
MP 4 – User Defined Datatypes and Recursion

CS 421 – Fall 2008

Revision 1.0

Assigned September 23, 2008

Due September 30, 2008 23:59

Extension 48 hours (20% penalty)

1 Change Log

1.1 Problem 13 example fixed, problem 11 brackets now visible

1.0 Initial Release.

2 Objectives and Background

The purpose of this MP is to help the student master:

1. user-defined datatypes
2. recursion over those datatypes

3 Instructions

You may use any and all library functions. We highly recommend the `List` library, for which documentation can be found at <http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html>.

In particular, you may want to look at:

- `length`
- `concat` (also `flatten`)
- `map`
- `fold_left`
- `fold_right`
- `mem`

You may also find other `List` library functions useful.

4 XML Datatype

XML (eXtensible Markup Language) is a very general HTML-like language for encoding documents and data. An XML document consists of some header information and a list of XML items. A typical XML item looks like this:

```

<class title = "CS421" instructor="Elsa L. Gunter">
  <mp name = "mp1">
    This MP tests your understanding of simple arithmetic.
    <problem> What is 2 + 2? </problem>
    <problem> What is 4 * 3? </problem>
  </mp>

  <mp name = "mp2">
    This MP tests your understanding of common sense.
    <problem> Which weighs more, a pound of feathers or a pound of bricks? </problem>
    <problem> Does an apple a day really keep the doctor away? </problem>
  </mp>

  <mp name = "mp3"/>
</class>

```

For this assignment, we are not interested in header information, comments, or other advanced features of XML. We model the core of XML with the following three datatypes:

```

type assignment = { attribute : string; value : string }
type tag = { tagname : string; assignments : assignment list }

type xml =
  Element of tag * xml list
  | CharData of string

```

The type `xml` is the type of XML items. Note that an XML item is either some text (`CharData of string`) or a tag followed by child items (`Element of tag * xml list`).

A tag consists of a `tagname` followed by a list of assignments, where an assignment is an attribute (string) paired with a value (also a string.) For example, the following XML item:

```

<outfit price='$14'>
  A really nice but inexpensive outfit.
  <pants price='$8' />
  <tie price='$2' />
  <shirt price='$4' />
</outfit>

```

is represented by the following data structure of type `xml`:

```

Element({tagname = "outfit"; assignments = [{attribute = "price"; value = "$14"}]},
 [
  CharData "\n  A really nice but inexpensive outfit.\n  ";
  Element ({tagname = "pants"; assignments = [{attribute = "price"; value = "$8"}]}, []);
  Element ({tagname = "tie"; assignments = [{attribute = "price"; value = "$2"}]}, []);
  Element ({tagname = "shirt"; assignments = [{attribute = "price"; value = "$4"}]}, [])
 ])

```

Note: from now on, we use `'` instead of `"` to allow us to represent an entire XML document as a string by wrapping it in double quotes (`"`). The function `xml_parse`, described below, takes such a string as input.

5 Provided Functions

The following functions are included in `mp4common.ml`, and you may use them freely:

- `xml_parse : string -> xml`
returns an object of type `xml` representing the given string of XML code. The string of XML code must consist of a single “root-level” XML tag and its content.
- `xml_print : xml -> unit`
prints a representation of the given `xml` object that uses extra whitespace to highlight the tree structure.
- `to_set : 'a list -> 'a list`
sorts a list of objects and removes duplicates. In the problems that follow, we will sometimes request that you use this function.

6 Provided XML Item Bindings

The examples for some problems will refer to the following XML items defined in `mp4common.ml`:

```
(*** XML Items ***)
let xa = xml_parse "
<person name='Joe' age='23'>
  Joe is a Junior in Computer Science.
  <project name='compiler'> Build a C++ compiler </project>
  <project name='interpreter'> Build a Java interpreter </project>
</person>
"

let xb = xml_parse "
<box width='12' height='4' length='3' width='12' height='4' width='3'>
  The outer box
  <box width='14' height='3'> Inner box A </box>
  <box width='3' color='blue'> Inner box B </box>
</box>
"

let xb_2 = xml_parse "
<box width='12' height='4' length='3' width='3'>
  The outer box
  <box width='3' color='blue' length='default'> Inner box B </box>
  <box width='14' color = 'default' height='3'> Inner box A </box>
</box>
"

let xc = xml_parse "
<node>
  <node>1</node>
  <node>2</node>
  <node>
    <node>3</node>
  </node>
"
```

```
    <node>4</node>
  </node>
  <node>
    <node>5</node>
    <node>6</node>
  </node>
</node>
"
```

```
let xc_2 = xml_parse "
<node>
  <node>1</node>
  <node>
    <node>6</node>
    <node>5</node>
  </node>
  <node>
    <node>3</node>
    <node>4</node>
  </node>
  <node>2</node>
</node>
"
```

```
let xc_3 = xml_parse "
<node>
  <node>1</node>
  <node>
    <node>8</node>
    <node>5</node>
  </node>
  <node>
    <node>3</node>
    <node>4</node>
  </node>
  <node>2</node>
</node>
"
```

7 Problems

7.1 XML Tags

1. (3 pts) Write `tag_make_pair` such that `tag_make_pair tag` returns the pair with first component the tag-name of `tag`, and with second component the assignment list of `tag`.

```
# let tag_make_pair tag = ...;;
val tag_make_pair : tag -> string * assignment list = <fun>
# tag_make_pair {
  tagname = "class";
  assignments = [{ attribute = "instructor"; value = "Elsa L. Gunter" }]
};;
- : string * assignment list =
("class", [{attribute = "instructor"; value = "Elsa L. Gunter"}])
```

Hint: you may want to replace `tag` in `let tag_make_pair tag = ...` with a more useful pattern. Alternatively, you can use matching or the dot (`.`) notation to access record fields.

2. (3 pts) Write `tag_get_attributes` such that `tag_get_attributes tag` returns the attributes of the assignments in `tag`, in the order that they are specified and retaining any duplicates,

```
# let tag_get_attributes tag = ...;;
val tag_get_attributes : tag -> string list = <fun>
# tag_get_attributes { tagname = "box"; assignments = [
  { attribute = "length"; value = "3" };
  { attribute = "width"; value = "4" };
  { attribute = "height"; value = "5" };
  { attribute = "length"; value = "7" } ] };;
- : string list = ["length"; "width"; "height"; "length"]
```

Hint: you may want to replace `tag` in `let tag_get_attributes tag = ...` with a more useful pattern. Alternatively, you can use dot (`.`) notation to access record fields.

3. (5 pts) Write `tag_assignments_to_set` such that `tag_assignments_to_set tag` returns `tag` with its assignments ordered and duplicate assignments removed.

Assignments are to be ordered alphanumerically by attribute, breaking ties by alphanumeric ordering on values.

Note that two assignments to the same attribute associating it with two different values are distinct and must both appear in the list returned. Only assignments that assign the same attribute to the same value are considered duplicates.

Hint: use the provided function `to_set`, which takes a list and returns that list, with ordering and duplicate removal handled as specified above.

```
# let tag_assignments_to_set tag = ...;;
val tag_assignments_to_set : tag -> tag = <fun>
# tag_assignments_to_set { tagname = "box"; assignments = [
  { attribute = "length"; value = "3" };
  { attribute = "width"; value = "4" };
  { attribute = "height"; value = "5" };
  { attribute = "length"; value = "7" } ] };;
```

```

    { attribute = "length"; value = "7" }; { attribute = "width"; value = "4" } ] };;
- : tag =
{tagname = "box";
 assignments =
  [{attribute = "height"; value = "5"}; {attribute = "length"; value = "3"};
   {attribute = "length"; value = "7"}; {attribute = "width"; value = "4"}]}

```

Hint: you may want to replace `tag` in `let tag_assignments_to_set tag = ...` with a more useful pattern. Alternatively, you can use dot (`.`) notation to access record fields.

4. (7 pts) Write `tag_find_duplicate` such that `tag_find_duplicate tag` returns `Some a`, where `a` is an attribute that is assigned to multiple *distinct* values in `tag`, and is alphanumerically the first such multiply-assigned attribute, or returns `None` if there is no such attribute.

Hint: use your solution to problem 3 to help determine which duplicated assignment is alphanumerically the first.

```

# let tag_find_duplicate t = ...;;
val tag_find_duplicate : tag -> string option = <fun>
# tag_find_duplicate { tagname = "person"; assignments = [
  { attribute = "name"; value = "Joe" };
  { attribute = "age"; value = "23" };
  { attribute = "name"; value = "Fred" };
  { attribute = "age"; value = "23" };
  { attribute = "living"; value = "true" };
  { attribute = "living"; value = "false" } ] };;
- : string option = Some "living"

```

7.2 XML Items

Note: You are free to do the problems in this section in any order, and may find it convenient to re-order some of them.

5. (6 pts) Write `xml_map` such that `xml_map f_tag f_chardata x` returns `x` with every tag `tag` replaced by `f_tag tag` and with every string `s` underneath a `CharData` constructor replaced by `f_chardata s`.

```
# let xml_map f_tag f_chardata x = ...;;
val xml_map : (tag -> tag) -> (string -> string) -> xml -> xml = <fun>
# xml_print (xml_map (fun x -> x) (fun s -> s ^ s) xa);;
```

```
<person name="Joe" age="23">

  Joe is a Junior in Computer Science.

  Joe is a Junior in Computer Science.

  <project name="compiler">
    Build a C++ compiler  Build a C++ compiler
  </project>
  <project name="interpreter">
    Build a Java intepreter  Build a Java intepreter
  </project>
</person>

- : unit = ()
```

6. (8 pts) Write `xml_fold` such that `xml_fold f_element f_chardata x` folds over `x`, returning `f_chardata s` for `xml` items matching `CharData s`, and returning `f_element tag rls` for `xml` items matching `Element (tag, xmls)`, where `rls` is the list resulting from folding over the child items `xmls`.

```
# let rec xml_fold f_element f_chardata x = ...;;
val xml_fold : (tag -> 'a list -> 'a) -> (string -> 'a) -> xml -> 'a = <fun>
# xml_print (xml_fold (fun tag xmls -> Element(tag, xmls)) (fun s -> CharData s) xa);;
```

```
<person name="Joe" age="23">

  Joe is a Junior in Computer Science.

  <project name="compiler">
    Build a C++ compiler
  </project>
  <project name="interpreter">
    Build a Java intepreter
  </project>
</person>

- : unit = ()
```

7. (4 pts) Write `xml_uppercase_text` such that `xml_uppercase_text x` returns `x` with all text data (that is, strings under the `CharData` constructor) converted to uppercase.

Use `String.uppercase : string -> string` to ensure all characters are handled correctly.

Hint: use `xml_map`.

```
# let xml_uppercase_text x = ...;;
val xml_uppercase_text : xml -> xml = <fun>
# xml_print (xml_uppercase_text xa);;
```

```
<person name="Joe" age="23">
```

```
    JOE IS A JUNIOR IN COMPUTER SCIENCE.
```

```
    <project name="compiler">
```

```
        BUILD A C++ COMPILER
```

```
    </project>
```

```
    <project name="interpreter">
```

```
        BUILD A JAVA INTEPRETER
```

```
    </project>
```

```
</person>
```

```
- : unit = ()
```

8. (4 pts) Write `xml_attributes_to_set` such that `xml_attributes_to_set x` returns `x` with the attributes of every tag sorted and with duplicates removed, as specified in Problem 3

Hint: use your solution to Problem 3 as a sub-routine.

```
# let xml_attributes_to_set x = ...;;
val xml_attributes_to_set : xml -> xml = <fun>
# xml_print (xml_attributes_to_set xb);;
```

```
<box height="4" length="3" width="12" width="3">
```

```
    The outer box
```

```
    <box height="3" width="14">
```

```
        Inner box A
```

```
    </box>
```

```
    <box color="blue" width="3">
```

```
        Inner box B
```

```
    </box>
```

```
</box>
```

```
- : unit = ()
```

9. (5 pts) Write `xml_frequency` such that `xml_frequency tagname x` returns the number of occurrences in `x` of tags with tagname `tagname`.

Hint: use `xml_fold`.

```

# let xml_frequency tagname x = ...;;
val xml_frequency : string -> xml -> int = <fun>
# xml_frequency "person" xa;;
- : int = 1
# xml_frequency "box" xb;;
- : int = 3

```

10. (7 pts)

Write `xml_filter_tags` such that `xml_filter_tags tn_lst x` returns `Some x'`, where `x'` is the result of removing from `x` all sub-items with tags whose names are not in `tn_lst`, or `None` if this would cause the root item to be excluded. The subitems left should occur in the same order as in `x`.

Hint: use `List.mem` to determine if an element is in a list.

```

# let rec xml_filter_tags tn_lst = ...;;
val xml_filter_tags : string list -> xml -> xml option = <fun>
# xml_filter_tags ["dog"; "cat"] xa;;
- : xml option = None
# xml_filter_tags ["person"; "dog"] xb;;
- : xml option = None
# let Some xa' = xml_filter_tags ["person"] xa in xml_print xa';;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
None

```

```
<person name="Joe" age="23">
```

```
  Joe is a Junior in Computer Science.
```

```
</person>
```

```
- : unit = ()
```

7.3 Supplying Default Values to XML Attributes (Helpers)

11. (7 pts) Write `xml_get_tags` such that `xml_get_tags x` returns the list corresponding to the set of all tags occurring anywhere in `x`, sorted and with duplicates removed, as by an application of `to_set`. You *must* sort and remove duplicates exactly as `to_set` does, so it is highly recommended that you use `to_set` as a sub-routine.

Hint: use `xml_fold`.

```
# let xml_get_tags x = ...;;
val xml_get_tags : xml -> tag list = <fun>
# xml_get_tags xa;;
- : tag list =
[{:tagname = "person";
  assignments =
    [{attribute = "name"; value = "Joe"};
     {attribute = "age"; value = "23"}]};
{:tagname = "project";
  assignments = [{attribute = "name"; value = "compiler"}]};
{:tagname = "project";
  assignments = [{attribute = "name"; value = "interpreter"}]}]
```

12. (7 pts) Write `xml_get_attributes` such that `xml_get_attributes tagname x` returns the list representing the set of all attributes ever associated with a tag with tagname `tagname` in `x`. The output should be sorted and without duplicates. You *must* sort and remove duplicates exactly as `to_set` does, so it is highly recommended that you use `to_set` as a sub-routine.

```
# let xml_get_attributes tagname x = ...;;
val xml_get_attributes : string -> xml -> string list = <fun>
# xml_get_attributes "project" xa;;
- : string list = ["name"]
# xml_get_attributes "box" xb;;
- : string list = ["color"; "height"; "length"; "width"]
```

7.4 Extra Credit

13. (10 pts) Write `xml_supply_defaults` such that `xml_supply_defaults x` returns `x` where, for each tag `t` with tagname `tn`, `t` assigns values to every attribute assigned to any tag with tagname `tn` anywhere in `x`, assigning to the value "default" if `t` is unassigned in `x`.

Additionally, all tags must have their assignments sorted and duplicates removed as specified in Problem 3.

Hint: you will want to use functions that you have defined previously.

```
# let xml_supply_defaults x = ...;;
- : Mp4common.Mp4common.xml -> Mp4common.Mp4common.xml = <fun>
# xml_print (xml_supply_defaults xb);;

<box color="default" height="4" length="3" width="12" width="3">

  The outer box

  <box color="default" height="3" length="default" width="14">
```

```

    Inner box A
  </box>
  <box color="blue" height="default" length="default" width="3">
    Inner box B
  </box>
</box>

- : unit = ()

- : unit = ()

```

14. (6 pts) Write `xml_reordering` such that `xml_reordering x y` returns true iff `xml_supply_defaults x` and `xml_supply_defaults y` are equivalent up to recursive re-ordering of sub-items.

Solutions that do not demonstrate progress toward a real solution will not receive credit, even if the auto-grader awards them points on some examples.

```

# let xml_reordering x1 x2 = ...;;
val xml_reordering : xml -> xml -> bool = <fun>
# xml_reordering xa xb;;
- : bool = false
# xml_reordering xb xb_2;;
- : bool = true
# xml_reordering xc xc_2;;
- : bool = true
# xml_reordering xc xc_3;;
- : bool = false

```