CS403: Programming Languages

# The Designer Programming Language

Version 4a

## Preliminary information

Your task is to build an interpreter for a general purpose programming language of your own design. Your language must support the following features:

- comments
- integers and strings
- dynamically typed (like Scheme and Python)
- classes/objects
- arrays with O(1) access time
- conditionals
- recursion
- iteration
- convenient means to access command line arguments
- convenient means to print to the console
- convenient means to read integers from a file
- an adequate set of operators
- anonymous functions
- functions as first-class objects (i.e. functions can be manipulated as in Scheme - e.g. local functions)
- (graduate only) an inheritance system

The only basic types you need to provide are integer and string and you do not need to provide methods for coercing one type to another (although you may find it convenient to do so). The efficiency of your interpreter is not particularly important, as long as you can solve the test problem in a reasonable amount of time. Your language also does not need to support reclamation of memory that is no longer needed. You are to write your program in a statically-typed, imperative language such as C, C++, or Java. Check with me first if you wish to use some other host language.

## Test Problem

You must implement an AVL tree in your language, with the implementation based upon the pseudocode found at `http://beastie.cs.ua.edu`
The AVL tree will store integers. You must also implement an interpreter that reads commands from a file. The commands are:

- 0 - display the tree
- 1 `NNN` - insert the number `NNN` into the tree

Place your program in a file named *iAVL*. Your *main* should look something like:

```
function main()
    var fp = open(getCommandLineArgument(1));
    interpreter(fp);
    close(fp);
    return 0;
    )
```

Suppose these commands are found in the file *commands*:

```
1 3
1 2
1 6
1 5
1 1
1 8
1 7
0
```

Then the output of:

```
run iAVL commands
```

would be an in-order traversal of the tree:

```
1(2+) 2+(3-) [3-] 5(6-) 6-(3-) 7(8+) 8+(6-)
```

A plus sign is used if the balance factor of the node is +1, while a minus sign is used if the balance factor is -1. Parent values are enclosed in parenthesis and the root value is enclosed in square brackets. There are no spaces or tabs before the first value and no spaces or tabs after the last (only a newline). There is exactly one space separating values. The UNIX command *diff* will be used to compare outputs, so these rules must be strictly followed.

The next section talks about how to set up the *run* shellscript.

## Grading

- [100 points] interpreter works, sample problem works
- [90 points] interpreter works, sample problem not implemented
- [50 points] pretty printing
- [30 points] recognizing

Failure to correctly implement the test program will result in a 10 point deduction. Each feature not correctly implemented will result in a 10 point deduction. You will receive a 10 point deduction if any rule in your makefile causes a pause for input (see the makefile rules below).

*For Undergraduates*: if you do not, at least, implement a recognizer for your language, you will fail the course.

*For Graduates*: if you do not, at least, implement an pretty printer, you will fail the course.

## Testing your implementations

Your makefile should respond the command

```
make
```

which builds your processor and to the following commands, each of which illustrates a feature of your language:

```
make error1
make error1x
make error2
make error2x
make error3
make error3x
make error4
make error4x
make error5
make error5x
make arrays
make arraysx
make conditionals
make conditionalsx
make recursion
make recursionx
make iteration
make iterationx
make functions
make functionsx      # shows you can pass functions and return nested functions
make lambda
make lambdax
make objects
make objectsx        # get and update field, method call (e.g. setter)
```

The first rule in a pair of rules should print out the appropriate input program, while the *x* rules should execute that program. In particular, the first three error rules should show off your parser detecting three different kinds of syntax errors, while the last to error rules should demonstrate the detection of two different kinds of semantic errors.

Your makefile should also respond to the commands:

```
make problem
make problemx
```

These commands display the test problem of your implementation and run the test problem, respectively. If you don't implement your AVL, then you should demonstrate summing the integers found in a file.

Finally, provide an executable shellscript named *run* that runs a given program.

```
#!/bin/bash
./mylang $*
```

Note: in the case of recognizing only, only the error rules need be present in your makefile. In the case of pretty printing only, the *x* rules should run the input program through the pretty printer.

Finally, your makefile should respond to the command:

```
make clean
```

This command should remove all compilation artifacts (such as `.o` or `.class` files) so that a clean compile can be performed.

Makefiles for graduate students should additionally respond to the commands:

```
make grad
make gradx
```

These rules should thoroughly demonstrate the additional requirements for graduate students.

## Submitting your assignment

To submit assignments, you need to install the *submit system*:

- *Linux or Windows Bash instructions*
- *Mac instructions*

For your submission, use the command:

```
submit proglan lusth dpl
```

The *submit* program will bundle up all the files in your current directory and ship them to me. Thus it is very important that only the files related to the assignment are in your directory (you may submit test cases and test scripts). This includes subdirectories as well since all the files in any subdirectories will also be shipped to me, so be careful. You may submit as many times as you want before the deadline; new submissions replace old submissions.

To prepare for submission, place all your source code, sample programs, a README detailing how to run and write programs in your language, and a makefile for building your system into one directory. Name the README file *README*.

## Change log

- Version 4 : Feb 21 10:25:30 : gave a more expansive example for AVL
- Version 3 : Feb 20 09:28:05 : added objects targets for makefile
  clarified run script
  clarified problem makefile targets
  removed delete from AVL specification
- Version 2 : Thu Feb 14 10:37:49 : changed the dropbox to *dpl*