

Lab 1
CMSC 445
Marmorstein
Spring 2009

The purpose of this lab is to gently introduce you to the wonderful world of flex and bison by implementing a very simple string processing language.

The language consists of four commands:

str1 CONCAT str2	concatenates strings str1 and str2
str1 PLUS str2	also concatenates strings str1 and str2
TRUNC str1 n	removes the last n characters of string str1 (or, if str1 is too small, returns the empty string)
TRIM str1 n	removes the first n characters of string str1 (again, if str1 is too small, we return the empty string)

The language has five kinds of tokens: the three keywords (CONCAT, TRUNC, and TRIM), string tokens, and integer tokens. The CONCAT operation is left associative.

Here are regular expressions for each of the five token types:

CONCAT	=	CONCAT PLUS
TRUNC	=	TRUNC
TRIM	=	TRIM
[a-zA-z]+	=	STRING
[0-9]+	=	INT

Here is a grammar for the language:

```
expression := expression CONCAT expression
             | TRUNC expression INT
             | TRIM expression INT
             | STRING
```

Yeah, it's that simple.

Step 1. Setting Up

Create a folder named "Lab 1". In that folder, we are going to create five files:

- StringHandler.h : will contain prototypes for three helper functions
- StringHandler.cc : will implement the three helper functions
- main.cc : will have the main function that kicks off parsing
- parser.y : will define the grammar as a Bison file
- scanner.l : will define a scanner as a Flex file

Step 2. Helper Functions

Create a file named "StringHandler.h" that declares three helper functions:

```
string* concat(string* str1, string* str2);
string* trunc(string* str1, int n);
string* trim(string* str1, int n);
```

We use pointers, because bison and flex are C programs that generates C code and don't really understand C++ objects that well. They understand pointers just fine, though, so we're good.

Now create a file named StringHandler.cc that implements these three functions. Make sure that in the trunc and trim functions you handle the case that the string is too small to trim or truncate. *You may find the C++ string class function "substr" helpful. You can find a guide to that function at <http://www.cppreference.com/cppstring/substr.html>*

Step 3. Writing a Bison Grammar

Create a file named "parser.y". At the top of that file add the following code:

```
%{
    #include "StringHandler.h"
    #include <iostream>
    #include <string>
    using namespace std;
    int yylex(void);
    int yyerror(const char* msg){
        cout << "Error: " << msg << endl;
    }
}%
```

The three include statements and the "using namespace" bit are just standard C++ headers.

The yylex declaration tells the parser that we are going to provide the scanner in a separate file. The yyerror function tells the parser what to do with error messages. Having these two declarations is very important -- your parser won't compile without them!

Because we have more than one type of variable (integers AND strings), we need to tell bison what possible types our variables can have. The best way to do this is with a "union" command, so add the following to your "parser.y" file:

```
%union {
    string* stringType;
    int numType;
}
```

To tell bison that the STRING and INT tokens have the appropriate types, add this code:

```
%token <stringType> STRING
%token <numType> INT
```

Now, add "%token" statements for the remaining tokens. They do not need to have type information. Remember to tell bison that the "CONCAT" token is left-associative.

We also need to tell bison the "return" types of each of the non-terminals in the grammar. The following code will accomplish this:

```
%type <stringType> start
%type <stringType> expression
```

Close the "DECLARATIONS" section with a "%%" line and add the following code to the grammar rules section:

```
start: expression { cout << $1 << endl; $$=$1;}
expression: expression CONCAT expression {$$=concat($1,$3);}
           | TRUNC expression INT {$$ = trunc($2, $3);}
           | TRIM expression INT {$$ = trim($2, $3);}
           | STRING {$$ = $1;}
```

Add a "%%" line to the bottom of the file, save your work and exit.

Step 3. Writing a scanner

Open up "scanner.l" and add the following code:

```
%{
    #include <stdlib.h>
    #include <string>
    using namespace std;
    #include "parser.tab.h"
}%
```

Make sure you do these in the right order. The "parser.tab.h" file is generated by bison from the "parser.y" file you provided. Since you use pointers to strings in that file, it is important that the compiler pull in the string class before you try to include "parser.tab.h"

Close the declarations section with a "%%" line and add regular expressions for each of the tokens. Instead of printing out the tokens, you should return the appropriate token identifier.

Here is code for the regular expression section of the scanner:

```
CONCAT      { return CONCAT;}
TRUNC       { return TRUNC;}
TRIM        { return TRIM;}
[a-zA-Z]+   { yylval.stringType = new string(yytext); return STRING;}
[0-9]+      { yylval.numType = atoi(yytext); return INT;}
```

Add a "%%" line, then save and quit.

Step 4. Writing a main function

Your "main.cc" file should look like this:

```
int yyparse(void);

int main(char* argv, int argc){
    yyparse();
}
```

Type in that code, save, and quit.

Step 5. Compiling your project

We need to tell bison to emit a "parser.tab.h" file that will define all the token identifiers. To do this, we use the "-d" flag:

```
bison -d parser.y
```

We can then generate a scanner:

```
flex scanner.l
```

And compile everything together:

```
g++ -g main.cc StringHandler.cc parser.tab.c lex.yy.c -ly -lfl -o MyParser
```

You can now test your file by running `./MyParser` and typing in the following input (followed by CTRL-D):

```
TRUNC hello 2 CONCAT ter CONCAT TRIM askelter 1
```

Your parser should skip out the text "helterskelter".