Code Generation Implementation
Deliverable Report E

Assignment 5

Team 3

Instructor:	**Prof. Dave Wortman**
Course:	**CSC488S Compilers & Interpreters**
Due Date:	**April 2nd 2015 1:00 PM**
Team Members:	**Adrian Banks (c1banksa)**
	Benson Quach (g0quachb)
	Kent Liang (c2liangt)
	Nicholas Dujay (c4dujayn)

1.0 Design Documentation

1.1 Overall Code Generation Design

All code generation methods can be found in the **A5/src/compiler488/codegen/CodeGen.java** file. We created helper functions to mimic the assembly code templates as closely as possible. For example, in regards to the ADDR machine instruction call we produced the following helper:

```
private void ADDR(short LL, short ON) {
        assembly(Machine.ADDR, LL, ON);
}
```

wrapping the assembly call to the machine number representing the **ADDR** call with the provided arguments. We did this for all machine instructions used within this assignment. The assembly function handles writing to the correct location in memory (and catching exceptions), and we can use the CodeGen classes generationAddress member for return address patching.

To reiterate once again, we've implemented and designed our AST for the compiler using the visitor pattern and as such the **CodeGen**.**java** file mentioned contains visit functions for each of the AST Node types. Within the respective nodes we implemented our assignment 4 code generation templates in the mentioned categories below. You'll find that the implementation are extremely intuitive and alike with our assignment 4.

Another convenient design decision we've made is the creation of **CODEGEN_ADDR**() function which helps determine and push the address of the specified argument on top of the runtime stack. The specified argument consists of 1D or 2D arrays along with scalar variables. Similarly we also have **CODEGEN**() which given an argument performs the respective machine instruction actions for the provided AST Node.

1.2 Expressions

Expressions implementation was identical to assignment 4's template. The only addition we've made in this section was short-circuiting/conditionals which we missed in the previous. For yield statements we implemented as similar to a routine function taking no arguments. Please refer to the routine section for further details.

arithmetic expressions

Within the ArithExpn visit function in CodeGen.java you'll find the respective implementation.

boolean expressions

Within the BoolExpn & BoolConstExpn visit functions in CodeGen.java you'll find the respective implementation.

comparisons

Within the CompareExpn visit function in CodeGen.java you'll find the respective implementation.

yield expression

Within the AnonFuncExpn visit function in CodeGen.java you'll find the respective implementation.

1.3 Statements

assignment

Within the AssignStmt visit function in CodeGen.java you'll find the respective implementation
Similarly with the help of our helper functions, we just simply follow our design document. We first call codegen address to get the address of the variable at the left hand side and call codegen to generate the expression at the right hand side.Then, store it.

if
Within the IfStmt visit function in CodeGen.java you'll find the respective implementation.
Similarly, with the help of our helper functions, we just simply follow our design document.

If statement without else:
SImply codegen the condition of the if statement and call BF(), then patch the address after the codegen of the if body is completed.

if statement with else:
Simply codegen the condition of the if statement, then patch the address of the start of else body when the codegen of if body is completed. After that, codegen the else body and patch the address where the if statement ends.

loop
Within the LoopStmt visit function in CodeGen.java you'll find the respective implementation.
Similarly, with the help of our helper functions, we just simply follow our design document. Save the address of the start of the loop and branch it back to there once the codegen of the loop body is finished. Afterward, patch the address of the exit statement and performs clean up of the global array when the patching is finished.

while
Within the WhileDoStmt visit function in CodeGen.java you'll find the respective implementation.
Similarly, by our helper functions, we just simply follow our design document. Save the address of the start of the while loop. Then, codegen the condition(expression). Push the end of the while loop's address which will be patched later, then call BF(). Codegen the body of the while loop and branch back to the start of the while loop. Afterward, patch the address of the exit statement and performs clean up of the global array when the patching is finished. Then, we patched the end of while loop address.

exit
Within the ExitStmt visit function in CodeGen.java you'll find the respective implementation.
Exit statement will codegen the expression and add the current generation address to a global array. After codegen the body of loop or while, the address will be patched.

get, put
Within the GetStmt & PutStmt visit function in CodeGen.java you'll find the respective implementation.

1.4 Functions/procedures

parameter passing
Within the visitPendingRoutine(s) in CodeGen.java you can find the implementation of the activation record setup. The parameters are evaluated before updating the display.

display & addressing
Within the visitPendingRoutine(s) in CodeGen.java you can find the implementation of the activation record setup.

local storage
The semantic checker calculated the offset to each variable, even in function scope based on the agreed upon activation record structure. With arrays, the offset was also calculated for the follow variables to be after all the values in the array.

embedded scopes
The symbol tables for embedded scopes were merged into their parent's symbol table. This required modifications to the Semantics.java file.

call of function/procedure
Within the FunctionCallExpn and ProcedureCallStmt visit functions in CodeGen.java you can find the implementation.

return
Within the visitPendingRoutine(s) in CodeGen.java you can find that all return statements just branch to the beginning of the activation record cleanup section.

2.0 Testing Documentation

2.1 Expressions

Expression test cases can be found in the expressions-codegen.488 file and cases tested are shown below:

array assignment + expressions test cases
% 1D beginning and end array assignment test
% 1D complete array assignment test
% 2D beginning and end array assignment test
% 2D complete array assignment test

arithmetic expressions test cases
% addition
% subtraction
% multiplication
% division

boolean expressions test cases
% negation
% boolean and
% boolean or
% short-circuit (AND)
% short-circuit (OR)

comparison test cases
% less than
% less than or equal
% greater than
% greater than or equal
% equal to
% not equal to
% unary minus

yield expressions test cases
% yield; scope variable declaration and return
% yield; outer scope variable modification and return

2.2 Statements

assignment
Assignment only has a basic test file. It's been thoroughly tested in expression. On top of that, we look at the memory by tron and troff to ensure assignment is correct.

if
The test case covers if with else and without else and nested if statement. The file name is self explanatory.
%if_basic
%if_else_basic
%if_nestedif

while, loop, exit
All combination of those three statements have been thoroughly tested.
For loops:

%loop_basic_exit
%loop_basic_exitwhen
%loop_nestedloop
For while loops
without exit
%while_basic
%while_nestedwhile

with exit
%while_exit
%while_exitwhen
%while_nestedwhile_exit there is a combination of exit and exit when inside the test case.

get, put
For get, we make sure that it works for all form of integer variable. (integer, integer 1d array, integer 2d array)
For put, we just print out all the possible outputs of put. i.e integer constant, expression, skip, text. Manually check it matches.

2.3 Functions/procedures

parameter passing
% test passing 0 parameters to a function
% test passing 1 parameter to a function
% test passing n parameters to a function

% test passing 0 parameters to a procedure
% test passing 1 parameter to a procedure
% test passing n parameters to a procedure

display & addressing
This part of testing is done by tron and troff. We as a team traced through the code to make sure the addressing and display is correct for simple procedure/function test cases.

local storage
% parameter access with 0 local variables
% parameter access with 1 local variable
% parameter access with n local variables

% local variable access with 0 parameters
% local variable access with 1 parameter
% local variable access with n parameters

embedded scopes
% An embedded scope with 1 local variable in the program scope
% An embedded scope with n local variables in the program scope

% An embedded scope with 1 local variable in a function scope
% An embedded scope with n local variables in a function scope

% An embedded scope with 1 local variable in a procedure scope
% An embedded scope with n local variables in a procedure scope

% 2 embedded scopes with n variables each within the program scope
% n embedded scopes with n variables each within the program scope

% An embedded scope with 1 local variable within an embedded scope
% An embedded scope with n local variable within an embedded scope

call of function/procedure

% calling functions with a higher lexical level than the caller.
% calling functions with a lower lexical level than the caller.
% calling functions with the same lexical level as the caller.

return

The return statement itself does not have a test case. It's tested by the other function and procedure test cases. In those test cases, return with no argument for procedure, return with expression, integer constant, and function calls for functions have all been tested.

3.0 Who-did-What

Adrian Banks (c1banksa)

Contributions:

Implemented part 3(procedure/function) and return stmt of the assignment and their respective test cases. Helped write the function/procedure design/testing documentation.

Benson Quach (g0quachb)

Contributions:

Implemented all expressions code generation and 1D & 2D array addressing with their respective test files/cases. Created outline for design document, wrote general design section and expressions section.

Kent Liang (c2liangt)

Contributions:

Implemented part 4 statements excluding return stmt and their respective test cases. Completed the statements section of the design/testing documentation. Help wrote a small portion of the testing documentation for function and procedure.

Nicholas Dujay (c4dujayn)

Contributions:

Implemented initial helper functions to map to the design document, implemented storage. Fixed A3 bugs and implemented offset calculation in Semantics.