AssistAssessment Documentation

1 Main Idea

The script accomplishes two main functionalities. The first is to format a XML file containing the problem specification to a latex-friendly format, and the second is to evaluate code for certain languages and compare the output to the output of a reference implementation in the problem file.

2 Dependencies

- This script was developed for python 3(3.8.5).
- For running Python code, it uses the command `python`. The source code is piped to stdin.
- For running Haskell code, it uses the command `stack ghc -- -o <binary> <sourceFile>`.
- For running CommonLisp code, it uses the command `sbcl --script <sourceFile>` (Steel Bank Common Lisp).

  The first line of the output is discarded as it is empty.
- For running PolyML code, it uses the command `poly --use <sourceFile>`.

  The solution definition is given by parameter and the tests are piped to stdin.

  The first line of the output is discarded as it is information about the version.

All of the above should be pretty painless to change as long as the behaviour of the new command is similar.

3 Problem description format

```
<problem>
<text>

</text>
<example language="Some Language">

</example>
<solution language="Some Language">

</solution>
<tests language="Some Language">

</tests>
</problem>
```

Above is the empty structure of the XML problem description file.
Some considerations:

- All whitespace get included in the formatted output, so the tags of this XML file and their contents shouldn't be indented.
- There can be multiple example, solution and tests tags but, for each specific language, only the first of each tag is considered.

  This means that problem descriptions in multiple languages can be contained in the same file.

  Example, solution and tests tags without a recognized language attribute will be ignored.
- Recognized languages: Python, Haskell, PolyML, CommonLisp.
- The `<text>` tag contains the text of the problem that will be used for formating.

  It supports a number of placeholders:

  - `[[function]]` will be replaced by the signature of the main function to be tested. Differs for each language.
  - `[[callExample]]`. Can have multiples of this placeholder. Each placeholder will be replaced by the coresponding line in the XML `<example>` tag. Can have fewer instances of this placeholder than lines inside the example tag.

  - `[[exampleResult]]`. Can have multiples of this placeholder. Each placeholder will be replaced by the coresponding result of running the respective example line against the solution. The number of these placeholders should match the number of `[[callExample]]` placeholders.
  - `[[genericExample]]` includes all examples specified inside the `<example>` tag and formats them in a sentence.

- The `<example>` tag expects a single example per line.

  An example is a simple call of a function. E.g. calling the Haskell function zip: `zip [1,2,3] ["a","b","c"]`.

- The solution tag should contain the correct implementation of the function that will be tested. For Haskell, its declaration should be on the first line. For Python and CommonLisp, the function to be tested should start on the first line. For PolyML, it doesn't matter.

- The `<tests>` tag is subject to the same restrictions as the `<example>` tag. It will be used to evaluate and compare desired sources to the reference implementation. The two tags are different to permit more extensive testing while maybe exemplifying more limited behaviour.

4 Usage

- For outputing a latex friendly format, the flags `-F, -p` and `-l` are relevant.

  The `-F` flag is mutually exclusive with `-E` and specifies the formatting operation.

  The `-p` should specify the path to a XML problem description file.

  For each `-p` there must be a corresponding `-l` flag that should specify one of the supported languages for which there are appropriately completed `<solution>` and `<example>` tags within the specified problem description file.

  There can be multiple pairs of `-p` and `-l` flags and the script will process them in the specified order.

  The formatted text is both outputted to stdout and written to a file for each pair of problem description and language.

E.g. `python assistAssessment.py -F -p zipList.xml -l Haskell -p zipList.xml -l Python -p genPrimes.xml -l PolyML` should return(the text displayed is not raw):

Write a function `myZip::[a] -> [b] -> [(a,b)]` which takes as arguments 2 lists and zips them. For example, `myZip ["a","b","c"] [1,2,3]` must definitely return `[("a",1),("b",2),("c",3)]`, while on the other hand, `myZip [1,2,3] ["a","b","c"]` should most certainly return `[(1,"a"),(2,"b"),(3,"c")]`.

Write a function `myZip(a,b)` which takes as arguments 2 lists and zips them. For example, `myZip(["a","b","c"],[1,2,4])` must definitely return `[('a', 1), ('b', 2), ('c', 4)]`, while on the other hand, `myZip([1,2,3],["a","b","c"])` should most certainly return `[(1, 'a'), (2, 'b'), (3, 'c')]`.

Write a function `primes = fn: int -> int list` which takes one argument, an integer n, and resturns a list containing the first primes up to n. E.g. `primes(20);` should return `[2, 3, 5, 7, 11, 13, 17, 19]: int list`.

- For evaluating source files and comparing their output to a reference implementation, the `-E, -p, -l, -s` and `-a` are relevant.

  The `-E` flag is mutually exclusive with `-F` and specifies the evaluating operation.

  The `-p` should specify the path to a XML problem description file.

  For each `-p` there must be a corresponding `-l` flag that should specify one of the supported languages for which there are appropriately completed `<solution>` and `<tests>` tags within the specified problem description file.

  - The `-s` flags without the `-a` should specify the path to a source file that should be compared to the reference solution. In this case, there can be multiple tuples of tags and there should be the same number of `-p, -l` and `-s` tags with the meaning that the source file specified by the first `-s` tag tries to solve the problem specified by the first `-p` tag in the language specified by the first `-l` tag and so on.

    E.g. `python assistAssessment.py -E -p flattenDeepList.xml -l Python -s sampleSolutions/p1.py -p genPrimes.xml -l PolyML -s sampleSolutions/genPrimes.ml` should return:

Evaluating sampleSolutions/p1.py:

For the correct answer:
[1, 2, 3, 4, 5, 6]
[2, 3, 4, 5, 6, 7]

The source file
sampleSolutions/p1.py
has produced the answer:
[1, 2, 3, 4, 5, 6]
[2, 3, 4, 5, 6, 7]

The given solution is correct.

Evaluating sampleSolutions/genPrimes.ml:

For the correct answer:
[2, 3, 5, 7, 11, 13, 17, 19]: int list
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]: int list

The source file
sampleSolutions/genPrimes.ml
has produced the answer:
[2, 3, 5, 7]: int list
[2, 3, 5, 7, 11, 13, 17, 19]: int list

The given solution is wrong.

– The `-s` flag, when the `-a` flag is specified, should point to an archive. The archive should have the proper extension and it will

be extracted by `shutil.unpack-archive`, which supports "zip", "tar", "tar.gz", "tar.bz", "tar.xz".

When using the `-a` flag, there should be a single `-s` flag specified.

The archive will be extracted to a directory of the same name. If said directory exists, the script will skip the extraction step, but the `-s` path should still point to the archive.

The expected structure of the archive is: the archive directly containing directories; one for each student; inside each directory is another archive which directly contains that student's source files.

E.g. `python assistAssessment.py -E -a -p flattenDeepList.xml -l Python -p zipList.xml -l Python -s studentśSolutions.zip` should return(It's a sample; the whole thing would be longer):

Grading bob's solutions:

Testing a new permutation of answers.

For the correct answer:
[1, 2, 3, 4, 5, 6]
[2, 3, 4, 5, 6, 7]
[3, 4, 5, 6, 7, 8]
[4, 5, 6, 7, 8, 9]
[5, 6, 7, 8, 9, 10]

The student
bob
has produced the answer:
[1, 2, 3, 4, 5, 6]
[2, 3, 4, 5, 6, 7]
[3, 4, 5, 6, 7, 8]
[4, 5, 6, 7, 8, 9]
[5, 6, 7, 8, 9, 10]

The given solution is correct.

For the correct answer:
[('a', 1), ('b', 2), ('c', 4)]
[(1, 'a'), (2, 'b'), (3, 'c')]
[(1, 1), (2, 2), (3, 3)]

The student
bob
has produced the answer:
[('a', 1), ('b', 2), ('c', 4)]
[(1, 'a'), (2, 'b'), (3, 'c')]
[(1, 1), (2, 2), (3, 3)]

The given solution is correct.

For this permutation of answers, bob has solved correctly 2/2 problems.

In the end, bob has solved correctly 2/2 problems.

Results:

```
Name | Number of sources | Grade
```

```
andrei | 1/2 | 1/2
costel | 2/2 | 0/2
bob | 2/2 | 2/2
```

5 Maintaining/Modifying

Both are encouraged. The script should be fairly readable, see comments for more details.

If support for another language is desired, there are two functions that are of concern.

The first is `evaluateSourceOnTest(source, test, language)`. This function is responsible for running code. Add an `if language == 'New Language'` block and assign to `result` the desired output.

The second is `formatTex(text, solution, example, language)`. This function is responsible for formatting the problem text to a TEX format. Add an `if language == 'New Language'` block and preprocess the `text`, `signature` and `example`.