This report will describe the challenges we have faced and how we addressed them:

1. Reduce/reduce & shift/reduce conflicts

When we first write our BNF grammar, at first execute, there were lots of reduce/reduce and shift/reduce warnings. Reduce/reduce conflict occurs when there are two or more rules that apply to the same sequence of input. This usually indicates a serious error in the grammar. Also it makes the language ambiguous which is not we want. On the other hand, shift/reduce conflict is the classic problem with "C" style if statements. Hence we have solved them all one by one by converting grammar to the unambiguous completely.

2. Objects with multiple constructors

In our language, we have objects like Line, Rectangle, Oval and we have built-in functions like drawLine, drawRectangle and drawOval. Since each of them averagely has 6 constructor type, it made us trouble when we are writing BNF grammar for them. This is because, we were always confronted with reduce/reduce and shift/reduce conflicts. For example, we have 6 constructors with all different parameter counts or types as it is seen:

```
rectangle = Rectangle(loc, size);
rectangle = Rectangle(loc, size, strokeWidth, isRounded);
rectangle = Rectangle(loc, size, strokeWidth, isRounded, color, isFilled);
rectangle = Rectangle();
rectangle = Rectangle(strokeWidth, isRounded);
rectangle = Rectangle(strokeWidth, isRounded, color, isFilled);
```

Thus, it was hard to write their grammar without making our grammar ambiguous. Each of them has a long grammar rules but we succeeded to make them unambiguous at the end.

3. Use before definition errors

In our language, we have variables, we have functions and we have extensions which are defined and used by the user. However, variables, functions and extensions cannot be called before they are defined. If they try so, we should give errors.

3.1. Use before definition errors for variables

First, we created (defined): `set<string> var_symbols;` at the the parser.y file.

Then in each assignment of variable, we added this new string variable to the set by calling this:
```
{var_symbols.insert(string($1));}
```

When user calls any variable from anywhere, we checked if that variable is at the set<string> or not by calling this in each call:
```
{if(var_symbols.count(string($1))==0) yyerror("variable " +
string($1) + " is not defined");}
```

As it is seen, if variable is not found, it also calls yyerror method which is defined at parser.y file and it informs the user. It is such simple method:

```
void yyerror(string s)
{
	numoferr++;
	cout << "errorat line " << yylineno << ": " << s << endl;
}
```

Besides informing user about error, we also inform about at which line the error is by calling yylineno which comes from yacc and which is defined at the parser.y file:

```
extern int yylineno;
```

As it is seen at the method, it also increments the numoferr which is again defined at the parser.y file:

```
int numoferr;
```

3.2. Use before definition errors for functions

First:

```
set<string> defined_func_symbols;

set<string> used_func_symbols;
set<int> used_func_line_no;
```

We defined them at the parser.y file.

Then, in each function definition below main function, we added that function name to the defined_func_symbols.

On the other hand, in each function call at the main function, we added that function name to the used_func_symbols. Also, we added the line_no for that call to the used_func_line_no at parallel index with used_func_symbols.

Finally, in main part of the parser.y file we called a void method funcCallCheck(). This method compares strings in the defined_func_symbols with used_func_symbols. If there is any used function which is not defined at defined_func_symbols, it gives the error to the user with the line number. It says that this function is not defined. Also, it increments the `numoferr`.

3.3. Use before definition errors for extensions

Actually, this has completely same logic with functions.

First:

```
set<string> defined_extension_symbols;

set<string> used_extension_symbols;
set<int> used_extension_line_no;
```

We defined them at the parser.y file.

Then, in each extension definition below main function, we added that extension name to the defined_extension_symbols.

On the other hand, in each extension initialization at the main function, we added that extension name to the used_extension_symbols. Also, we added the line_no for that call to the used_extension_line_no at parallel index with used_extension_symbols.

Finally, in main part of the parser.y file we called a void method extensionCallCheck(). This method compares strings in the defined_extension_symbols with used_extension_symbols. If there is any used extension which is not defined at defined_extension_symbols, it gives the error to the user with the line number. It says that this extension is not defined. Also, it increments the `numoferr`.

4. Multiple errors catch with error keyword

We discovered the error keyword which is provided by yacc. When program could not go anywhere defined at the grammar, it goes to error and informs user about the error. Also, it increments the numoferr.

While we can define the error type by calling yyerror, we have an option to not define it. At this situation, the error will be simply called as "syntax error". Examples:

Pre-defined error:

At this example, if conditional at for loop is not valid, we catch it:

```
loop_cond:
    conditional |
    error {yyerror("conditional expected");}
```

Not-defined error:

At this example, it simply gives "syntax error" if it is not valid statement:

```
statement:
    expr SEMICOLON |
    for_stmt |
    while_stmt |
    do_while_stmt |
    if_stmt |
    error
```

5. Program with no error

We mentioned that when any error is caught, numoferror is incremented. At the main part of the parser.y file, we finally check whether or not program is compiled with success. (Hence, this is why we keep "numoferr" variable.)

```
if (numoferr>0) {
          cout << "Parsing completed with " << numoferr << "
errors" <<endl;
     } else {
          cout << "Successfull" <<endl;
     }
```