National Research Tomsk State University

# Laboratory 2 – Static Analysis: Hands on

**Passive Testing Techniques for Communication Protocols**

February 19, 2016

The goal of this laboratory is to introduce you to the magic world of static code analysis. In this laboratory, we will be only using the principles learned and implementations of your own. Your implementations will be using widely-adopted tools, lexer/parser generators, Lex and bison You will be making your very first static analyzers. All the methods can be applied to larger languages, but given our restrictions in time we will not.

Most of the times, different Operating Systems (OS) or tools within them can provide similar results, for instance, Bison can be compiled for Windows architectures. As stated before in the course introductory lecture, you are free to use the OS or tool of your choice. However, instructions, hints, commands, libraries, and etc., will be provided only for the a standard Linux environment. Furthermore, you should in any case provide a Makefile to compile your tools in the submission. As usual, I will be able to provide more assistance in the stated lab environment. In short, you're on your own if you decide to choose your own OS or tools, sorry :)

**Submissions:** Submissions should be done in a single compressed file (zip, or gz, including flavors of .tar.gz, and .tgz) of a directory containing all the necessary files. Regardless of how jealous you are about sharing your code, always include the code of your assignations. Include all your documents, executable files, and a Makefile to compile them. If you feel the need of putting a README.txt file, to clarify what are the contents of your submission feel free to do so. All submissions should be done via The Moodle Cloud Platform, https://pttcc.moodlecloud.com/ Make sure you are able to login and that you assign the course to yourself.

## 1 Getting familiar with Lex and Bison

For this purpose you are given a compressed archive placed on the MoodleCloud and the local FTP, SPA.tgz.

For the first assignation, we will print in a hierachical manner the following language: $S \mapsto SS|(S)|\{S\}|[S]|\{\}|[]|()$, the language of well formed items (parenthesis, bracket, and curly brackets), further referred as the parenthesis (a bit more) language.

Decompress the contents of the tarball. For this first exercise we will be using the directory named “1”. The contents of the the 1 are as follows:

- Makefile: this is the file that allows you to easily compile your source files. Do not modify it unless you know what you’re doing (still, take a look at it, you might modify a copy of it for the part 3)
- paren.c: This file contains the implementation of auxiliary functions to build the AST for the parenthesis language.
- paren.h: This file contains the header functions of auxiliary functions to build the AST for the parenthesis language. Read from this file the available functions you have to create the AST (mainly createParen)
- spa: parenthesis language hierarchical printing (based on the AST) executable compiled for the VM
- spa.c: The main parenthesis language hierarchical printing implementation based on the AST. This file contains functions to obtain the AST based on the parser start symbol return object, to read the file, which will be the input to parse, and to print hierarchically the “parenthesis” objects. This file written completely in C
- spa.cpp: Same as spa.c but for C++ (as usual, I’m able to provide better help for C implementations)
- lexer.l: The Lex lexer for the parenthesis language, this is where the original string gets tokenized with the corresponding regular expressions (more information below)
- parser.y: The Bison parser for the parenthesis language, this is where the grammar rules are stated, and the associated objects of the AST for each grammar rule (more information below)
- *.txt: test inputs containing some examples of the parenthesis language

It might look scary and big, but, I coded all for you just to use the functions stated above and implement the functionality. The instructions how to run and to compile:

- To compile the hierarchical parenthesis printing with the c file run make c
- To compile the hierarchical parenthesis printing with the c file run make cpp
- To generate the lexer only, run `lex lexer.l`
- To generate the parser only, run `bison parser.y`
- To clean the generated parser, lexer and executables run make clean
- To execute the hierarchical parenthesis printing run ./spa <filename>, where <filename> is the name of a file (e.g., test.txt0

At the moment, the executable file prints the contents in a simple hierarchical manner, prints the children of an element with one indented tab recursively. For instance, for the input "{[({})]}([])" in the file tt.txt, it prints out:

```
[Eva-01@localhost 1]$ ./spa tt.txt
{}
        []
                ()
                        {}
()
        []
```

A hierarchical view on the elements. For this part of the laboratory, you should get familiar on how Lex/Bison works, you are asked to extend the functionality of the software by accepting the balanced less than (<), and greater than (>) symbols. **Lexer**: Start by looking at lexer.l file. The Lex files have a very simple, it is divided into 3 sections, definitions, rules, and user code. These sections separated by %%.

The first section usually contains the headers you want to include into your parser, the options, and the definitions of regular expressions.

The rule section contains mostly actions what to do in case a regular expression is met. As you can see in our file, whenever we encounter the proper element, we return the proper token, e,g,:

```
27 {LPAREN} {return TOKEN_LPAREN;}
```

Please note, that the name of the returned tokens you can choose. Also note that, the rules that are conflicting (let's say you define$a = [0 - 9]$, and $b = 0$, the priority is taken by the first rule it appears defined.

The final section is reserved for user code, this is copied directly into the lexer and not checked. Further, the compiler can check this code. It can be useful for some pre-processing rules replacing comments into white spaces. For this lab you will not have to define any user code in your Lex files, but you will look at one example (later).

Now, that you know about lex, probably you know that there are two places where you need to add something to extend the language. The RE definition of the new symbols and the associated actions. Do so.

One remark is that the Kleene-star($^*$), is denoted as plain$*$, in the Lex regular expressions (very common practice), also the $+$ for the short notation $^+$. $*$ or $+$ are self-represented characters.

The lexer alone can be generated by the command `lex lexer.l`, this will generate the lexer.h and lexer.c files. You can modify this option in the file. Note that lex is a lexer generator for C, that later on we will use (we describe with REs and recognize with a lexer). You don't need to do this alone. The make c or make cpp will do this for you.

**AST**: Before moving to the grammar, we are going to modify the data structure for the AST so that the grammar will use it. Take a look at paren.h, there are 3 main data structures that you can check. The first is asttype, which contains the list of possible data structures associated to the AST (for this particular case, we have one);

the second, the data structure for the type of “parenthesis”, ptype; and finally, the paren data structure that represents a parenthesis (it uses the ptype structure to differentiate between parenthesis, brackets and curly brackets). As you can see this AST is quite simple and it is composed of a single type of object.

At this point you should recognize that you just need to add a new type of “parenthesis”. Do so.

Take a look at paren.c, since the implementation is so good (thank you), you can see that you do not need to modify the actual functions. But, it will be useful for you to know what they are actually doing.

**Parser**: If you are understanding by this point it is magnificent, the rest will be without a doubt simple. If you are not, do not worry, I am sure the parser will help you clarify everything.

The file parser.y (extension y due to the YACC parser generator), is composed of 4 sections, prologue, declarations, rules, and epilogue.

The prologue contains macro definitions and function declarations that are later used in the rules. The prologue is enclosed between {% and %}.

The declarations are specific instructions for handling Bison grammars. Perhaps the most important to understand are the %union, %token, and %type. In short, union defines all the possible types of objects that the grammar is going to use, according to the AST and to the symbols returned by the lexer. For instance, for our grammar we have a paren type pointer, called parenobj. Wherever we perform operations with paren, we can use the name parenoj. %token defines the tokens we use (should match with the lexer). Finally, %type defines the objects used for the non-terminals of the grammar. Please, pay special attention to %union.

The grammar rules are the core and essence of a parser. We describe the non-regular language in the Bakus-Naur Form (BNF), in our regular notation the production rule arrow (↦) is replaced by a colon (:), and each production rule ends with a semicolon.

The production rules can be quite simple, they describe with the tokens and non-terminals the grammar as you know. The details that you need to pay attention to are the actions, whenever a matching rule is found. Generally speaking these rules will contain the return types for the object mappings. For non-terminal to return something, it should be assigned to the variable \$\$. You can refer to the value of each element in the production rule by the variables \$1, \$2, \$3,,. . . , etc. However, for a clearer syntax, you can enclose in brackets the name that you want to use, for example:

```
            : P2[le] P[ri] { $$ = createParen(NONE, $le, $ri);}
```

Could’ve been written as:

```
            : P2 P { $$ = createParen(NONE, $1, $2);}
```

In here is where we use the definitions of our data structures. Note that since Bison generates Left-Right parsers (we didn’t have time to cover this, but trust me (:), it is very important the order of the elements as they appear. But, all exercises and explanations while deriving parse trees in the class were done this way, so you should be familiar with it. To better understand this, if exchanging the previous rule as:

```
            : P[le] P2[ri] { $$ = createParen(NONE, $le, $ri);}
```

The first elements that appear in the source will appear at the right hand of the structures, thus having to perform a right-first depth search for visiting the AST.

As you can see in this file, you need only to create two token definitions, and two rules for the grammar. Do so.

As a last note, the non-terminal input is not necessary, but, it is a custom to start with such a symbol. The start symbol is considered the first non-terminal it appears on the grammar. For our case, as you can see, it is just a transition and it returns exactly the same object as P does.

The last section of Bison, the epilogue is similar to Lex user code. This code is directly copied into the parser.

Note that the parser alone can be generated by executing the command `bison parser.y`. This will generate the parser (similar to the lexer), with the filenames parser.h and parser.c.

**Static Analyzer** The file spa.c (or spa.cpp) contains functions to retrieve the AST from the parser (getAST). Generally speaking, you just need to change the root of the AST (start symbol) in that function for other grammars. For our case, the only thing that needs to be changed is the printAST function, to add one more type and display. Do so.

Check the rest of the functions, they might help you understand how everything works together.

Re-compile all (make c, lstinlinemake cpp, lstinlinemake clean; make c, etc.) and test new files with the added symbols.

Imagine that you need to add another group of symbols /\\, I hope you see how fast this can be added (do not add such symbols).

By this point, I hope you can see how fast it can be to modify grammars and parsing files when doing it with formal languages. Let's now move to a really fun part. Guaranteeing certain properties on the input, let's build our first static analyzer.

## 2 Your very first static analyzer

In this section you will create your first static analyzer for the parenthesis language. You can modify the spa.c (or spa.cpp) files to do so. We want to check the property that for any depth level, no more than d times the same symbol is used consecutively. That means that [[[[()]]]] violates the property, but, <<< (((< {} >))) >>> {} does not. An easy way to get you started.

By the end of this exercise you should feel confident while statically analyzing code for simple properties. Let's build a more realistic analyzer.

## 3 Static Analyzer for a Register Machine Language

The previous exercises of this lab were a warm-up for this section. Any static analyzer has probably started with a reduced grammar to prove the approach is valid. This is what we will do.

First, take a look at the ANSI-C grammar for YACC/Bison available in here. You can see how extensive the grammar is and how the author uses precedence binding for expressions and avoiding ambiguity (putting at lower level of production rules the strongest bindings). Since he uses such construction to describe a function call uses many production rules. Starting from the production rule of statement, what will be the derivation for a function call (e.g., malloc(10))? Write this down in a PDF (you can do this at the end of the lab if you wish).

Now that we have the motivation to use a reduced language, let's introduce the language that we will be using. This is a theoretical language that belongs to the family of the register machine languages. The language is described next.

The language consists of a list of statements (instructions), these statements can be of the following types: variable assignations, labels, goto, if, input and output. All statements are finished with a semicolon(;), in exception of a label.

Variable assignations take on the left hand side a variable name in the classical identifier format (ID = $[a-zA-Z\_]^{+}[0-9a-zA-Z\_]^{*}$), the assignation operand (:=), and on the right hand side an expression (potentially nested).

Expressions are based on integers (positive and negative), and variables. They operations can be adding (with the + sign), multiplying (with the $*$ sign), and comparing (with the = sign). If the comparison yields true the returned value is 1, and 0 if not (not important for you this aspect though). Expressions cannot be grouped / nested with parenthesis (you're welcome).

A possible assignation statement looks like this:

```
a := b + 18 * c;
```

Labels consist of an identifier followed by a colon (:), for example:

```
begin:
a := b + 18 * c;
```

goto statements consist of the goto keyword and one identifier. For example:

```
goto begin;
```

The meaning behind a goto is an unconditional jump to the instruction (statement) where the label is found.

input statements consist of the input keyword followed by an identifier. For example:

```
input c;
```

The meaning behind the input statement is to input the register machine an integer and store it in the variable described by the identifier.

The output statements are complementary to the input ones. For example:

```
input c;
output c;
```

Will "echo" back an integer.

Conditional branching (jumping) can be achieved with the if statements. They are formed by the keyword if followed by an expression, followed by the keyword goto, followed by a label. For example:

```
if n = 20 goto begin;
```

The meaning is conditional branching. It will jump to the instruction (statement) where the label is found if the expression evaluates to non-zero.

With this simple register language complex calculations can be made. Take the following example:

```
a := 0;
b := 1;
input n;
i := 1;
F := 0;
if n = 0 goto done;
if n = 1 goto one;
Fib: if i = n goto done;
        F := a + b;
        a := b;
        b := F;
        i := i + i;
        goto Fib;
one:
        F := 1
done:
        output F;
```

Guess what this will output? This example can be found in the folder named 3, with the filename fib.rl. Indeed the C equivalent of this register machine language C code is also included in fib_reg.c. Furthermore, a compiled executable file is found in fib under the same directory (compiled with gcc −o fib fib_reg .c). This just to demonstrate the power of such small language.

For this exercise, you have to make a static analyzer that checks that property that no variable is used (read, i.e., not as the target of an assignation) without first being assigned a value.

You have all the tools you need, here are some hints:

- Copy the files from 1, it will be simpler to start with a skeleton.
- The Makefile is the one that controls how to compile. If you add different names to your C/CPP/header files, modify the Makefile accordingly. There is a lot of

documentation on Makefiles on the Internet. It will be great for you to lean this. If not, you can always ask me how to add files/compile.

- Remember that %union needs to have all the different data types your grammar uses. If you use an int and name it value, you will refer to all int values like this. This is needed for your lexer, for instance to return a value in int expressions you will do it like this:

  ```
  {INT} { sscanf(yytext, "%d", &yylval->INT_VAR_IN_BISON);
          return TOKEN_INT; }
  ```

  Where INT_VAR_IN_BISON is the name of the int value in the union in Bison. The same applies to text values (but they are text values returned by default)

- Be careful to include the header files in both Lex and Bison files of your data structures

- Be specially careful to include the header files in your final static analyzer. And careful with the order they are included, they should be in the order of data structures, parser, lexer. (this is a word of wisdom and experience)

This might look like a huge task. But, if you successfully completed the previous parts you will see that this will go smooth. Start designing your grammar, regular expressions and data structures.

Good luck!

P.S. Actually building an interpreter for the register machine language with the objects of the AST is a straightforward task. Building an interpreter and converting it to an **abstract interpreter**, not calculating precise values, but, abstract symbols that help the static analysis is the principle of the widely adopted technique abstract interpretation.

P.S. 2. I hope you enjoy this lab! :)