Concordia University
Department of Computer Science
and Software Engineering

Compiler Design (COMP 442/6421)
Winter 2019

Assignment 4, Code Generation

Deadline:	Saturday April 10th, 2019
Evaluation:	10% of final grade
Late submission:	not accepted

In this assignment, you are to implement a code generation phase for the language as described in assignment 2 and 3. The following is a list of different specific constructs/concepts for which code generation needs to be implemented for all the aspects of the language to become executable:

1. Memory allocation: As variables are declared, memory space must be allocated that will eventually be used to store their values. Memory cells are either identified using a unique Moon assembly code label, or an offset relative to the stack frame on which the variable will reside during execution of the scope in which it is declared. The size of the allocated memory block should depend on the declared type/kind of variable (integer, float, array, object, array of objects, etc.)

1.1 Allocate memory for basic types (integer, float).
1.2 Allocate memory for arrays of basic types.
1.3 Allocate memory for objects.
1.4 Allocate memory for objects with inheritance.
1.5 Allocate memory for objects having object members.
1.6 Allocate memory for arrays of objects.

2. Functions: The code generated for functions should be associated with a Moon assembly code subroutine, i.e. a block of code that can be jumped to using a label, then when the function resolves, jump back to the instruction following the initial jump. Parameters should be passed/received during the function call. The return value should be passed to the calling function when the function resolves. In order to allow recursive function calls, a function call stack mechanism needs to be implemented, which implies that memory allocation for variables must all be done using offsets relative to the bottom of the current function's stack frame. If a call to a member function is made, the member function should have access to the data members of the object from which it was called.

2.1 Branch to a function's code block, execute the code block, branch back to the calling function.
2.2 Branch to a function that has been branched upon.
2.3 Pass parameters as local values to the function's code block.
2.4 Upon function resolution, pass the return value back to the calling function.
2.5 Function call stack mechanism and recursive function calls.
2.6 Call to member functions.
2.7 Call to deeply nested member function.

3. Statements: Implementation of Moon code for every kind of statement as defined in the grammar: assignment conditional statement, loop statement, input/output statement, return statement. Correct implementation the specific branching mechanisms for control flow statements.

3.1 Assignment statement: correct assignment of the resulting value of an expression to a variable, independently of what is the expression.
3.2 Conditional statement: correct implementation of branching mechanism, including for imbricated conditional statements.
3.3 Loop statement: correct implementation of branching mechanism, including for imbricated loop statements.
3.4 Input/output statement: execution of a read() statement should result in the user being prompted for a value from the keyboard by the Moon program, and assign this value to the parameter passed to the read() statement. Execution of a write() statement should print to the Moon console the result of evaluating the expression passed as a parameter to the write() statement.
3.5 Return statement: execution of the return statement should result in passing the value of the expression passed to the return statement back to the calling function.

4. Aggregate data members access: Aggregate data types such as arrays and objects contain a group of data values. Code must be generated so that contained member values in such an aggregated value can be accessed when referred to as factors in an expression, or the left hand side of an assignment statement. This includes access to array elements and object members.

4.1 For arrays of basic types (integer and float), access to an array's elements, single or multidimensional.
4.2 For arrays of objects, access to an array's elements, single or multidimensional.
4.3 For objects, access to members of basic types.
4.4 For objects, access to members of array types, as well as the elements of the array.
4.5 For objects, access to members of object types, as well as the elements of this object.
4.6 For objects, access to deeply nested objects.
4.7 For objects, access to the members of a superclass.

5. Expressions: Computing of the resulting value of an entire expression, including the simple case when an expression is either a variable name of even a single literal value, up to complex expressions involving a kinds of operators, array indexed with expressions, deeply nested object members, etc. This involves memory allocation for temporary results, register allocation/deallocation

5.1 Computing the value of an entire expression.
5.2 Expression is a single variable or literal value.
5.3 Expressions involving all of: arithmetic, relational and logic operators in one expression
5.4 Expression involving an array factor whose indexes are themselves expressions.
5.5 Expression involving an object factor referring to deeply nested object members.
5.6 Memory allocation for temporary results.
5.7 Register allocation/deallocation scheme.

Documentation

1. **Analysis**
   - 1.1. Description of the register allocation/deallocation scheme used in the implementation.
   - 1.2. Description of the memory usage scheme used in the implementation (tag-based, stack-based), including for temporary variables, function calls (free functions or member functions, data members, and calculation of variables' allocated memory).
   - 1.3. Description of the purpose of each phase involved in the implemented code generation. For each phase, mapping of semantic actions to AST nodes, along with a description of the effect/role of each semantic action.
2. **Design**
   - 2.1. Description/rationale of the overall structure of the solution and the roles of the individual components used in the applied solution.
3. **Use of Tools**
   - 3.1. Description/justification of tools/libraries/techniques used in the analysis/implementation.
   - 3.2. Successful/correct use of tools/libraries/techniques used in the analysis/implementation.

Assignment submission requirements and procedure

You have to submit your assignment before midnight on the due date using the ENCS Electronic Assignment Submission system under the category "*programming assignment 4*". The file submitted must be a **`.zip`** file containing:

- all your code
- a set of input files to be used for testing purpose, as well as a printout of the resulting output of the program for each input file (moon code output)
- a simple document containing the information requested above/below

You are also responsible to give proper compilation and execution instructions to the marker in a README file.

Evaluation criteria and grading scheme

Analysis:		
Description of the register allocation scheme and general memory usage scheme used in the implementation	ind 2.1	1 pts
Description of the purpose of each phase involved in the implemented code generation. For each phase, mapping of semantic actions to AST nodes, along with a description of the effect/role of each semantic action.	ind 2.2	4 pts
Design/implementation:		
Description/rationale of the overall structure of the solution and the roles of the individual components used in the applied solution.	ind 4.3	3 pts
Correct implementation according to the above-stated requirements.	ind 4.4	17 pts
Output of executable Moon code in a file.	ind 4.4	3 pts
Completeness of test cases.	ind 4.4	17 pts
Use of tools:		
Description/justification of tools/libraries/techniques used in the analysis/implementation.	ind 5.2	2 pts
Successful/correct use of tools/libraries/techniques used in the analysis/implementation.	ind 5.1	3 pts
Total		**50 pts**