

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

Assignment 2, scanning and parsing a simple makefile-like source language file using Python regular expressions in the **re module** along with file input and string functions. **Assignment is due at 11:59 PM on March 1, 2010.**

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

```
cp ~parson/ProcLang/minimake_assn2.zip ~/ProcLang
cd ~/ProcLang
/bin/unzip minimake_assn2.zip
cd ./minimake
gmake clean test
```

You will get a diff because you need to write a line-oriented parsing function in source file `mmcompile.py` that is currently missing.

Also, you must have your `PYTHONPATH` environment variable set for this code to work. If the following command fails in your shell:

```
echo $PYTHONPATH
```

Then edit the file called `“.login”` in your login directory and add this line:

```
setenv PYTHONPATH $HOME/ProcLang
```

Log out and then log back in again. Now `PYTHONPATH` should be set.

Comments in source file `mmcompile.py` give the details of your assignment:

ProcLang/minimake/mmcompile.py

```
1 # Module mmcompile in package 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 import re
7
8 def parserules(infile):
9     """
10     STUDENT ASSIGNMENT #2 is to write parserules as follows.
11
12     Parameter infile is a file that has been opened for input reading
13     by the caller. This function does not close this file.
14
```

15 This function reads infile, a line at a time, and parses make-like
16 rules using Python's regular expression library and string manipulation
17 functions. There can be the following FOUR TYPES OF LINES in infile.
18 Note that if any line ends with \ as its last character, parserules must
19 concatenate the following line (if any) to the line ending with \
20 before parsing that line. Two or more lines can be thus concatenated
21 using multiple adjacent lines ending in \. (Implementor's note:
22 Python's readline() function leaves a \n and possibly a \r attached
23 to the end of a line. When checking for a trailing \ character, it
24 will be followed by one or two of \n and \r.)
25

26 "Whitespace" in the descriptions below is any combination of " "
27 (space character), \t (TAB), \n (NEWLINE) and/or \r (CARRIAGE RETURN)
28 in any number, combination and order. It may be multiple characters.
29

30 1. A "RULEHEAD" starts out with a non-whitespace sequence of
31 characters, with this sequence ending with mandatory whitespace
32 followed by a ":" (colon). Note that there can be no leading whitespace
33 at the start of the line. The initial string is known as
34 the TARGET.
35

36 Following the colon is optional whitespace, followed by a sequence
37 of zero or more whitespace-separated DEPENDENCIES. The TARGET
38 depends on these sub-TARGETS.
39

40 The RULEHEAD ends at the end of its line, although \ at line ends
41 may cause a RULEHEAD to span several lines.
42

43 Any RULEHEAD must be followed by either an ACTION line or a COMMENT
44 line or end-of-file. Having one RULEHEAD immediately after another
45 is an ERROR.
46

47 2. An "ACTION" a line of text following a RULEHEAD. Continuation
48 character \ allows a single ACTION to span multiple lines, similar
49 to RULEHEAD. An ACTION LINE MAY NOT MATCH a RULEHEAD, and it may
not
50 match a COMMENT. A line is by default an ACTION line if it matches
51 neither the syntax of RULEHEAD nor COMMENT.
52

53 Following each RULEHEAD line are zero or more ACTION lines.
54 This sequence of ACTION lines is completed when reaching the next
55 RULEHEAD or COMMENT line or ERROR line.
56

57 3. A "COMMENT" line is one of the following:
58 3a. An empty line, consisting of no characters or solely of whitespace,
59 is a COMMENT LINE.

60 3b. A line with zero or more whitespace characters folowed by a
61 ‘#’ character (pound symbol) is a COMMENT line. Any
62 trailing ‘\’ character continues the COMMENT.
63

64 4. An “ERROR” line is any other line that does not match the above
65 formatting constraints. Also note that the presence of end-of-file,
66 a COMMENT line, or an ERROR line terminates a rule’s sequence of
67 ACTION lines. Therefore, if an ACTION line appears outside the scope
68 or a RULE (i.e., when it is not one of a sequence of ACTION lines
69 that immediatly follow at RULEHEAD line), treat it as an ERROR line.
70

71 If parserules encounters an ERROR line, then it must raise a ValueError
72 exception with a descriptive string that supplies the offending
73 line number and text of the ERROR line.
74

75 If parserules encounters two consecutive RULEHEAD lines with no
76 intervening ACTION or COMMENT lines, then it must raise a ValueError
77 exception with a descriptive string that supplies the offending
78 line number and text of the second RULEHEAD line.
79

80 If parserules encounters a final line (end-of-file) ending in
81 continuation character \, then it must raise a ValueError
82 exception with a descriptive string that supplies the offending
83 line number of the final line.
84

85 Function parserules parses its infile lines until end-of-file or
86 ERROR. For each RULEHEAD it creates a Python dictionary entry that uses
87 the RULEHEAD’s TARGET as a key. The dictionary data for TARGET consist of
88 a two-tuple (ordered pair). The first element is a tuple of zero or more
89 DEPENDENCIES for the RULE. The second element is a tuple of zero or more
90 ACTIONS for the RULE, with leading and trailing whitespace stripped from
91 ACTIONS using the string strip() operation. STUDENTS can construct these
92 values in mutable lists, and then convert these lists to immutable tuples
93 in the return value.
94

95 Function parserules returns this dictionary of all the rules that it
96 has parsed. If parserules finds a RULEHEAD that defines a TARGET that
97 is already defined in a previous RULEHEAD, it must raise a
98 ValueError exception with a descriptive string that supplies the offending
99 line number and TARGET. Exception ValueError will also be raised if an
100 ERROR line is parsed.
101 ““““““
102 return {} # **STUDENT** delete this line and write this function.

Here is the completed test driver supplied by me:

ProcLang/minimake/test_mmcompile.py

```
1 # Module test_mmcompile tests package minimake module mmcompile.
2 # Initial code by D. Parson.
3
4 from minimake.mmcompile import parserules
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):
12         filehandle = open(argv[1], "rU") # portable linefeed handling
13     else:
14         stderr.write("Invalid usage, test driver takes optional file name." \
15                     + '\n')
16         exit(1)
17     symtable = parserules(filehandle)
18     if (len(argv) == 2):
19         filehandle.close()
20     keys = symtable.keys()
21     keys.sort()
22     for k in keys:
23         print "TARGET " + k + ":"
24         depends = list(symtable[k][0])
25         print "\tDEPENDS :", depends
26         actions = list(symtable[k][1])
27         for act in actions:
28             print "\tACTION: ", act
```

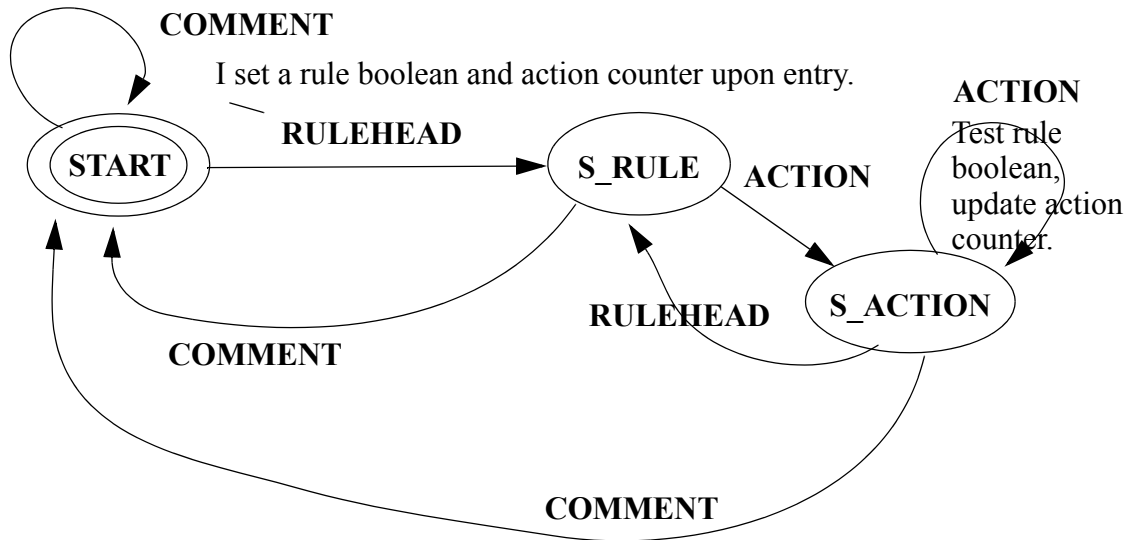
We will go over strategies for implementing the above function in class. Here are some strategies that I used in writing my version of function `parserules`.

1. I gave *parserules* a helper function for reading zero or more lines ending in a “\” continuation character, returning after reading a line with no continuation character, and appending the line strings together a line at a time. **Any helper function(s) that you write must be nested inside parserules!** My helper function takes one parameter, the current line number in the input file, and it returns a 2-tuple of two fields, namely the “logical text line” that it has read (from optional lines ending with “\” terminated with a line with no continuation character), followed by the updated line number. The helper function cannot modify variables local to *parserules*, but it can read

them. It can use the *infile* parameter. Of course you can design your own helper function(s). Helper functions are optional.

2. I used `re.compile` to compile patterns for CONTINUATION line checking, target RULEHEAD checking, and COMMENT line checking. An ACTION line is basically any line that does not match RULEHEAD or COMMENT. My helper function gets rid of CONTINUATION lines by gluing them together before matching for RULEHEAD or COMMENT lines.

3. A RULEHEAD line may not be adjacent to a RULEHEAD line without an intervening ACTION or COMMENT line. You can use a few boolean or integer counter variables within *parserules* to keep track of whether parsing is currently within the scope of a RULEHEAD.



Above is a deterministic finite automaton (DFA) that describes the parsing states that your parser will go through. The source file being parsed can end in any state, as long as it does not end on a “\” continuation character. Each RULEHEAD starts a new rule identified by its TARGET field. Any given TARGET symbol can be defined only once in its symbol table. My comments mean that I set a boolean flag to true when parsing a RULE, false when parsing a COMMENT, and test it when parsing an ACTION. I set an action counter to 0 when starting a RULE, increment it for every ACTION, and allow the RULEHEAD transition from S_ACTION back to S_RULE only if this action counter is > 0.

Python dictionaries, string slicing, and re matching are integral to this assignment. My supplied extensive tests should help. We will plan for this assignment in class. Please attend!