PDA parse with lambda productions

Due: Sun April 17, 11:59pm.

You may collaborate with *one or two* other students on this homework if you wish, or work alone. Collaboration must be true collaboration however, which means that the work put into each problem should be roughly equal and all parties should come away understanding the solution. Here are some suggested ways of collaborating on the programming part.

- Pair programming. Two or three of you look at the same screen and only one of you operate the keyboard. The one at the keyboard is the "driver" and the other is the "navigator". The driver explains everything they are doing as they do it and the navigator asks questions and makes suggestions. If the navigator knows how to solve the problem and the driver does not, the navigator should not dictate solutions to the driver but instead should tutor the driver to help them understand. The driver and navigator should switch roles every 10-15 minutes. Problems solved this way can then be individually submitted.
- Code review. The members of the collaborative each try to solve the problem independently. They then take turns analyzing each other's code, asking questions trying to understand each other's algorithms and suggesting improvements. After the code reviews, each of you can then fix your code using what you learned from the reviews. Do not copy code. If the result of code review is that your code needs changes to be more like your partner's, do not copy it. Instead recreate your own variant without looking at your partner's.

The goal is to learn enough from one another so that you each can do similar problems independently in an exam situation.

If you want a collaborator but don't know people in the class, you can ask on Discord and/or use the group-finding post on Piazza.

Homework Assignment

You analyzed the following context-free language as part of homework on Canvas. Use that analysis to write a PDA parser similar to the one found in http://krovetz.net/135/module_cfl/parse_intro.html

S' → S$
S → BA
A → +BA | -BA | λ
B → DC
C → *DC | /DC | λ
D → a | (S)

$

As is usually the case, S is the start symbol in this grammar, but an additional production was added that is not reachable from S. The production S' → S$ exists only to force $ into the follow set of S. The $ in this grammar represents the end of the token input stream and should not be taken litterally as a token. When the token stream is empty, next() will return None, and it is in this condition that you should activate a production with $ as one of its predictors. This means, for example, that when top=='A' and tok==None you should follow the A → λ production because $ is a predictor for that production in this case.

Starter Code

Here is the code from http://krovetz.net/135/module_cfl/parse_intro.html modified slightly to show you the detection of $ and also what to do when λ is the right hand side of a selected production (use the `pass` keyword to indicate a no-op). A page will soon appear on Mimir for you to submit your completed code.

```
class scanner:
    # toks[i] must evaluate to the i-th token in the token stream.
    # Assumes toks does not change during parsing
    def __init__(self,toks):
        self._toks = toks
        self._i = 0

    # If no more tokens exist or current token isn't s, raise exception.
    # Otherwise pass over the current one and move to the next.
    def match(self,s):
        if (self._i < len(self._toks)) and (self._toks[self._i] == s):
            self._i += 1
        else:
            raise Exception
```

```python
    # If any tokens remain return the current one. If no more, return None.
    def next(self):
        if self._i < len(self._toks):
            return self._toks[self._i]
        else:
            return None


# Input can be any type where len(input) is defined and input[i] yields a
# string (ie, string, list, etc). Raises Exception on a parse error.
def parse(input):
    toks = scanner(input)
    stack = ['S']
    while len(stack) > 0:
        top = stack.pop()      # Always pop top of stack
        tok = toks.next()      # None indicates token stream empty
        if top in ('a', 'b', 'x'):     # Matching stack top to token
            toks.match(top)
        elif top == 'A' and tok == None: # next == $
            pass # "pass" is how you say do nothing in Python
        elif top == 'S' and tok == 'a':  # S -> aSa must be the next
            stack.append('a')            # production to follow here
            stack.append('S')
            stack.append('a')
        elif top == 'S' and tok == 'b':  # S -> bSb must be the next
            stack.append('b')            # production to follow here
            stack.append('S')
            stack.append('b')
        elif top == 'S' and tok == 'x':  # S -> x must be the next
            stack.append('x')            # production to follow here
        else:
            raise Exception    # Unrecognized top/tok combination
    if toks.next() != None:
        raise Exception

try:
    parse("aabxbaa")
except:
    print("Reject")
else:
    print("Accept")
```