

## CSC 310 - Imperative Programming Languages, Spring, 2010, Dr. Dale E. Parson

Assignment 5, re-implementing Assignment 3 using Python classes and objects for the opcodes. Assignment is due at 11:59 PM on April 30, 2010.

Perform these steps to access the initial source, which is given below.

```
cp ~parson/ProcLang/objects_minimake_assn5.zip ~/ProcLang
cd ~/ProcLang
/bin/unzip objects_minimake_assn5.zip
cd ./objects_minimake
gmake clean test
```

This test fails initially. You need to re-implement the opcode logic from Assignment 3 using Python classes, objects, and references to bound object methods in place of the nested functions and closures of Assignment 3. You also need to add a new, second testing subdirectory similar to the sortdemo/ testing subdirectory.

Files you must change are `mmcompile.py`, `makefile` and a new subdirectory for testing that is similar to `sortdemo/`. You can have a simple testing subdirectory that compiles a simple C++ source file and runs it. You need to have a second test case in addition to `sortdemo/`, but it need not be complex.

### objects\_minimake/mmcompile.py

```
1 # Module mmcompile in package objects_minimake defines the compiler for
2 # a mini-make-like compiler. This module is the basis for several student
3 # projects in Spring 2010 CSC 310 Imperative Programming Languages.
4 # Initial code by D. Parson.
5 # *****
6 # Assignment 5 requires reimplementing of the VM opcodes in the
7 # form of Python classes and objects. See compileRules() below.

246
247 def compileRules(symboltable):
248 # Using my solution to Assignment 3 in ~parson/ProcLang/compile_minimake
249 # file mmcompile.py as a model for the algorithms, re-implement function
250 # compileRules as follows.
251 # 1. There are *NO* nested functions compileDependency, opcodeDependency,
252 # compileTimeCheck, opcodeTimeCheck, compileAction or opcodeAction.
253 # 2. Replace the above functions with three Python classes NESTED
254 # WITHIN THIS FUNCTION named classDependency, classTimeCheck and
255 # classAction. Each is derived from Python's "object" base class.
256 # 3. Each of these classes has a constructor (named __init__ according
257 # to Python built-in convention) that takes a "self" parameter
258 # for the "this" pointer, followed by the parameters that will be
```

```
259 # needed to run the opcode later. These additional parameters are
260 # simply the parameters accepted by compileDependency,
261 # compileTimeCheck and compileAction of Assignment 3, along with
262 # any other variables or parameters bound by the opcode closures
263 # of Assignment 3. Instead of storing these parameters in a
264 # closure, the __init__ constructor must store them in fields in
265 # the (classDependency, classTimeCheck or classAction) object being
266 # constructed.
267 # 4. Each of these classes must define a method called "eval" that
268 # takes the object reference (self) and a timestamp as parameters,
269 # and that returns a timestamp according to the logic of the opcode
270 # closures of Assignment 3.
271 # 5. In addition to defining these three nested classes, function
272 # compileRules must initialize the result dictionary as before,
273 # and then populate it with a mapping from executable target names
274 # to opcodes as before. The compilation code of compileRules uses
275 # the same basic logic as Assignment 3, but for Assignment 5 an
276 # opcode is a reference to a METHOD BOUND TO AN OBJECT. That means
277 # that, for every a) dependency, b) timecheck and c) action, compileRules
278 # must construct an object of class classDependency, classTimeCheck
279 # or classAction using the appropriate parameters, and then must store
280 # a pointer to the "eval" method FOR THAT OBJECT. I will hand out
281 # some demo code concerning Python classes, objects, unbound method
282 # references and bound method references that will clarify the
283 # form that an opcode must take.
284 # 6. Check the "makefile" in the objects_minimake for additional STUDENT
285 # work.
286 #
287 # SUMMARY: This assignment replaces the function closures of Assignment 3
288 # with object closures as defined in Section 3.6.3 of the textbook.
289 #
290 # SEE: ~parson/ProcLang/compile_minimake/mmcompile.py for the algorithms
291 # for compileRules and the opcodes.
292 #
293 pass # STUDENT Replace this line of code with code spec'd above.
294
295 def executeRules(opdictionary, target):
296     """
297     This function accepts as input parameters an opcode dictionary
298     returned by compileRules and a rule target name that is a key in
299     that dictionary. It executes the compiled opcodes for target and
300     all rules on which it depends, invoking action lists in
301     the operating system shell for all triggered rules.
302     """
303     if (not opdictionary.has_key(target)):
304         raise ValueError, "Rule target " + target + " is unknown."
```

```

305 ops = opdictionary[target]
306 timestamp = 0
307 try:
308     timestamp = os.stat(target).st_mtime # time of last mod.
309 except Exception:
310     timestamp = 0 # No target file, assure that it will fire.
311 for op in ops:
312     newstamp = op(timestamp)
313     if (not newstamp):
314         break
315     if (newstamp > timestamp):
316         timestamp = newstamp

```

#### objects\_minimake/test\_mmcompile.py (no changes required)

```

1 # Module test_mmcompile tests package objects_minimake module mmcompile.
2 # Initial code by D. Parson. No changes are needed for Assignment 5.
3
4 from objects_minimake.mmcompile import *
5 from sys import argv, stdin, stderr, exit
6
7 if __name__ == '__main__':
8     # Do this only if this module is executed as the main module.
9     if (len(argv) == 1):
10         filehandle = stdin
11     elif (len(argv) == 2 or len(argv) == 3):
12         filehandle = open(argv[1], "rU") # portable linefeed handling
13     else:
14         stderr.write(
15             "Invalid usage, test driver takes optional file name and target." \
16             + '\n')
17         exit(1)
18     symtable = parseRules(filehandle)
19     if (len(argv) > 1):
20         filehandle.close()
21     keys = symtable.keys()
22     keys.sort()
23     for k in keys:
24         print "TARGET "+ k + ":"
25         depends = list(symtable[k][0])
26         print "\tDEPENDS :", depends
27         actions = list(symtable[k][1])
28         for act in actions:
29             print "\tACTION: ", act
30 if (len(argv) > 1 and argv[1] == "makefile.txt"): # assn. 3
31     if (len(argv) == 3 and argv[2] == "compile"):

```

```

32     print "RULE COMPILER TESTS ADDED FOR PROJECT 5"
33     codedict = compileRules(symtable)
34     executeRules(codedict, 'test')
35 else:
36     print "RULE INTERPRETER TESTS ADDED FOR PROJECT 5"
37     interpretRules(symtable, 'test')

```

#### objects\_minimake/makefile (changes required)

```

# makefile for minimake project, objects_minimake assignment 5
# CSC310, Spring, 2010, Dr. Dale Parson.

test:         testc teststudent

testc:
    cd ./sortdemo && /bin/rm -f manysorts *.o *.out *.dif
    cd ./sortdemo && python ../test_mmcompile.py makefile.txt compile > ../
testc1.out 2>&1
    diff testc1.out testc1.ref > testc1.dif
    cd ./sortdemo && python ../test_mmcompile.py makefile.txt compile > ../
testc2.out 2>&1
    diff testc2.out testc2.ref > testc2.dif
    cd ./sortdemo && /bin/rm -f manysorts *.o *.out *.dif

# STUDENT: Write target teststudent that cd's into some other
# subdirectory that you create, and that contains a makefile.txt
# that drives the compilation and testing of some program. You can use
# target testc: and subdirectory sortdemo/ as a testing model, but you
# must supply a different makefile.txt in that subdirectory along with
# the rules to build and test a target application of your choice OTHER
# than the one in sortdemo/. Your makefile.txt must at a minimum contain
# the rule target 'test' since that is the target executed by my test driver
# in test_mmcompile.py.

teststudent:

```