KRUN - A Language for Literate Programming

Khang Bao

March 11, 2021

ABSTRACT

KRUN is a language for literate programming, metaprogramming, and reproducible research. KRUN was designed in the way that a user of ed, the standard editor, can use it almost effectively as much as Emacs or Vi users would. By declarative language design, users can quickly utilize many great programming techinques, and easily script a demonstration that performing a demonstration.

Example output:

```
C       GPL
Java    GPL
Python  GPL
```

Users can produce the result like above with the input in the "file.kdoc":

```
.HEAD 1 "One Liner Filtering Using AWK"
.PP
Pipe the text to a file name LanguageData.txt using \*[ICS]echo\*[ICE].
.CODES bash Example1.sh pass111
echo "
AWK     DSL
grep    DSL
C       GPL
eqn     DSL
Java    GPL
Python  GPL
" > LanguageData.txt
.CODEE pass111
.
.PP
Using awk tool to filter all language that is General Purpose Language (GPL).
.CODES bash Example1.sh pass333
awk '/GPL/' LanguageData.txt
.CODEE pass333
.
.PP
Now we can remove the file:
.CODES bash Example1.sh pass222
rm -r LanguageData.txt
.CODEE pass222
.RUN bash Example1.sh
```

Processing the file above with the command:

```
cat file.kdoc | krun -r
```

1. Introduction

KRUN is a domain specific language for literate programming, reproducible research (RR), and metaprogramming. It's designed for programmers, researchers, or power users. It was initally a small project for me to submit for a CS113 final project assignment while I was working on KDOC macro package, and the first version was around 500 lines in Python written over an evening but is still actively maintained for more features or reimplemented in different language. KRUN can operate as a processor to execute the source code or as a preprocessor which can be used with kodelist (a code listing preprocessor written by Khang Bao) to pipe to Troff typesetting engine to produce typesetted paper.

KRUN was inspired partly by Donald Knuth published paper on literate programming system, Org-mode Babel, and Jupyter. KRUN addressed many hurdles of many popular RR tools and designed to aim for larger domain of users. Org-mode and Org-mode Babel are mostly usable on Emacs which makes it very hard for other users from wider range of text editor, IDE to use. Jupyter is mostly using Json file format which can be a challange to edit directly in the file. Most of popular RR tools bounds strongly to a interactive graphical environment, a single file format, and perfoms action on the file (alter the file) which makes them not easy to port for other software and workflow. KRUN designed as a simple filter which can be manipulated in many ways and can be used from within any text editors or environment while reserving the source file unaltered and safe.

KRUN can be used as processor/preprocessor for kdoc, a structural marked up language (a macro package written by Khang Bao in Troff language) but also can be easily configured to work with other marked up language or simple text file. Users can write their own macro package in Troff and use KRUN without any complicated modification.

This paper discusses design philosophy, language/program specification, and usage. Through reading this manual, users will not only able to use the program but also understand the code logic beind the program and some its application.

2. Design philosophy

KRUN follows many principles of the Unix philosophy and is designed to be used with Unix utilites. The program is minimal which allows other developers and users can quickly understand the code logic to extend the program to their wishes using Python programming language. KRUN is designed as a simple filter and a mini language which are noticiable concepts in the Unix design philosophy. Rather a large complex program that do-it-all, filter allows combining with other software easily and also swap with newer implementation effortless. Filter allows better separation between the raw source code and the output. Through some simple lines of shell script, users can set up a tool chain for producing quality paper and performing literate programming.

KRUN is never designed to be general-purposed like programming language but a language that specializes in enabling literate programming and metaprogramming. The language is very declarative, thus, making it very easy to learn, use and extend for more functionality, and it's much closer to natural language.

Keeping it portable and simple by using only robust Python standard modules, therefore, users can also easily run KRUN on their Unix based phone through the terminal emulator when they are on the go or wrap around a graphical interface for other purposes.

The language specification is compact, descriptive, and clear therefore, developers can easily implement it in other language like C or Rust. The program doesn't have a standard configuration file, the reason to this is that implementation in Python is already very reable and users can extend much further than a config file. Although, without a doubt, having a config file is much safer approach and probably easier to configure without the need to understand the engine program logic.

Some of the hurdles of literate programming keeping track of the current line number of the code session is almost impossible, the companian program KODELIST, by default, adds line number to each code session when outputting to Troff.

3. Program Mechanism

KRUN parses the document and collects every single code snippets enclosed by a set of special markers. krun engine can both able to parse from standard input or read from file, enabling some flexibilites and features. The program stores each session with its affiliated records like name, language, processor, and mode etc. Some commands can act like a switch altering different aspects like intepreter/compiler setup, session language, session name, etc.

4. Language Specification

4.1. Comamnds/Requests Summary

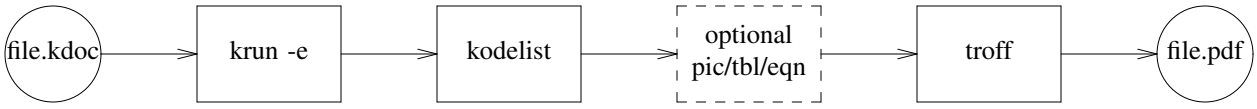
- `CODES [language] [name] [password]`: request to start a code segment.
    - `[language]`: optional argument to specify a language. If not specify or passing empty argument, the language is set as None.
    - `[name]`: optional argument to specify the session name. If not specify or passing empty argument, the name is set as None.
    - `[password]`: optional argument to distinguish which `CODEE` is actual and is very important when users want to embed a code snippet that also contains `CODEE`. If not specify or passing empty argument, the password is set as None.
- `CODEE [password]`: request to end a code segment.
    - `[password]`: optional argument to specify the password. The password should be matched with `CODES`, otherwise the requests will be treated as part of the code snippet.
- `LANGUAGE [language] [processor] [argument1] [argument2] ...`: request to add a new language to be interpreted.
- `PROCESSOR [language] [name] [processor] [argument1] [argument2] ...`: request to specify the interpreter/compiler for a code session. This request only works if the language is already specified to be run in the krun program source code or language added by `LANGUAGE` request.
- `RUN [language] [name]`: request to make the code snipped executed with run option/mode.
- `EVALUATE [language] [name]`: request to make the code snipped executed with evaluate option/mode.
- `CAPTURE [language] [name] [keyword]`: a request to capture the output to a keyword (use in metaprogramming).
- `TOFILE [language] [name] [filename]`: a request to output the code session to the file with tofile option/mode.
- `ORDER [language] [name]`: request to setup the order of the code snippet execution.
- `RESULTS [language] [name] [password]`: request to start a result block, this request should be generated by krun for kodelist preprocessor when evaluate option/mode is used.
- `RESULTE [password]`: request to end a result block, this request should be generated by krun when evaluate option is used.
- `HIDE`: Hide the block of code when producing paper. This command is actually implement in kdoc macro package than a command of KRUN but it's worth to mention in this list.

4.2. Built-in Keyword Summary

- `STDIN@STDIN@STDIN`: if invoke with getstdin mode, the standard input will replace the the keyword with the text stream. When in use, should be delimit left and right with `:::`.

5. Commandline Options

- Usage: `krun [option[argument]] [option[argument]]`
- Options:
    - `-help` or `-h` or `-?`: print the help message.
    - `-evaluate` or `-e`: use krun as evaluate mode and read from standard input.
    - `-run` or `-r`: use krun as run mode and read from standard input.
    - `-tofile` or `-t`: use krun as tofile mode and read from standard input.
    - `-output` or `-o`: use krun as output mode and read from standard input.
    - `-check` or `-c`: print the report of different checking.
    - `-stat` or `-s`: print out the statistic (in the yaml format) of the code snippets.
    - `-id <session-id> [option]`: operate only the session with specified id.
        - `<sessiom-id>`: session id of a code session in the file. The convension for a session-id combining the language of the session and the session name seperating by an alt symbol: `<language>@<name>`. If either language or name is empty, then the lanugage or the name should be called as `None`.
        - Options include:
            - `-evaluate` or `-e`: evaluate mode
            - `-run` or `-r`: run mode
            - `-tofile` or `-t`: tofile mode
            - `-output` or `-o`: output mode
            - `-statistic` or `-s`: print statistic in yaml format.
- Use krun in a pipeline:



6. Installation

krun program has no dependency or any external package beside the Python interpreter.

Users might need to install Kdocutils, Heirloom Troff/Groff/Neatroff, and ImageMagick (to use ps2pdf to convert PostScript to PDF) to produce paper in PDF.

Read README in the project package for installation guide.

7. Basic

To mark a code block in the file, use the marker `CODES` to start a block and `CODEE` to end a block.

```
1 .CODES python program.py pass123
2 print("Hello World", "Python")
3 .CODEE pass123
```

"kdoc" is the language for a session. "BasicExample.kdoc" is the name for a session. "pass123" is the password for that particular code segment.

8. Run Mode

Write the following code into a text file name "file.kdoc":

```
1 .CODES bash program.sh
2 echo -e "Hello World."
3 echo -e "Introduction to KRUN."
4 .CODEE
5 .RUN bash program.sh
```

Run the following command in the terminal get the output from the code block inside the "file.kdoc":

```
1 cat file.kdoc | krun -r
```

The following output will be printed to the standard output:

```
| Hello World.
| Introduction to KRUN.
|
```

9. Evaluate Mode

Write the following code into a text file name "file.kdoc":

```
1 .CODES bash program.sh
2 echo -e ".CS\nprint(1000 + 10)\n.CE" | sed "s/CS/CODES/g" | sed "s/CE
  /CODEE/g"
3 .CODEE
4 .EVALUATE bash program.sh
```

The following output will be printed to the standard output and can also be processed further by kodelist to produce Troff code:

```
1 cat file.kdoc | krun -e
```

```
| .CODES bash program.sh
| echo -e ".CS\nprint(1000 + 10)\n.CE" | sed "s/CS/CODES/g" | sed "s/CE
  /CODEE/g"
| .CODEE
| .EVALUATE bash program.sh
| .RESULTS bash program.sh resultPassword123331375
| .CODES
| print(1000 + 10)
| .CODEE
|
| .RESULTE resultPassword123331375
|
```

10. Tofile Mode

Write the following code into a text file name "file.kdoc":

```
1 .TOFILE markdown README.md
2 .CODES markdown README.md
3 # Introduction
4 KODELIST is a preprocessor formatting code listing for Troff typesett
  ing.
5 KODELIST can be used with KRUN.
6 .CODEE
```

Run the follwing command in the terminal:

```
cat file.kdoc | krun -t
```

After that, the file README.md will be created in the current directory and contain the specified code block.

11. Output Mode

Write the following code into a text file name "file.kdoc":

```
.CODES java DriverClass.java
public class DriverClass {

    public static void main(String[] args) {
        System.out.println("Hello World.");
    }
}
.CODEE
```

Run the follwing command in the terminal to get the output:

```
cat file.kdoc | krun -o
```

We will get the following standard output in the terminal:

```
public class DriverClass {

    public static void main(String[] args) {
        System.out.println("Hello World.");
    }
}

```

12. Hide A Code Segment

Hiding a code segment help when you want to use a code snippet to perform generative coding or just want to display the output. The actual feature is implemented in KDOC macro package rather than KRUN or KODELIST themselves.

```
.HIDE
.CODES python program.py
print(divmod(25, 10))
.CODEE
```

13. Get Statistic

Get statistic is one of the useful option to keep track the document or making reports. The output of the statistic option is formatted in yaml which is readable for human and also easy to parser via standard yaml parser in many language. The output of the statistic will include the statistic the default languages (included the ones setup by LANGUAGE request) and each session informations. The session are arranged top down according to their execution order. Write the following code into a text file name "file.kdoc":

```
.CODES python program.py
import sys

for arg in sys.argv:
    print(arg)
.CODEE
```

Run the follwing command in the terminal to get the statistic:

```
cat file.kdoc | krun -s
```

We will get the following standard output in the terminal:

```
--- # default-language
None:
    default-processor:  {processor:'python', arguments: []}
python:
    default-processor:  {processor:'python', arguments: []}
ipython:
    default-processor:  {processor:'ipython', arguments: []}
bash:
    default-processor:  {processor:'bash', arguments: []}
zsh:
    default-processor:  {processor:'zsh', arguments: []}
cmd:
    default-processor:  {processor:'cmd', arguments: []}
awk:
    default-processor:  {processor:'awk', arguments: []}
perl:
    default-processor:  {processor:'perl', arguments: []}
R:
    default-processor:  {processor:'R', arguments: []}
kdoc:
    default-processor:  {processor:'krun', arguments: ['-r']}
C:
    default-processor:  {processor:'tcc', arguments: ['FILE_NAME']}
java:
    default-processor:  {processor:'javac', arguments: ['FILE_NAME']}

yacc:
    default-processor:  {processor:'yacc', arguments: ['FILE_NAME']}
lex:
    default-processor:  {processor:'lex', arguments: ['FILE_NAME']}

--- # session
python@program.py:
    line count:         4
    segments:           [[2, 6]]
    run at:             none
    evaluate at:        none
    tofile :            none
    order at:           none
    capture keyword:    none
    custom-processor:   none

```

14. Add New Language to be Interpreted

Request `LANGUAGE` will add a new interpreter/compiler for the current file.

```
.LANGUAGE perl "/usr/bin/perl"
```

Besides, you can also use the special keyword `FILE_NAME` in the arguments part of the LANGUAGE, this will replace by the name of the session when running the interpreter/compiler.

```
.LANGUAGE perl "/usr/bin/perl" "FILE_NAME"
```

15. Set Up Custom Processor For Session

Comparing to `LANGUAGE` request, `PROCESSOR` will only affect the specified session rather than every session. `PROCESSOR` will only work if the language is defined.

```
1 .PROCESSOR python program.py python2
2 .CODES python program.py
3 print "Hello World"
4 .CODEE
5 .RUN python program.py
```

```
| Hello World
|
```

16. Working With Specific Session Using ID

Write the following code into a text file name "file.kdoc":

```
1 .PP
2 Hello World in Python scripting:
3 .CODES python program.py
4 print("Hello World.")
5 .CODEE
6 .PP
7 Hello World in Bash scripting:
8 .CODES bash program.sh
9 echo -e "Hello World."
10 .CODEE
```

To output only session with language "python" and name "UseIDExample.py", run the following command in the terminal:

```
1 cat % | krun -id python@program.py -o
```

We will get the following standard output:

```
| print("Hello World.")
| echo -e "Hello World."
|
```

17. Check dependency, file existence

krun engine also contains a `-check` option allowing users to keep track of the avaibility of different interpreters/processors on the user machine.

Write the following code snippet in the terminal:

```
1 echo "
2 .TOFILE python Example.py
3 .CODES python Example.py
4 print(\"Hello World\")
5 .CODEE
6 " | krun -c
```

Depending user machine, user may find different result from this check option. The Example.py file is clearly does not exist on my current directory, therefore, I can safely use tofile mode to write the code of the session to Example.py file.

Interpreter for cmd (MSDOS) language doesn't exist on my machine, so I shouldn't run any code snippets using the language.

```
| --- # check file existence
```

```
python@Example.py:
    Example.py:         NO
--- # check default-interpreter available
    python:             YES
    python:             YES
    ipython:            YES
    bash:               YES
    zsh:                YES
    cmd:                NO
    awk:                YES
    perl:               YES
    R:                  NO
    krun:               YES
    tcc:                YES
    javac:              YES
    yacc:               YES
    lex:                YES
```

18. Using getstdin Mode

This is the special built-in keyword in KRUN. This is another way if user want to get standard input directly from KRUN rather than standard input from a code snippet. This should only be used when using file as a filter.

```
1 mkdir getstdinExampleTemp
2 echo -e ".CODES\nprint(\"\"\":::STDIN@STDIN@STDIN:::\"\"\")\n.CODEE\n
  .RUN" > getstdinExampleTemp/getstdinExample.kdoc
3 echo Hello | krun -g getstdinExampleTemp/getstdinExample.kdoc -r
4 rm -R getstdinExampleTemp
```

We will get the following result:

```
Hello
```

19. Use KRUN for Metaprogramming

Metaprogramming technique will help users a lot in automate report, assignment, research paper, or development. By using simple text replacement mechanism, KRUN allows user to perform metaprogramming. Users can capture output of one program and put it in anywhere in the document or in a variable of a python program which then process the string further. This feature provided by KRUN, allows users to use R, Python, Perl, and languages together easily and predictable inside a single file. Metaprogramming also enables generative coding technique. Instead of writing a block of text by hand, users can write a code snippets and generate a code for the document, thus, partially automate, parameterize, and increase consistency.

Org-mode metaprogramming feature allows user to pass a data to a function in more direct way while KRUN is designed to perform text replacement like a macro. Each language has their own strength, KRUN intents to makes the code sessions more predictable when performing metaprogramming.

20. Use KRUN for Literate Programming

Literate Programming was introduced and popular by Donald Knuth. Literate Programming was the first feature to be implemented in KRUN. Literate Programming focuses more on explaining the code logic to a human being and embedding code snippets along the way. This technique requires 2 processors: a processor for collecting code snippets for execution and another processor for typesetting to produce quality paper. KRUN and

KODELIST (as a preprocessor for Troff formatting engine) are designed to take on these task.

KRUN engine has a variety of modes for users to extract the code session or multiple code sessions. Output mode would extract all code of all session to standard output and if invoke with id option, then only the select session will output to standard oupout. Another mode to use for literate programming is tofile mode. By specifying the session and optional output filename, running the tofile mode, KRUN will write all of specified sessions to specified files. This can be convenient for people to manage configuration files, scripts, or data.

21. Use KRUN for Reproducible Research

Reproducible Research technique can be a handy technique data scientists, programmers or researchers in general. This tecnique enables displaying the process of getting the result in the most organized manner. This user manual example output are all generated by krun engine. Anybody with have the right dependency and setup can easily recompile this document again to look exactly the same.

22. More Example

22.1. Use for Homework

This is an another demonstration using KRUN to enable generative coding techniqne to automate the document writing. Functions to produce kdoc and EQN code. When processing further using EQN preprocessor, it will produce Troff code for mathematical expression. This technique is very convient for student working on school assignment which has a lot of repetitive procedure or requires consistency.

```
.HIDE
.CODES python mathHW1.py pass123
def formatEuclid(a, b):
    i = 0
    while True:
        i += 1
        if b == 0:
            return (i - 1)
        quotient, remainder = divmod(a, b)
        print(".EQ")
        if i == 1:
            print('"' + str(i) + ') " ' + str(a) + ' = ' + str(b) + '
('  + str( quotient) + ") + " + str(remainder))
        else:
            print('"' + str(i) + ') " ' + str(a) + ' = ' + str(b) + '
('  + str( quotient) + ") + " + str(remainder))
        print(".EN")
        a = b
        b = remainder

n = formatEuclid(45, 37)
.CODEE pass123
.EVALUATE python mathHW1.py
.CAPTURE python mathHW1.py mykey1
.PP
Euclid Algorithm:
:::python@mathHW1.py@mykey1:::
```

The code above when running: `cat file.kdoc | krun -e` will produce the following output to the standard output.

```
| .HIDE
| .CODES python mathHW1.py pass123
```

```
def formatEuclid(a, b):
    i = 0
    while True:
        i += 1
        if b == 0:
            return (i - 1)
        quotient, remainder = divmod(a, b)
        print(".EQ")
        if i == 1:
            print('"' + str(i) + ') " ' + str(a) + ' = ' + str(b) + '
(' + str( quotient) + ") + " + str(remainder))
        else:
            print('"' + str(i) + ') " ' + str(a) + ' = ' + str(b) + '
(' + str( quotient) + ") + " + str(remainder))
        print(".EN")
        a = b
        b = remainder

n = formatEuclid(45, 37)
.CODEE pass123
.EVALUATE python mathHW1.py
.CAPTURE python mathHW1.py mykey1
.PP
Euclid Algorithm:
.EQ
"1) " 45 = 37(1) + 8
.EN
.EQ
"2) " 37 = 8(4) + 5
.EN
.EQ
"3) " 8 = 5(1) + 3
.EN
.EQ
"4) " 5 = 3(1) + 2
.EN
.EQ
"5) " 3 = 2(1) + 1
.EN
.EQ
"6) " 2 = 1(2) + 0
.EN
```

23. Some Observation

KRUN can be an important postprocessor for many other preprocessors. KRUN can also be modified as a module in Python to be used as part of another Python program.

Due to KRUN implementation in Python language, the boot time of Python interpreter can be a slow if the users have a lot of krun invokes in the document. KRUN implementation in C would a better choice if people need absolute speed for constant processing and future proof.

Developers or tinkers can also edit the source code slightly to make it work with LaTeX, MS, MOM, or Markdown.

KRUN does not bind to any development environment, therefore, users from any text editor can be almost as efficiency when switching to different ones.

It's also very easy (possibly even more enjoyable) to write kdoc document without any syntax highlighting.

Due to nature of kdoc as a textual, structural file format, users can collaborate with other people productively without a complicated setup and use standard version control tool like git to manage.

Users should combine the KRUN system with other Unix utilites for further text processing or analytic of data or codes.

24. About

This document is produced with the commands:

```
1 cat file.kdoc | krun -e | kodelist | soelim | klean | klush | pic | t
  bl | eqn | troff -kdoc -x | dpost | ps2pdf - - > file.pdf
```

This document using kodelist, klean, klush, krun, kdoc macro package of Kdocutils; modified version pic, tbl, eqn of Plan9; Troff of Heirllom Troff; and ps2pdf of ImageMagick.

All the example output are all generated evaluate mode of KRUN.

Author of KRUN and this paper: Khang Bao.

KRUN version: 0.4