Project 001 – Writing a Lexical Analysis Engine

By Tim Henderson (tadh@google.com)

What?

In this project you will be writing the core of a lexical analysis engine. Specifically you will implement the "ThompsonVM" algorithm adapted for the lexical analysis problem. (see Russ Cox's Article as well as the posted slides)

Lexical Analysis Requirements

The requirements for a lexical analysis engine are:

1. It must match prefixes of the input
2. It must match multiple patterns at once
3. It must pick the pattern which matches the longest prefix
4. It must break ties by picking the pattern which appears earliest in the user supplied list of patterns. Ie. break ties by user priority.

As a reminder the adaptions needed to a standard regular expression engine for lexical analysis are:

1. Keep going until no new states are generated.
2. Track the matches found.
3. Return the highest priority match for those matches of the longest prefix.
4. After state exhaustion, reset the automaton and find the next match.
5. After all input consumed ensure it was all matched. Otherwise return a failure.

Your Task

Your task will be to take input and a precompiled NFA program and then emit the tokens. Here is an example:

The NFA Program. Each MATCH instruction represents a pattern. This program matches three patterns: `"a"`, `"a+"`, and `"( |\t|\n)+"`. (notes on the format of the NFA programs are below).

```
0 SPLIT 1 3
1 CHAR 97 97
2 MATCH
3 SPLIT 4 9
4 CHAR 97 97
5 SPLIT 6 8
6 CHAR 97 97
7 JMP 5
8 MATCH
9 SPLIT 10 12
10 CHAR 32 32
11 JMP 16
12 SPLIT 13 15
13 CHAR 9 9
14 JMP 16
15 CHAR 10 10
16 SPLIT 17 25
17 SPLIT 18 20
18 CHAR 32 32
19 JMP 24
20 SPLIT 21 23
21 CHAR 9 9
22 JMP 24
23 CHAR 10 10
24 JMP 16
25 MATCH
```

The input

```
'a aa a aaaa a aaaa a'
```

To invoke your program, which in this example is `pr01`, the following command will be run:

```
cat input.txt | ./pr01 nfa-program.ns
```

Your program will output the instruction index of the MATCH `: "lexeme"` to the STANDARD OUT.

```
2:"a"
25:" "
8:"aa"
25:" "
2:"a"
25:" "
8:"aaaa"
25:" "
2:"a"
25:" "
8:"aaaa"
25:" "
2:"a"
```

Example inputs, outputs, implementation, and "student-grader"

To aid you in your quest to complete this programming project an example implementation of the assignment has been provided in compiled, binary form. Find it in `pr01/bin/pr01`.

To generate new NFA programs from a list of regular expressions (for testing) you can use `pr01/bin/lexc` which takes a list of regexs and produces an NFA program.

Your program should be able to process all of the test cases in the examples directory (`pr01/examples`). Each example `N` has:

1. An input `N-ex-input.txt`
2. A NFA program `N-ex-program.ns`
3. A `sh` command for running the example `N-ex-cmd.sh`
4. Output `N-ex-output.txt`
5. (Optionally) and error file `N-ex-error.txt`. The error will only exist of the example is a *negative* example meant to cause the lexical analyzer to fail in some way.

Finally, a program `bin/student-grader` has been provided. This program will automatically run your program against a set of test cases and produce a report on how many you passed. To run this program

```
$ ./bin/student-grader <PATH_TO_YOUR_EXECUTABLE>
```

NFA Program Format

Correct NFA Programs will all be in the following format. It is possible that *incorrectly* formatted programs will be fed to your program. Please use defensive program practices to detect these errors and print a helpful message to the user. A stack trace, core dump, or seg fault is not a helpful message.

Each line will have the following components:

```
INSTRUCTION-INDEX OP-CODE [OPERAND-A [OPERAND-B]]
```

For example

```
0 SPLIT 1 3
```

The fields will always be delimited by a space `" "`. The lines will always be delimited using unix line endings: `"\n"`.

The Op Codes

1. `CHAR` takes two operands: the start of the character range, the end of the character range (inclusive). The operands are given as the ascii decimal encoding of the character.

   ex. `1 CHAR 97 97` Matches `a`

   ex. `1 CHAR 97 98` Matches `a` or `b`

In this way, the CHAR operand could really be called CHARRANGE. But, CHAR is shorter and having a single operand CHAR operator is redundant.

2. `JMP` takes one operand: the instruction index of the jump target.

   ```
   7 JMP 5
   ```

3. `SPLIT` takes two operands: the instruction indices of the jump targets.

   ```
   5 SPLIT 6 8
   ```

4. `MATCH` takes no operands.

Requirements for the Command Line Interface of your program

Your program must read the input to tokenize on the STANDARD IN. It must output its output to the STANDARD OUT (in the appropriate format described above). It must take one command line argument, a path to the file containing the NFA program.

In general, you should be able to substitute your program's name in to the example `N-ex-cmd.sh` scripts and have them continue to function.

Your program must use the appropriate exit code

When your program exits it should use the appropriate exit code. If the program succeeds (tokenize all of the input properly) it *should exit with status 0*.

If your program fails (does not tokenize all the input properly) it *should exit with status 1 or more*.

This is important so that our automatic testing harness can verify your programs behavior.

Exiting from python

```python
#!/usr/bin/env python2

import sys

# main should return the integer status code
def main(args):
    if len(args) <= 0:
        print >>sys.stderr, "expected an argument got none"
        return 1
    print "my args are: ", args
    return 0

if __name__ == '__main__':
    # sys.exit sets the proper status code
    sys.exit(main(sys.argv[1:]))
```

Exiting from java

```java
package techx.sopl.pr01;

import java.util.Arrays;

public class LexCore {

    public static void main(String[] args) {
        if (args.length <= 0) {
            System.err.println("expected an argument got none");
            // System.exit sets the status code to > 0 to indicate failure
            System.exit(1);
        }
        System.out.println(
          String.format(
```

```
            "my args are: %s",
            String.join(" ", Arrays.asList(args)));
        // System.exit sets the status code to 0 to indicate success
        System.exit(0);
    }

}
```

Exiting from go

```
package main

import (
    "os"
    "fmt"
)

// run runs the program it returns the proper exit code and consumes the
// commandline arguments
func run(args []string) int {
    if len(args) <= 0 {
        fmt.Fprintln(os.Stderr, "expected an argument got none")
        return 1
    }
    fmt.Printf("my args are: %v", strings.Join(args, " "))
    return 0
}

// main is the entry point for the program
func main() {
    os.Exit(os.Args[1:])
}
```

What you will turn in

1. The source code for your implementation of NFA simulation via the "virtual machine" method. Also, any ancillary code needed (parsing code for NFA programs).
2. A README file that explains
    - How to compile your program
    - How run any associated tests you have written
    - How to run your program
    - Where the code is, and how to read it.
3. A `pr01` "command" which must conform the command line interface described above and demonstrated in the examples.
    - C/C++/Go/Rust/D simply name your binary `pr01`
    - Java, write a simple "shell script" wrapper that invokes the `java` command and passes any arguments to your program. (example below)
    - Python, write a top level `pr01` script, make it executable that implements the required interface. DO NOT PUT YOUR ENTIRE PROGRAM IN THIS SCRIPT – it should just be an entry point.
    - etc...
4. (Optional) Unit Tests. (10% extra credit)
5. (Optional) Use a build system. (10% extra credit)
    - For Java: gradle
    - For C/C++: make
    - For Python: virtualenv + pip
    - For Go: congrats! It is part of the language build your code with the `go build` command.
    - For Rust: use cargo. Don't invoke rustc directly.
    - etc...

Note: Using *only* Make for things other than C/C++ is not encouraged. However, it is fine to use it as part of your build setup. Just explain what you did in the README.

Ideally, I should be able to `cd` into your project directory, run 1 command, and have a `pr01` "program" (it can be a shell script wrapper) that implements the required interface.

Example Skeleton Programs

I have provided two skeletons that you may use to get started. There is one for python and one for java. See the associated readme files for either. I do not have the bandwidth to provide skeletons for other languages, sorry.

Example Shell Script Wrappers

Java

For Java, the easiest way to do this is to use gradle. I will also show you a manual way to accomplish this. Let's do the gradle setup first. Go to https://gradle.org and install gradle. (Already done on codio your welcome. This example uses gradle 5.1.1: codio may have a different version, that's ok.)

Now do this:

```
$ gradle init --type java-application

Select build script DSL:
  1: groovy
  2: kotlin
Enter selection (default: groovy) [1..2] 1

Select test framework:
  1: junit
  2: testng
  3: spock
Enter selection (default: junit) [1..3] 1

Project name (default: java): pr01
Source package (default: pr01): techx.sopl.pr01
```

Now you are setup to complete this assignment the EASY way.

Compile your program

```
$ ./gradlew classes
```

Test your program

```
$ ./gradlew test
```

hmm... The console output seems kinda, sparse. Here is the fix: https://stackoverflow.com/a/46533151. Edit your build.gradle and add the following line to the `plugins` section:

```
// test logger
id "com.adarshr.test-logger" version "1.6.0"
```

Now force rerun

```
 $ rm -rf build
 $ ./gradlew test

 > Task :test

 techx.sopl.pr01.AppTest

   Test testAppHasAGreeting PASSED

 SUCCESS: Executed 1 tests in 632ms
```

```
BUILD SUCCESSFUL in 1s
3 actionable tasks: 3 executed
```

That is better.

Now let's build that shell script:

```
$ ./gradlew installDist
```

The script is located at `./build/install/pr01/bin/pr01`

Let's run it:

```
$ ./build/install/pr01/bin/pr01
Hello world.
```

Huzzah!

Java The Hard Way

To get a java program to run you need two things

1. A java runtime, the `java` command.
2. A "classpath" that says where to get all your classes.

Gradle nicely puts that classpath together for you in a reliable way that lets you use third party libraries easily. Now you are doing this part by yourself. You will need to write script that looks kinda like this:

```
#!/usr/bin/env bash

java -classpath /absolute/path/to/your/classes:any/other/jars techx.sopl.pr01.MainClass
```

good luck!

Python

For python this is pretty easy. You will want to setup things in the following way.

1. Create a `activate.sh` script that sets the PYTHONPATH
2. Add a `bin` folder that contains your "executable" command
3. Add your implementation to the dir pointed at by the PYTHONPATH

Setup

(Assumes you are using virtualenv)

```
$ mkdir pr01
$ cd pr01
$ virtualenv --no-site-packages env
$ cat >activate.sh <<EOF
source env/bin/activate
export PATH=\$(pwd)/bin:\$PATH
export PYTHONPATH=\$(pwd)/src:\$PYTHONPATH
EOF
$ mkdir bin
$ mkdir src
```

Activate your setup

You have now made your setup to use it source the activate scripte

```
$ source activate.sh
```

You should do this every time you want to work on your program. It sets the appropriate environment variables. You only need to do this one time per shell.

Example

```
$ source activate.sh
(env) $
```

The `(env)` lets you know you activated your environment.

Directory structure

```
$ tree -I env --charset ANSI
.
|-- activate.sh
|-- bin
`-- src
```

You should put your python code in the src directory.

Final example with script

Layout

```
$ tree -I env --charset ANSI
.
|-- activate.sh
|-- bin
|   `-- pr01
`-- src
    `-- hello.py
```

Contents of `bin/pr01`

```
$ cat bin/pr01
#!/usr/bin/env python

import sys

import hello

def main(args):
    print hello.greeting()
    return 0

if __name__ == '__main__':
    sys.exit(main(sys.argv[1:]))
```

Make sure you make `bin/pr01` executable

```
$ chmod +x bin/pr01
```

Contents of `src/hello.py`

```
$ cat src/hello.py
#!/usr/bin/env python2

def greeting():
    return "hello world"
```