CnC-Python Partition String Example

Please refer to the Find-Primes example before starting with this example.

First, define the CnC graph for the partition string example:

PartitionString.cnc

```
<string singletonTag>;
<string spanTags>;
[string->string input];
[string->string span];
[string->string results];

<singletonTag> :: (python createSpan);
<spanTags> :: (python processSpan);

// program inputs and outputs

env -> [input];
env -> <singletonTag>;
[results] -> env;

// producer/consumer relations

[input] -> (createSpan) -> <spanTags>, [span];
[span] -> (processSpan) -> [results];
```

This graph is a little more interesting than the Find-Primes example since it has two Steps: `createSpan` and `processSpan`. `createSpan` produces Items for the `span` Item Collection. This in turn trigger the prescription of the `processSpan` step which writes results to the `results` Item Collection. The environment reads off these values from the `results` Item Collection.

Next, run the CnC-Python translator:

```
cnc_t PartitionString.cnc
```

Next, we need to provide the python implementations for the Steps using the generated templates (as explained in the Find-Primes example). Note the implementation of the `createAwaitsList` function in both the python Step classes which defines the input dependences on values from the Item Collections.

userPartitionStringApp.py

```
class Application:
  @staticmethod
  def onStart(args, input, singletonTag):
    # TODO fill out the body of the function
    # e.g. operation on output item collections: anItemCollection.put(aTag, aValue)
    # e.g. operation on tag collections: aTagCollection.putTag(aTag)
    if len(args) > 0:
      firstArg = args[0]
      print("py: processing " + firstArg)
      input.put(firstArg, firstArg)
      singletonTag.putTag(firstArg)
    else:
      print("py: usage PartitionStringMain <the_string_to_process>")

  @staticmethod
  def onEnd(results):
    # e.g. operation on input item collections: anItemCollection.get(aTag)
    # e.g. operation on input item collections: anItemCollection.printContents()
    results.printContents()
```

userCreateSpanStep.py

```
class CreateSpanStep:
  @staticmethod
  def createAwaitsList(tupleContainer, tag , inInput):
    # TODO fill out the body of the function
    # e.g. tupleContainer.add(itemCollection, tagValue)
    # e.g. operation on item collections: anItemCollection.get(aTag)
    tupleContainer.add(inInput, tag)

  @staticmethod
  def compute(tag , inInput, ctrlSpanTags, outSpan):
    # TODO fill out the body of the function
    # e.g. operation on input item collections: anItemCollection.get(aTag)
    # e.g. operation on output item collections: anItemCollection.put(aTag, aValue)
    # e.g. operation on tag collections: aTagCollection.putTag(aTag)
    def computeRun(x, y):
      if len(x) == 0:
        return [y]
      else:
        curRun = x[-1]
        if curRun[-1] == y:
          x[-1] = curRun + y
          return x
        else:
          x.append(y)
          return x

    inStrVal = inInput.get(tag)
    inList = reduce(computeRun, inStrVal, [])
    for spanStr in inList:
      outSpan.put(spanStr, spanStr)
      ctrlSpanTags.putTag(spanStr)
    return True
```

userProcessSpanStep.py

```
class ProcessSpanStep:
  @staticmethod
  def createAwaitsList(tupleContainer, tag , inSpan):
    # e.g. tupleContainer.add(itemCollection, tagValue)
    # e.g. operation on item collections: anItemCollection.get(aTag)
    tupleContainer.add(inSpan, tag)

  @staticmethod
  def compute(tag , inSpan, outResults):
    # e.g. operation on input item collections: anItemCollection.get(aTag)
    # e.g. operation on output item collections: anItemCollection.put(aTag, aValue)
    # e.g. operation on tag collections: aTagCollection.putTag(aTag)
    try:
      inStr = inSpan.get(tag)
      if len(inStr) % 2 != 0:
        outResults.put(inStr, inStr)
      return True
    except Error:
      print("py: NativeProcessSpanStep.compute error!")
      return False
```

Running this program with an input of `lllllllllaaaaaaaammmmmmmeeeeeeeeeeeeeee` should produce the following output:

```
Running PartitionStringMain
Starting PartitionStringMain...
...
py: processing lllllllllaaaaaaaammmmmmmeeeeeeeeeeeeeee
...
Contents of py:PartitionString.ResultsItemCollection [size=3]
'eeeeeeeeeeeeee' = eeeeeeeeeeeeee
'lllllllll' = lllllllll
'mmmmmmm' = mmmmmmm
```