1 Grammar

```
program = {typedef | function}

typedef = "record" , "{" , {type , identifier , ";"} , "}" , identifier , ";" |
          "variant" , "{" , {type , identifier , ";"} , "}" , identifier , ";"

function = (type | "void") , identifier , "(" , {type , ["*"] , identifier} , ")" , instruction

type = "int" | "bool" | "string" | identifier | "[" , type , "]"

instruction = "{" , {instruction} , "}" | function |
    "if" , "(" , value , ")" , instruction , ["else" , instruction] |
    "while" , "(" , value , ")" , ["as" , identifier] , instruction |
    "break" , [identifier] , ";" | "continue" , [identifier] , ";" |
    "return" , [value] , ";"
    type , identifier  , ["=" , value] , ";" |
    lvalue , ("=" | "+=" | "-=" | "*=" | "/=" | "++" | "--") , value , ";" |
    value , ";" |

value =  lvalue | "(" , value , ")" |
    integer | "true" | "false" | '"' , string , '"' |
    value , binary_op , value |
    unary_op , value |
    identifier , "(" , {["&"] , identifier} , ")" |
    "[]" | "[" , {value , ","} , value , "]" |
    "{" , {identifier , "=" , value , ";"} , "}" |
    ":" , identifier , "(" , value , ")" |
    "case" , value , "of" , {"|" , ":" , identifier , "(" , identifier , ")" , "->" , value} |

binary_op = "+" | "-" | "*" | "/" | "==" | ">" | ">=" | "<" | "<=" | "&&" | "||"

unary_op = "!" | "-"

lvalue = identifier | lvalue , "[" , value , "]" | lvalue , "->" , identifier
```

2 Description

My language is C-like imperative language with some extra features.

- Loops: classic `while(condition)` loop with optional named label (`while(condition) as label`), which allows for breaking specific loop. `break;` and `continue;` instructions modify flow of innermost while loop. `break label;` and `continue label;` affect loop named as `label` (unwrapping call stack if necessary) or result in error if no loop named as `label` is executed. Labels cannot be overriden.
- Built-in types: `int`, `string` and `bool`
- Built-in functions: `print(string str)`, `string int_to_string(int n)`, `string bool_to_string(bool b)`
- User defined types: `record` – like C struct; `variant` – tagged union, can be accessed using `case ... of` construction, similarly to Haskell
- Functions: can be `void` or return value of specific type. Arguments can be passed by value (`void f(int a)`) or by reference (`void f(int *a)`). Functions can be arbitralily nested and called recursively.
- Variables: only staticly typed variables are allowed. Variables are staticly bound with usual override semantics. Variables within blocks are accesible only from within that block. Variables outside all blocks are global variables, accesible after declaration from any instruction.
- Comments: start with `//` and ends with end of line

3 Example program

```
record {
    int number;
    [int] array;
    my_variant var;
} my_record;

variant {
    int number_case;
    bool boolean_case;
} my_variant;

int while_example() {
    int res = 0;
    int while_example_inner() {
        while (true) as inner {
            res++;
            if (res == 5) break inner;
            if (res == 10) break outer;
        }
    }
    while (true) as outer {
        while_example_inner();
    }
    return res;
}

void main() {
    void print_variant(my_variant v) {
         string str = case v of
             | :number_case(n) -> ":number_case(" + int_to_string(n) + ")"
             | :boolean_case(b) -> ":boolean_case(" + bool_to_string(b) + ")";
        print(str);
    }

    void negate_variant(my_variant *v) {
        v = case v of
            | :number_case(n) -> :number_case(-n)
            | :boolean_case(b) -> :boolean_case(!b);
    }

    my_record a = {
        number = 42;
        array = [1, 2];
        var = :number_case(1);
    };
    a->array[0]++;
    negate_variant(&(a->var));
    print_variant(a->var); // output: ":number_case(-1)"
    print(int_to_string(while_example())); //output: "10"
}
```