# CS1319 - Monsoon 2023 - Assignment 3

**Lexical and Syntactical Debuggers**

Aryika Mehrotra and Aaryan Nagpal

This is our document explaining our assignment. We have used GCC Version: `11.4.0`, on an Intel Processor.

## MakeFile

We first started with creating a MakeFile and making our pipeline ready as we code. Our code in MakeFile is:

```
build:
	bison 26_A3.y --defines=26_A3.tab.h -o 26_A3.tab.c
	flex -o lex.yy.c 26_A3.l
	gcc -o parser lex.yy.c 26_A3.tab.c 26_A3.c -lfl -Werror

build_2:
	bison 26_A3.y --defines=26_A3.tab.h -o 26_A3.tab.c
	flex -o lex.yy.c 26_A3.l
	gcc -o parser lex.yy.c 26_A3.tab.c 26_A3.c -ll -Werror

test:
	bison 26_A3.y --defines=26_A3.tab.h -o 26_A3.tab.c
	flex -o lex.yy.c 26_A3.l
	gcc -o parser lex.yy.c 26_A3.tab.c 26_A3.c -lfl -Werror
	./parser < 26_A3.nc > test.out

clean-test:
	rm -f lex.yy.c 26_A3.tab.c 26_A3.tab.h parser
	rm -f test.out

clean:
	rm -f lex.yy.c 26_A3.tab.c 26_A3.tab.h parser
```

Only specific relevant commands from our MakeFile are mentioned here since we had a lot of other commands as well to assist us.

Right now, all are empty files. We made `build` using `-lfl` and `build_2` using `-ll`.

For test cases, the `test` command parses the `26_A3.nc` andstores the output in `test.out`. There is another command in the file called `test-all`, which checks for the cases provided in the guide and `test-2` checks the other test cases we made including 2 for factorial and 2 error cases: `array_runtime_error, factorial, factorial_recursive, syntax_error`.

# Main C File

Our main C file includes the code:

```c
#include <stdio.h>
#include "26_A3.tab.h"

extern void yyerror(char *s);
extern int yyparse();

int main(){
    int parser = yyparse();
    return 0;
}
```

This works as an entry point for running our parser. We have included `"26_A3.tab.h` which we generated from our Bison specification.

# Parser File

We followed the rules given in the assignment. In the beginning we mentioned the necessary declarations and directives that had to be included. We specified that the lexer file (yylex()) is external and should be included to link with the lexer.

## Union

The *%union* section defines the types of attributes that the parser will associate with different tokens. In this case, two types are defined: `int val` and `char *name`. These attributes will be used to carry information associated with tokens during the parsing process.

```
    %union {
    int val;
    char *name;
};
```

## Start

The *%start* directive specifies the starting symbol for parsing. In this case, it is translation_unit, indicating that the parser will begin with the translation unit of the source code.

```
    %start translation_unit
```

## Keywords

Other than the keywords and variables, we also defined Punctuators having double characters as directly writing them was not being interpreted properly. These are also separately written in the lexer file.

```
%token ARROW // ->
%token LESSTHANEQ // <=
```

```
%token GREATERTHANEQ // >=
%token EQ // ==
%token NEQ // !=
%token LOGICALAND // &&
%token LOGICALOR // ||

%token <name> IDENTIFIER
%token <name> STRING_LITERAL
%token <name> CHARACTER_CONSTANT
%token <val>  INTEGER_CONSTANT
%%
```

### Rules

We used the rules given in the Assignment doc for outcomes of these particular tokens.

We made separate rules for all the `*_opt` rules as well. For example:

```
argument_expression_list_opt:
    argument_expression_list
    |
    ;
```

### End

In the end, we defined the `yyerror()` to catch errors as:

```
void yyerror(char *s) {
    printf("Error: %s on '%s'\n", s, yytext);
}
```

## Lexer File Changes

We used the same lexer file that was used in the previous assignment. The only changes we made were:

- Error in Character Sequence and String Sequence was fixed to include empty characters or strings. The following was written:

  ```
  CHAR_SEQUENCE       {C_CHAR}*
  S_CHAR_SEQUENCE     {S_CHAR}*
  ```

- Replacing `printf` statements for the rules to `return`. For example:

  ```
  {IDENTIFIER}            {return IDENTIFIER;}
  {CHARACTER_CONSTANT}    {return CHARACTER_CONSTANT;}
  {INTEGER_CONSTANT}      {return INTEGER_CONSTANT;}
  {STRING_LITERAL}        {return STRING_LITERAL;}
  ```

- Editing the Integer constant rule to the one mentioned in the guide:

```
NONZERO_DIGIT       [1-9]
ZERO_DIGIT          [0]
INTEGER_CONSTANT    {ZERO_DIGIT}|{NONZERO_DIGIT}{DIGIT}*
```

- Separately wrote rules for double-character punctuates (which are terminals in the Parser):

```
"&&"                    {return LOGICALAND;}
"||"                    {return LOGICALOR;}
"<="                    {return LESSTHANEQ;}
">="                    {return GREATERTHANEQ;}
"=="                    {return EQ;}
"!="                    {return NEQ;}
"->"                    {return ARROW;}
```

- Included `26_A3.tab.h` file:

```
%{
    #include "26_A3.tab.h"
%}
```

## NanoC File

To use as a test, we will use a nested function code to check our parser. It is as follows:

```
int multiply(int a, int b) {
    return a*b;
}

int divide(int a, int b) {
    if (b == 0) {
        return -1; // Division by zero error
    } else {
        return a/b;
    }
}

int main() {
    int x = 10;
    int y = 2;
    int z;

    if (x > 0) {
        if (y > 0) {
            z = multiply(x, y);
            if (z < 20) {
```

```
                printStr("Result is less than 20.\n");
            } else {
                printStr("Result is greater than or equal to 20.\n");
            }
        } else {
            z = divide(x, y);
            if (z == -1) {
                printStr("Division by zero error.\n");
            } else {
                printStr("Division result: ");
                printInt(z);
                printStr("\n");
            }
        }
    } else {
        printStr("x is not positive.\n");
    }

    return 0;
}

```