Secure and Dependable Systems
Jacobs University Bremen
Dr. Florian Rabe

Homework 2
given: 2017-02-08

Homework 2

You have to submit your solutions as announced in the lecture.
Unless mentioned otherwise, all problems are due 2017-03-07, before the lecture.
There will be no deadline extensions unless mentioned otherwise in the lecture.

Problem 2.1 *Shellshock*

Points: 2+2+1+1+1+1+1+1

Consider the shellshock example from the lecture notes. In this problem, we implement a minimal shell that could exhibit the fault but will not because we design it well.

You can use any programming language. However, it is best to use a programming language that supports good system design. SML will work well; Java or C++ are OK. I personally recommend Scala.

1. Implement datatypes for the following grammar, which represents the commands our shell can handle.

commands			
$COMM$	::=	$val\ NAME = EXP$	
	|	$fun\ NAME(NAME)\{COMM\}$	function definition
	|	$run\ EXPR$	shell call
	|	$NAME(EXPR)$	function call
	|	$COMM; COMM$	command sequence
expressions			
$EXPR$	::=	$NAME$	variable
	|	" (\\ | \" | [^\"])* "	string
names			
$NAME$	::=	alphanumeric string	

   where red color indicates BNF meta-symbols.

   You need one datatype per non-terminal with one constructor per production. Let $Comm$ and $Expr$ be the types for $COMM$ and $EXPR$.

   This language is pretty boring in order to be simple. Feel free to add, e.g., if commands, number expressions, functions with return values etc. However, watch out that writing the parser will become increasingly work-intensive.

2. Implement a parser for your data type. It should be of the form

```
fun parseCommand(command : String) : Comm =
   ...
fun parseExpr(expr : String) : Expr =
   ...
```

3. Implement an interpreter for your data type. It could be of the form

```
fun interpret(context : List[Def], command : Comm) : List[Def] =
   ...
fun evaluate(context : List[Def], expr : Expr) : string =
   ...
```

   where Def is the type of those commands that are definitions, i.e., define a value ($val \ldots$) or a function ($fun \ldots$).

   It should interpret commands as follows:

   - for value or function definition, return itself
   - for $run\ e$: evaluate e to a string and then run it in the shell
   - $f(e)$: evaluate e to v, retrieve the function definition $fun\ f(x)C$ from the context, interpret C with an additional $val\ x = v$ in the context

- $C; D$: interpret C, then interpret D with the definitions returned by C added to the context

and evaluate expressions as follows

- n: retrieve $val\ n = v$ from the context and evaluate v
- "s": return the string s with the escapes removed

4. Implement main function that takes a string s, calls $c = parseComm(s)$, then calls $interpret(Nil, c)$.

   Optionally, you can also

   - read an entire file and parse+interpret every line in it
   - read commands from standard input, at which point you have an actual shell

   Those steps are not required but recommended because they help with testing.

5. Now for the faulty functionality of bash, modify your program as follows:
   - At the beginning, try to parse every available environment variable that starts with fun into a $Comm$.
   - Whenever that succeeds, interpret the resulting commands, which returns $defs : List[Def]$. (*)
   - Now call interpret on the input with $defs$ instead of Nil as the initial context.

6. Activate the fault by showing that data in environment variables may lead to the execution of arbitrary shell commands.

7. Isolate the fault (This is how it was "fixed" in bash.) by considering only environment variables whose name begins with a certain prefix, e.g., $SHELL\_FUNC$.

8. Remove the fault by making sure that data is never executed. To do so, change (*) so that it never calls $interpret$.

For questions 1-5, submit a single program. For questions 7-8, submit a single program. Use comments as needed to understand which part solves which question.

For question 6, submit a screenshot from a normal shell session that demonstrates the failure of your shell.