Grammar

1. <program> → { <decl_stmt> } { <function> }
    a. Add variables to the global symbol table.
    b. Add function metadata to the global symbol table.
2. <decl_stmt> → (int | float) ID [:= INT_LITERAL | FLOAT_LITERAL]
    { , ID [:= INT_LITERAL | FLOAT_LITERAL] } ;
    a. Extract variable metadata and add to appropriate symbol table.
3. <function> → (int | float) ID ([(int | float) ID { , (int | float) }])
    { { <decl_stmt> } { <imper_stmt> } <return_stmt> }
    a. Extract the parameter metadata and add to the local symbol table template.
    b. Extract the local variable metadata and add to the local symbol table template.
4. <return_stmt> → return <expr>;
    a. Emit: return
5. <imper_stmt> → <print_stmt> | <read_stmt> | <expr_stmt> | <if_stmt> | <while_stmt>
6. <print_stmt> → print [(STR_LITERAL | <expr>) { , (STR_LITERAL | <expr>) }] ;
    a. Emit: a separate print for each item.
7. <if_stmt> → if (<expr>) <block> [else <block>]
    a. Emit: if (with location of the instruction beyond the end of the first block).
    b. If the else exits, Emit: at the end of the first block, goto to the end of the second block.
8. <while_stmt> → while (<expr>) <block>
    a. Emit: if (with location following the goto instruction after the block -- see below).
    b. Emit: at the end of the block, insert a goto to the beginning of the while loop implementation (i.e., the if).
9. <block> → <imper_stmt> | { { <imper_stmt> } }
10. <read_stmt> → read [ID] ;
    a. Emit: read the a value into the identified variable
11. <expr_stmt> → [<expr>] ;
    a. Emit: clear the stack.
12. <expr> → ID := <expr>
    a. Emit: assign to ID.
13. <expr> → <C> { < | > | == | <> <C> }
    if the comparison operators exist then:
    a. Emit: appropriate compare operator
14. <C> → <T> { + | - <T>}
    a. Emit: add or subtract operator.

15. <T> → <F> { * | / <F>}
    a. Emit: multiply or divide operator
16. <F> → ID | INT_LITERAL | FLOAT_LITERAL
    a. Emit: push the specified value onto the stack.
17. <F> → ([-] <expr>)
    a. Emit: if the negation is specified, negate.
18. <F> → ID [([<expr> {, <expr>}])]
    a. Emit: call the identified function

Assignment Specification

Instruction	Description
Assign ID	Pop the value from the stack, assign it to the identified variable, push the value back on the stack
Add	Pop two values from the stack, push the sum back on the stack
Subtract	Pop value b from the stack; pop value a from the stack; push a-b on the stack
Multiply	Pop two values from the stack, push the product on the stack
Divide	Pop value b from the stack; pop value a from the stack; push a/b on the stack
Negate	Pop a value form the stack, negate it, push the result back on the stack
Lessthan	Pop value b from the stack; pop value a from the stack; push a<b back on the stack (i.e., 0 or 1)
Greaterthan	Pop value b from the stack; pop value a from the stack; push a>b back on the stack (i.e., 0 or 1)
Equalto	Pop value b from the stack; pop value a from the stack; push a==b back on the stack (i.e., 0 or 1)
Notequalto	Pop value b from the stack; pop value a from the stack; push a<>b back on the stack (i.e., 0 or 1)
Push value or ID	Push value on the stack. If ID is specified then get the value from the symbol table.
Goto location	Set the current program location to the specified location
Call ID	Lookup the specified function in the function symbol table. Determine how many parameter values need to be popped from the stack. Setup a new local symbol table for the function, initializing the parameter values using the popped values. Save the current program location in the new local symbol table and make this symbol table be active. Set the program location to the first instruction of foo.
Return	Leave expression value on the stack. Set the program location to the value saved in the local symbol table. Delete the local symbol table. Restore the previous symbol table (if any).
Print [string]	If string is specified, print it. Otherwise pop a value from the stack and print it.
Read [ID]	If ID is specified then read a value from the console into the identified variable. If an ID is not specified then throw away any inputs.
If location	Pop value from stack. If 0 set program location to the specified location

Implementation Hints

```
Class CInstr
{
        public:
        int     m_iInstr;               // Instruction code
        int     m_iValue;               // Any integer value (or program location)
        float   m_fValue;               // Any float value
        string  m_strValue;             // Any identifier

        // Add constructors and destructors here…
};

// A program is essentially an array of instructions.
typedef vector<CInstr> CProgram;


Class CSymTabEntry
{
        public:
        int     m_iType;        // indicates int, float, or location
        int     m_iValue;       // holds integer value or location value
        float   m_fValue;
};

Class CLocalSymbolTable
{
        public:
        int                             m_iReturnAddr;
        map<string, CSymTabEntry>       m_mapSymbols;
};

// Every time a function is called, add a new local symbol table to the top of the stack of
// local symbol tables.  This allows recursion – every time a function is called, a fresh local
// symbol table is created for it.
typedef stack< CLocalSymbolTable> CStackLocSymTable;
```

```
// The following two classes can be used to store information about functions.
Class CFuncSymTabEntry
{
        public:
        string   m_strName;    // Name of parameter or local variable
        int      m_iType;      // indicates int or float
        int      m_iValue;     // holds integer value or location value
        float    m_fValue;
};


Class CFunction
 {
        public:
        string   m_strID;
        int      m_iRetType;
        int      m_iLocation;
        int      m_iNumParams;
        vector< CFuncSymTabEntry > vecSymTabEntry;
}
```

```
int k;     // add k to global symbol table
int n;     // add n to the global symbol table
int foo(int p1)  // add info about foo (including location, and parameters) to function symbol table.
{          int m;    // add m to function symbol table info about foo
           print “enter m:”;          // emit: print (str, “enter m:”) Note: this is location 0
           read m;                    // emit read m
           print “k: ”,  k, “ and m: ”, m;       // emit: print (str, “k:”)
                                                 // emit: push value of k (production 16)
                                                 // emit: print
                                                 // emit: print (str, “ and m: ”)
                                                 // emit: push value of m (production 16)
                                                 // emit: print
                                                 // emit: print (str newline)
           k := 3 + p1 * m;           // emit: push 3
                                      // emit: push p1
                                      // emit: push m
                                      // emit: multiply
                                      // emit: add
                                      // emit: assign k
                                      // emit: pop (clear the stack because of the ‘;’, we don’t need to save
                                      //         the result)
           return (-m);       // emit: push m
                              // emit: negate
                              // emit: return
}
```

```
int main()  // add info about foo (including location, and parameters) to function symbol table.
{
        n := k := 4;        // emit: push 4.  Note: this is location 19
                            // emit: assign k (TOS is popped, assigned to k, and then pushed back)
                            // emit: assign n (TOS is popped, assigned to n, and then pushed back)
                            // emit: clear stack (because of the ';', we don't need to save the result)

        print foo(n);       // emit: push the value of n on the stack (production 16)
                            // emit: call foo (this means, lookup foo in the function symbol table,
                            //                 determine how many parameter values need to be popped
                            //                 from the stack.  Setup a local symbol table for the function
                            //                 initializing the parameter values using the popped values.  Save
                            //                 the current program location in the new local symbol table and
                            //                 make this symbol table be active. Set the program
                            //                 location to the first instruction of foo.
                            // emit: print

        if (n > 0) {                        // emit: if with location of the instruction following the else
           print "n is positive, ";         // emit: print (str "n is positive, ")
           print n;                         // emit: print
        }                                   // emit: print newline
                                            // emit: goto location of instruction right after the else block
        else
           print "n is negative";           // emit: print (str "n is positive, ")
                                            // emit: print newline

        return 0;           // emit: push 0
                            // emit: return
}
```