V1.3

The Unnamed Language

As yet unnamed language is similar to the C programming language except that it supports the `string` datatype. A program must have a function in it called `main` that has return type `void` and accepts no parameters. There is no support for global variables.

Lexical Elements

Keywords: Keywords are shown in **bold** in the grammar: int, float, char, string, boolean, void, if, else, while, print, println, return, true, false

Punctuation Symbols: Shown in **bold** in the grammar: ( ) , { } ;

Identifiers: An identifier, shown as `id` in the grammar, is a sequence of letters, digits and the underscore character. An identifier cannot start with a digit.

Binary Operators: Binary operators, shown as `op` in the grammar, have the same precedences as in C or Java. The binary operators are shown in increasing order of precedence, however + and – have the same precedence: `== < + - *`

Constants: There are four types that can be specified as constant values:

`integerconstant`:	a sequence of decimal digits, for example: 34
`stringconstant`:	a sequence of quoted characters, for example “this is a string”
`charconstant`:	a single character enclosed in single quotes, for example 'a'
`floatconstant`:	a sequence of decimal digits followed by a . followed by a sequence of decimal digits, for example: 34.123

Comments: Single line comments only, where comments are prefixed by the two characters //

Sample Program

```
// sample.ul
int factorial (int n) {
      if (n == 1) {
            return 1;
      }
      else {
            return n*factorial(n-1);
      }
}
void main () {
      print "The factorial of 8 is ";
      println factorial(8);
}
```

V1.3

Grammar

```
Program             → Function+
Function            → FunctionDecl FunctionBody
FunctionDecl        → CompoundType id ( FormalParameters )
FormalParameters    → CompoundType id MoreFormals*
                    →
MoreFormals         → , CompoundType id
FunctionBody        → { VarDecl* Statement* }
VarDecl             → CompoundType id ;
CompoundType        → Type
                    → Type [integerconstant]
Type                → int
                    → float
                    → char
                    → string
                    → boolean
                    → void
Statement           → ;
                    → Expr ;
                    → if ( Expr ) Block
                    → if ( Expr ) Block else Block
                    → while ( Expr ) Block
                    → print Expr ;
                    → println Expr ;
                    → return Expr? ;
                    → id = Expr ;
                    → id [ Expr ] = Expr ;
Block               → { Statement* }
Expr                → Expr op Expr
                    → id [ Expr ]
                    → id ( ExprList )
                    → id
                    → Literal
                    → ( Expr )
Literal             → stringconstant
                    → integerconstant
                    → floatconstant
                    → characterconstant
                    → true
                    → false
ExprList            → Expr ExprMore*
                    →
ExprMore            → , Expr
```