ECE 251 Assignment #1: Abstract Datatype Language (v2)

Patrick Lam[*]

Due: (1) Test cases, September 28; (2) Code: October 5

Be sure to consult the clarifications posted on the web page as well as the Javadoc for my sample solution for hints.

1 Problem Description

Suppose that you are assigned the job of teaching some high school students about abstract datatypes as part of a summer enrichment programme. You do not know if these students have any programming experience and you want your class to be accessible even to those who do not have any prior programming experience. This is a brief summer programme, so you don't have time to start by teaching them a conventional programming language: you'll never get to abstract datatypes if you start with procedures etc.

What to do? One option is to make this course just a pencil and paper exercise. Is there another option? Thankfully you've taken ECE251, so you have some skills at your disposal.

2 Solution: Design a Language

We design (and implement) a small domain-specific language to enable introductory students to experiment with abstract datatypes. Let us call this language ADL for 'abstract datatype language'.

2.1 Language Overview

Variables in ADL have one of three types: atom (single value), list, or tree. Variables do not need to be explicitly declared.

There are three kinds of statements: read, write, and assignments. Assignment statements have some kind of operation on the right-hand side. Variables are introduced by read statements and assignment statements.

Read and write statements work with three different file types (one for each kind of variable): `atom` (single value), `txt` (list), `xml` (tree). A read statement introduces a variable with the same name as the file. For example, the following statement introduces a new variable named `book` of type tree:

```
read book.xml
```

[*]Thanks to Prof. Derek Rayside for formulating and writing up this assignment!

Write statements are similarly simple: they are of the form `write var`, and cause a file named `output/var.ext`[1] to be written to disk, where `var` is the name of the variable and `ext` is the extension that corresponds to to the type of the variable. Print statements instead write the variable to standard output (to help you with interactive testing).

Assignment statements are the most complicated. The LHS names the variable being introduced. The RHS calls built-in functions.

A more sophisticated version of ADL might also include tables and graphs. However, including those abstract datatypes in the version of ADL for this assignment would make this assignment take too long to complete. Moreover, you will work with tables on the next assignment.

2.2 Examples

Consider a document as an ordered tree. This may take you a second. Think of a technical document that has numbered chapters and sections: for example, paragraph I.2.b of the Canadian Charter of Rights and Freedoms guarantees freedom of thought, belief, opinion, and expression.

If we wanted to skim the Charter we might traverse this tree breadth first: I.1 Rights and Freedoms in Canada, I.2 Fundamental Freedoms, I.3 Democratic Rights of Citizens, ... I.34 Citation; I.2.a freedom of conscience and religion; I.2.b, I.2.c, I.2.d, ...

If we wanted to read the Charter like a novel from beginning to end we'd do a depth-first traversal of this tree: I.1, I.2, I.2.a, I.2.b, I.2.c, ..., I.34.

The following program computes the table of contents for the Charter and writes them to `toc.txt`:

```
read charter.xml
toc := preorder (truncate 2 charter)
write toc
```

The first line creates a variable named `charter` of type tree with the contents of the file `charter.xml`. The second line contains two operations: it first chops off the lower levels of this tree, performs a preorder (depth-first) traversal of the intermediate tree, and stores the result in `toc` (which is a list). The last line writes out `toc.txt` with the value of variable `toc`.

[1] Use the appropriate `java.io.File.separator`.

Figure 1 `charter.xml` and `toc.txt`

charter.xml

```
<part> Canadian Charter of Rights and Freedoms

    <section> Guarantee of Rights and Freedoms
        <subsection> Rights and freedoms in Canada
            <paragraph>
             1.  The Canadian Charter of Rights and Freedoms
             guarantees the rights and freedoms set out in it subject
             only to such reasonable limits prescribed by law as can
             be demonstrably justified in a free and democratic
             society.
            </paragraph>
        </subsection>
    </section>

    <section> Fundamental Freedoms
        <subsection>
            <paragraph>  2.  Everyone has the following fundamental freedoms:
                <clause> (a) freedom of conscience and religion; </clause>
                <clause> (b) freedom of thought, belief, opinion and
                expression, including freedom of the press and other
                media of communication; </clause>
                <clause> (c) freedom of peaceful assembly; and </clause>
                <clause> (d) freedom of association. </clause>
            </paragraph>
        </subsection>
    </section>

    <section> Democratic Rights
        <subsection> Democratic rights of citizens
            <paragraph>
             3.  Every citizen of Canada has the right to vote in an
             election of members of the House of Commons or of a
             legislative assembly and to be qualified for membership
             therein.
            </paragraph>
        </subsection>
        <subsection> Maximum duration of legislative bodies
            <paragraph> 4.
                <clause> (1) No House of Commons and no legislative
                assembly shall continue for longer than five years from
                the date fixed for the return of the writs of a general
                election of its members. </clause>
                <clause> (2) In time of real or apprehended war,
                invasion or insurrection, a House of Commons may be
                continued by Parliament and a legislative assembly may
                be continued by the legislature beyond five years if
                such continuation is not opposed by the votes of more
                than one-third of the members of the House of Commons
                or the legislative assembly, as the case may
                be.</clause>
            </paragraph>
        </subsection>
        <subsection> Annual sitting of legislative bodies
            <paragraph>  5.  There shall be a sitting of Parliament
            and of each legislature at least once every twelve months.
            </paragraph>
        </subsection>
    </section>

</part>
```

toc.txt

```
Canadian Charter of Rights and Freedoms
Guarantee of Rights and Freedoms
Fundamental Freedoms
Democratic Rights
```

Figure 2 A simple example application of `map`

input **pi.txt**

```
3
1
4
1
5
9
```

program **pi_map.adl**

```
read pi.txt
pi_increment := map +1 pi
write pi_increment
```

output **pi_increment.txt**

```
4
2
5
2
6
10
```

Figure 3 A simple example application of `reduce`

input **pi.txt**

```
3
1
4
1
5
9
```

program **pi_reduce.adl**

```
read pi.txt
pi_sum := reduce sum pi
write pi_sum
```

output **pi_sum.atom**

```
23
```

2.3 Semantics

Programming Language Pragmatics §10.5 explains the higher-order functions `map`, `filter`, and `reduce`. These are also widely documented on the internet, and are built-in to most functional programming languages — and many other programming languages, such as Python. The `reduce` function is sometimes known as `fold`.

These functions are called 'higher-order' because one of their arguments is a function. For example, `sort` takes a comparator function that determines the order of the sort.

Tree traversals are well defined in many data structures textbooks and online on Wikipedia etc. Preorder is also known as depth-first. Levelorder is also known as breadth-first.

Applying expressions `sort`, `filter`, `reduce`, `*order`, and `truncate` on an `atom` give back the same `atom`.

2.4 Syntax

The grammar of ADL is listed in Figure 4. A few notes:

- Comments are shell-script style: i.e., any character after a # is considered to be a comment.
- VAR indicates a variable name. Variable names start with a letter, possibly followed by letters, digits, or underscores.
- LIT indicates a literal, which may be a sequence of digits representing an positive integer, or may be a quote-enclosed string. Your interpreter will need to distinguish integer literals from string literals.
- The plus ('+') operator concatenates strings and adds integers. Other arithmetic operators do not apply to strings. All predicates apply to strings.
- ADL does not have a notion of variable scope: there are no blocks nor procedures etc. Therefore you may simply use a hashtable to store the values of variables.

Figure 4 Grammar for ADL

```
STMT        := READ | WRITE | PRINT | ASSIGN
READ        := "read" VAR "." EXT
EXT         := "atom" | "txt" | "xml"
WRITE       := "write" VAR
PRINT       := "print" VAR
ASSIGN      := VAR ":=" EXP
EXP         := VAR | LIT
             | "(" EXP ")"
             | "sort" COMPARATOR EXP
             | "map" TRANSFORMER EXP
             | "filter" PREDICATE EXP
             | "reduce" ACCUMULATOR EXP
             | ("preorder" | "postorder" | "levelorder") EXP
             | "truncate" INT EXP
COMPARATOR  := "ascending" | "descending"
TRANSFORMER := ("+" | "-" | "*" | "/") LIT
PREDICATE   := (">" | ">=" | "<" | "<=" | "=") LIT
ACCUMULATOR := "sum" | "avg" | "count" | "concat"
```

3 Your Job

Your task is to implement an interpreter for ADL and to write some test cases.

A test case involves an ADL program with a corresponding set of input files and another corresponding set of output files. Test cases may share input files (i.e., you may do different computations with the same inputs).

3.1 Tools

We recommend that you do these assignments in Java and use the ANTLR parser-generator. We also recommend that you use the XML library discussed in tutorial. ANTLR is well documented at `antlr.org` and elsewhere.

You are free to choose other Java-based parser-generators (e.g., javacc), and other XML libraries without needing explicit staff permission. You may also implement your parsers entirely by hand, without using a parser-generator tool (although this is not recommended). The staff cannot support these alternative tools, libraries, and approaches. (You can ask questions, we don't guarantee answers.)

You will need explicit permission of the instructor to use a different programming language.

3.2 General Design and Implementation Strategy

We recommend that you approach this assignment in the following way:

1. Write your test cases.
2. Design and implement an API for the ADL operations:
   (a) For each file type, in order of increasing complexity (`atom`, `txt`, `xml`), implement:
      i. read
      ii. write/print
      iii. appropriate functions (e.g., sort, map, etc.)

   (b) Write a few small unit tests to ensure that your API does the right thing.
3. Implement a parser for ADL. The parser actions do nothing at this point. Just get parsing working.
4. Make the ADL parser actions invoke the API developed in step #1.

3.3 Parse Errors

For this assignment you may assume that every program your interpreter will encounter will be syntactically valid, so you don't have to worry about parse errors.

Generating nice error messages from poorly-formed input is hard and can make your parser much larger and more complicated.

The Eclipse Java Development Environment is an interesting case study. It has (at least) two parsers: one that produces nice error messages for the programmer, and one that acts as a front-end for the compiler. The Eclipse developers decided that it was better to maintain two smaller parsers rather than one large parser that tried to do two jobs.

3.4 Types

Each variable has a type: atom, list, or tree. Lists and trees may also contain either integers or strings, which affects the legal operations on those variables. Later in this course you will learn how to statically check that the program is type-correct. For this assignment you may assume that every program is type-correct.

However, your interpreter will need to dynamically track the type of each variable so that it can compute and process `write` and `print` statements correctly.

3.5 Hand-In Instructions

Your solution should be submitted online at `http://cms.csuglab.cornell.edu/web/guest/?action=externallogin`. The `main` method of your program should be in a Java class named `ca.uwaterloo.ece251.ADL`. It should take one argument, which is the name of an an ADL program file. Submit both the source code and the `.class` files, as well as any external libraries we need to run the code. **You must acknowledge any external code that you used** (apart from trivial code snippets, e.g. 3-5 lines to read files).

Please submit your test cases by Tuesday, September 28. I will then publish all of the test cases on the course web site along with a test harness, so that you can test your program on the set of test cases.

3.6 Marking Scheme [20 marks total]

- Quality and diversity of your 6 hand-written test cases [**3 marks**].
- Number of interpreters that fail your test cases [**3 bonus marks**]. We will use your test cases as inputs for ADL interpreters written by other students. You get bonus marks if you make their interpreters choke.
- Number of test cases your interpreter passes [**17 marks**]. We will run your ADL interpreter on tests written by the staff and by other students. Try to pass as many tests as possible.

If you hand in an interpreter that does not compile, you will receive 0 points for the interpreter. We reserve the right to inspect the source code and give you more marks for especially high-quality code or take away marks for code that looks too "hacky".

`atom` and `txt` output files will be compared with the standard GNU diff tool. `xml` output files will be compared with the diffxml tool (`http://diffxml.sf.net/`). We will not consider output on standard output. (That is, the "print" statement is purely to help you during debugging.)