CS 3100 – Models of Computation – Fall 2011

Assignment 7, Posted on: 10/7. Due by **11/1/11 midnight**

- For those who chose the **L** ("large") option, this assignment is worth 20% of the total points for assignments. For **M** ("medium"), it is 14%, and for **S** ("small"), it is 8%.
- For ease of final grade calculation and display, we will regard S as having done asg7, M as having done asg7a in addition, and L as having done asg7b in addition.
- *By way of submission, you are required to submit one or more URLs for a website that contains a description of your work with links to the source files.* You must describe what you did and provide evidence (run results, new source code, figures, etc). Clear presentation is required to earn the points.
- Those who did S will thus furnish one URL. This must point to a single PDF called **asg7S.pdf**, and worth 8 points. You must carry out the work required in § 2 to earn these points.
- Those who did M will furnish the **asg7S.pdf** and *an additional URL*. This must point to a single PDF called **asg7M.pdf**, containing the extra work that is worth 6 points. (8+6 points). You must, in addition, carry out the tasks described in § 2 also carry out the tasks described in § 3.
- Those who did L will furnish the **asg7S.pdf** and **asg7M.pdf** URLs, and *an additional URL*. This must point to a single PDF called **asg7L.pdf**, containing the extra work that is worth 6 points. (8+6+6 points). You must, in addition to doing the tasks in § 2 and § 3, also carry out the tasks described in § 4.
- To preserve our sanity, there will be no late submissions allowed for asg7. Thus no late submission filenames need to be defined.
- If you chose "S" or "M", please keep the link to the code that I send you private (I'll be sending these links to the emails you've sent me).
- All of you will receive the full set of byte-codes to experiment with in a tarball `pyc.tar.gz`. There will be questions common to S, M, and L that are based on this tarball.
- I hope I don't overlook any of those emails; if you chose M or S and did not receive an email, do contact me. *If you do not send me an email, the default assumption will be that you chose the L option.*

1 Learning to use Python Regular Expressions

Out in the real world, you will be lexing and parsing in various ways. Often you will try to get away with a regular-expression based script. This is what this section teaches you to write. Hopefully you will come away with an impression of the power and perils of overlooked cases. For more "bullet-proof" lexing and parsing, you are better off using lex and yacc (or a tool such as `ply`).

Run the script below and explain how it is able to pick out function definitions from a file and print its arguments without leading blanks. Focus your description on how the `group`, `split`, and `match` functions work. This script is provided in file `pickout_fn_defs.py`.

```
# --- Script printFnArgs ---
#!/Library/Frameworks/Python.framework/Versions/Current/bin/python3
# Or wherever your python3 is
```

```
import os
import os.path
import sys
import re

def read_filename():
    while True:
        filename = input('Enter filename: ' )
        if os.path.exists(filename):
            print('Found file ' + filename)
            break
        print('Did not find file ' + filename + '. Try again')
    return filename

def printFnName(mLine):
    """Pick out the filename component of the match object.
    """
    print("Function: ", mLine.group(2))

def printArgs(mLine):
    """Split the argument list further based on commas. Then dig in and strip leading blanks.
    """
    for arg in re.split(',', mLine.group(3)):
        print("Arg: ", arg)

FnDefRe = re.compile(r'(^.*def\W*)(\w+)\W*\((.*)\):')
# Look for "def" and a blank in group 1; then word-char+; then (...) as group 3
# group-1 is 'def', group-2 is the fn name, group-3 are the args
#
def matchDefLine(line):
    return FnDefRe.match(line)

def process_all_lines(filename):
    f = open(filename)
    i = 1
    lineLst = f.read().split('\n')
    for line in lineLst:
        #print(line)
        mLine = matchDefLine(line)
        if mLine:
            print('---')
            printFnName(mLine)
            printArgs(mLine)
        #--
        i = i + 1
    f.close()

def main():
    filename = read_filename()
    print ('Filename = ' + filename)
    print ('Processing file ' + filename)
    process_all_lines(filename)

if __name__ == "__main__":
    main()
```

```
#*************************************************************
```

Now, if you run this program on an input file (say nfa.py) you find that

- It prints args for all functions that *do* have the args
- Alas, it prints the leading white spaces in front of each arg

So this function is nearly working.

2 ASSIGNMENT "S" TASKS

You are required to improve the above script (`printFnArgs`) in two ways:

- **1 point**: Make it print functions with no args also. In that case, print "No Args" after the function name is printed (saying that the function has no args).
- **1 point**: Eliminate the leading white spaces before each arg, using RE pattern matching.
- **1 point**: Repair the program so that it does not produce the following offending output:

```
---
Function:  W
Arg:  \w+)\W*\((.*)\
---
Function:  matchDefLine
Arg:  line
```

- **The following commands can be executed *after* going into the directory `pyc/` and issuing the command `from remindfa import *`.**
- **1 point**: Generate the outputs corresponding to case `n==4` of `remindfa.py`. The relevant part of the code to exercise is:

```
elif (n==4):
    re2nfa('(a+b)*babb', 'endsinbabbnfa')
    re2dfa('(a+b)*babb', 'endsinbabbdfa')
    re2mindfa('(a+b)*babb', 'endsinbabbmindfa')
```

  Go through the `pdf` files generated and confirm that the machines are correct.

- **1 point**: Write a short description of the conversions involved in the above three calls (about 3 sentences describing each call).
- **1 point**: Demonstrate that a linear increase in the size of an RE can give rise to an exponential increase in the size of the minimal DFA. Demonstrate using three successive REs that are grown in this manner.
- *Note that the state names keep changing as you play with the given code. All state names in this assignment correspond to the initial state names you see after you start the Python session afresh.*
- **2 points**: Take the NFA generated by the call `re2nfa('(a+b)*babb', 'endsinbabbnfa')`. Eliminate states `S4` and `S6`, and verify that the results are correct. You shall make a GNFA using function `mk_gnfa_N`, display this NFA using `dot_nfa`. Then you shall use function `del_state_from_gnfa_N` to delete states.

3 ASSIGNMENT "M" TASKS

To earn your extra 6 points, implement the missing items within `reparse.py`. *Do not reverse-engineer from the byte-code or obtain hints from friends who have the sources.*

4 ASSIGNMENT "L" TASKS

To earn your extra extra 6 points (*i.e.*, in addition to doing § 3 which fetches you the first 6 extra), you must implement function `nfa2dfa` (3 points) and function `minDFA`.

5 Documentation for the Python files given to you

You have been given several files that will be described in the sequel.

5.1 lang.py, praut.py

Function `dotsan_map` has grown to include `(` and `)`. There are also many more print utilities. Please study them.

5.2 dfa.py

There are functions to make `gnfa`s and also to delete a chosen state from a GNFA (function `del_state_from_gnfa`). Study these and ask me questions. Rerun the tests in these files. Also note that there are functions to make a DFA 'total' and also functions to make DFAs from a partial delta specification (after which you would typically make the DFA "total"). There is also a `prdfa` utility here - generates a handy tabular listing!

5.3 nfa.py

Be sure to play with all the NFA-related functions such as `eclosure`, `run_nfa`, `accepts_nfa`, and `nfa2dfa` (oops those who chose "L" will be writing this!). Everyone can of course see the tests and learn!

5.4 minDFA.py

Here, one can see the `minDFA` function as well as how to draw a DFA with representative states. Again, those who chose the "L" option will be writing these functions.

5.5 ply, reparse, remindfa

You've been provided the `ply` directory containing the lexer and parser. File `reparse.py` has the full parser (with parts shaved off for those who chose M or L). `reparse.pdf` has the output of running various calls. The call `re2nfa`, `re2dfa`, and `re2mindfa` the three calls executed by reparse.py The first merely produces an NFA from an RE, the second converts the NFA to a DFA, and the third converts it to a minimal DFA.

The best illustration is of course obtained by running `remindfa.py`. Some of the outputs generated will also be saved in the tarballs you receive. In addition, I am arranging *all* of you to be receiving a `pyc` directory which has the byte-code for running all these programs. This allows the `S`, `M`, and `L` groups to do `asg7` and `asg8`.

6 Overall Problem: Write a Compiler!

Create a program to read a regular expression, parse it according to the context-free grammar below, and emit an NFA:

```
RE -> RE + RE | RE RE | ( RE ) | RE* | epsilon | string
```

You must use `ply-3.4` for lexing and parsing, available from `http://www.dabeaz.com/ply/`. Unfortunately, the above grammar is ambiguous, and so you must rewrite it, as described in `http://www.eng.utah.edu/~cs3100/lectures/l15/notes15.pdf`
More specific problem statement:

- Implement the function `reparse` that, given the input shown, parses it successfully

  ```
  reparse(' ( (a) (b + (c) (d) )* ( (c) ) ) + ( (g + (a) (b + (c) (d) )* (e) ) ) ( ( (f) (d) ) (b + (c) (d) )* (e) )* ("" + (f) + ( (f) (d) ) (b + (c) (d) )* ( (c) ) ) ')
  ```

- After parsing, it returns an NFA corresponding to this input RE. To explain this properly, I am going to show you how `reparse` provides function `re2mindfa`:

So all the action is packed into `yacc.parse`. This function is explained in the remaining sections. Yes, you'll be writing your first compiler! You can go with a huge advantage into your compiler class *and* also learn the 3100 material really really well! If anything is unclear, send teach-cs3100 and email.

7 Lexing and Parsing

Your best source for learning how to lex and parse is to read through `example/calc/calc.py` and `doc/ply.html`.

The idea is this. Your parser will be driven by grammar rules. To begin understanding how CFG rules work, start reading my notes on context-free parsing that I'll post as `http://www.eng.utah.edu/~cs3100/lectures/l15/notes15.pdf`.

Now, we don't want to parse REs ambiguously. That is, RE

```
a b + c d
```

is to be parsed as

```
(a b) + (c d)
```

and not as

```
a (b + c) d
```

To remove ambiguity, we need to stratify the CFG for REs as described in `http://www.eng.utah.edu/~cs3100/lectures/l15/notes15.pdf`

So for those who chose the L option, your goals are this:

- Define the rules for your Lexer
- Define a stratified grammar to parse REs unambiguously
- Once the parser is found to work, you must add semantic actions, which are to make an NFA.

8 Code Generation

This part is not too hard. You must call `mk_nfa` and make NFAs, and return them. Here is the rub: when you generate new NFA, you must generate unique state names. One trick I used to generate new state names is as follows:

```
def gensym(seed=0):
    n = seed
    while True:
        yield("s"+str(n))
        n += 1

St = gensym()
```

When you need to generate a new state name, do `next(St)` and that'll give you the new state name as a strings.

9 One last parsing task common to all

You must study the design of `calc` and see how the author "got away" stating the expression grammar as

```
E -> E+E | E*E | ...
```

To understand this, carry out this TASK

Re-express the RE grammar as

```
RE -> RE + RE | RE RE | RE* | ( RE ) | STR | EPS
```

meaning, go back from the current grammar to this grammar.
The current grammar is

```
expression -> expression + catexp | catexp

catexp -> catexp ordyexp | ordyexp

ordyexp -> ordyexp STAR | ( expression ) | STR | EPS
```

Then try parsing REs, and see what happens when you parse `a b + c d`. Write down your observations.