## Tern: Use shlex to parse Dockerfile RUN instruction commands

## About me

1. Name: Junlai Wang.
   GitHub nickname: ForgetMe17
2. School: Beihang University, Beijing, China.
   Program: Signal Processing.
   Year: First year in the 2.5 years of schooling(Postgraduate), enrolled in 2019.9.
3. Time zone: UTC+8
4. Breif Resume:
   a. Received Bachelor Degree of Electronic Enginering in Beihang University, Beijing, China, at 2019.06.
   b. Developed a robot controlling system using Python which use WiFi and Website to control. Beihang University, Beijing, China, 2017.09 ~ 2018.05.
   c. Internship at Lenovo Research. Programming on Raspberry PI and Arduino with Python and C for a smart chair project. Beijing, 2019.4 ~ 2019.7.
   d. Developed a Wechat Mini-program for the EE Department in Beihang University. 2018.03~2018.06

## Code contribution

1. Document YAML data output that Tern produces #561
2. Parsing ARG varibales #580
3. Record git project name and sha #571
4. Find Git Project URL #606

## Project Infomation

1. Sub-org name: Tern

2. Project Abstract: Use shlex to parse Dockerfile RUN instruction commands. On looking at the type of parsing needed for full shell scripts embedded in the run command, we may need to develop a shell script parser to catch all places where software could have been installed.

3. Detailed Description: There are a few parsers for shell script, here are some references:
   a. sh. A shell parser, formatter, and interpreter. Supports POSIX Shell, Bash, and mksh. Requires Go 1.13 or later.
   b. bash-parser. Parses bash source code to produce an AST.
   c. bashlex. A Python port of the parser used internally by GNU bash.

   **Parsing**
   In this project, we need to catch all places where software could have been installed. In a shell script, most softwares are installed by some common command like tar, unzip, wget, git clone, apt install etc. What we need to do first is parsing the script

into statement parts: command, variables, if-statement, for-statement and case-statement. For most cases, we can match the installation place with the arguments of installation commands. Thus we need a command parser for a single line command which is like the implemented function parse_command in common.py. Then we can use a parsing rule to parse a command, eg, for command apt, words start with install should be a installation command and followed by the software name. We can make it as a yml file like base.yml, its structrue is like this:

- command name: installation command we need to match.
- subcommand: subcommand we need to match in words.
- software name: option arg or words.
- installation place: option arg or words.

Here are some examples:

- command name: apt, apt-get
- subcommand: install
- software name: 'words'(using parsed words, since we have a subcommand 'install' , so the first word in words which is 'install' will be excluded)
- installation place: apt default installation place

- command name: tar
- subcommand: None
- software name: 'words'(using parsed words)
- installation place: '-c'(using parsed option lists)

But it is unlikely to include all possible installation commands so we can also find the path-like variables and path-like arguments of commands. I do not know if this is necessary, so i just write it but not making plan on this.

**Output**

The output should include the installation place, corresponding script command and software name if possible.

**Plan**

step1. We can try to split commands from scripts first, and parse the command into command name, words and args. Then using the parsing rule to find the software location. If it contains variables, we just use the name not the value.

step2. Some commands may contains variables and we need to replace the variables into its value.

step3. Towards case, if , for statement, handle the nested commands and variables. For branch like case and if, we can give the output by branch, like

- branch : ${dpkgArch##*-} == amd64
    - software name, installation place

This can be implemented by giving the parsed variables a branch property which record its value by branch. When we need this variable as the output, we can form the output by its possible value.

4.Weekly timeline:

a. From now on, To May 19, 2020(Before Week 1): Working on current PRs and get started to the project. Communicating with the mentors about the project to get better understanding of the program. Making progress on step1.
b. From May 19, 2020 , To  June 16 - 20, 2020(Before First Eval): Complete step1 and move towards step2.
c. From June 16 - 20, 2020, To July 14 - 18, 2020(Before Second Eval):Working on step3.
d. From July 14 - 18, 2020, To August 11 - 18, 2020(Before Final Eval): Complete the full parser.