

Augment your basic set definitions to also allow sets to be members of other sets. Remember that this is legal only if the member is of a lower level than the containing set. Make sure all of the functions in the previous part now work with higher-level sets. Also write the following additional functions:

Constructors

19. `add-set` *set*
    Creates and returns a new set including all the elements of *set*, as well as *set*, itself. The returned set is thus one level higher than *set*.

20. `power-set` *set*
    Return a new set which includes as elements, all the subsets of *set* (including *set*, itself. The returned set is thus one level higher than *set*.

21. `flatten-set` *set*
    Returns a level-one set with all the (level-zero) items that are set-submembers of *set* (see below).

Accessors

22. `level` *object*
    Returns the level of an object (recall that an atom is a level-zero object). If the object is not part of the universe and is not a valid set, then return `*error*`.

Predicates

23. `higher-level` *set1* *set2*
    true if *set1* is of a higher level than *set2*

24. `set-submember` *object* *set*
    true if *object* is a member of *set* or (recursively) if *object* is a set-submember of any member of *set*.

3 Implementation Notes

Although you are implementing sets which do not have an intrinsic ordering to their elements, you will probably want to represent these internally using some canonical ordered representation, so that searching, insertion, deletion, and merging are faster. Thus, for creating sets you will probably want to have a sorting function. Make sure you choose a good one and justify your selection in Part I. Also, you will probably want to have easy ways to access the n^{th} element, as if it were a list, similar to the `nth-item` function. If you assume this function works for sets, then `powerset` *set* could be defined recursively as follows, where n stands for `(cardinality` *set*`)` :
`(powerset` *set*`)` $=$ *set* $\cup\, [\bigcup_{1\leq i\leq n)}$ `(powerset (rem-element (nth-item` *i set*`)` *set*`))`].

In order to sort the items, you will need a comparison function to order the elements. While you could use the order provided as input to `set-universe`, here is a handy lisp function that might help:

```
(defun elt-comp (element1 element2)
    (string< (princ-to-string element1) (princ-to-string element2)))
```

This function will work, as long as distinct elements print differently.

4 Grades

This project will count for 45 points total, towards your programming project grade. Each part will count for 15 points (part 3 includes making the functions in Part 2 work for sets of different levels). You also must make your functions conform to the design you describe in Part 1.

Sample test data will be provided shortly via the web page.

Use `submit 4 <filename>` to hand in your program. If you choose to use more than one lisp file, make sure to tar all your files together and submit the tar file. Also, please provide a *loader file*, which will load all your files into lisp. This file should contain a sequence of lines such as:

```
(load "<filename1>")
(load "<filename2>")
```

etc.