ECE220: Computer Systems and Programming **Fall 2018 ZJUI**
Machine Problem 11 **due: SUNDAY 23 December at 11:59:59 p.m.**

Code Generation for an LC-3 Compiler

This assignment requires you to use recursion to translate a description of a program in a subset of the C language into LC-3 assembly code. Developing such code provides you with experience with the use of recursion and pointer-based data structures in C, both of which are important tools. You will also gain a better understanding of the methods and challenges involved in automating translations from one form of program to another. Most of the compiler will be provided to you, including both code that translates C code into an abstract syntax tree (AST) as well as C library routines to support basic arithmetic and I/O on LC-3. Your focus will be on turning an AST into a sequence of LC-3 instructions.

Please read this document in its entirety before you begin to program.

On Reading the Code

You are not expected to read the more than 3,300 lines of C, lexer, parser, and LC-3 code provided to you. You are welcome to do so, of course, but you do not need to read them to complete this assignment. All relevant material from headers, *etc.*, is included in this document. Furthermore, you should be aware that a substantial amount of code (around 4,500 lines!) is automatically generated when you build your compiler, and none of the automatically generated code (*e.g.*, `ece220_lex.yy.c`) is meant to be human-readable. Don't try. If you want to remove the automatically-generated files so that you can see what was there in the original distribution, type `make clear`.

A Subset of C

To simplify your task, the compiler supports only a narrow subset of the full C programming language. Support is included for `if` statements, `for` loops, `return` statements, and expressions using most of the C operators. Compound statements are **required** as opposed to simply being good practice. For example:

```
int i;

for (i = 0; 10 > i; i++)   /* This code will generate a syntax error.  */
    printf ("%d\n", i);

for (i = 0; 10 > i; i++) { /* To avoid, include braces around loop body.  */
    printf ("%d\n", i);
}
```

Programs written for this compiler can declare global variables as well as a single subroutine (`int main ()`, of course). Local variables can be declared within `main`, but variable declarations are not otherwise allowed (*e.g.*, within compound statements). Only two types are supported: `int`'s and one-dimensional arrays of integers. Pointer types are **not** supported, and you can use arrays only with bracketed expressions for indices. For example:

```
int a[1];
scanf ("%d", a);     /* NOT legal--will generate a syntax error */
scanf ("%d", &a[0]); /* legal */
```

Constant strings can be used, but only as arguments to function calls, as shown by the format strings in the `printf` and `scanf` examples above.

Most C operators are supported, including assignment (`=`), basic arithmetic (`+`, `-`, `*`, `/`, `%`, and negation), pre- and post-increment and decrement (`++`, `--`), logical operators (`!`, `&&`, `||`), and comparisons (`<`, `<=`, `==`, `>=`, `>`, `!=`). Taking the address of a variable or array element is also possible (as is necessary for `scanf`), but almost no type checking is done for you.

Thus the compiler accepts the code below (as does `gcc`, but accompanied by warnings):

```
int i;
int j;
i = &j;
printf ("%d\n", "Hello!\n");
printf ("i at %d, j at %d\n", &i, &j);
scanf ("%d", i);
```

The output produced by the `printf` calls in the code above is unpredictable and of little value, but the `scanf` call passes the value of variable `i` instead of its address and thus could change memory unpredictably. In the code above, `i` was set previously to the address of `j`, but such tricks are confusing and error-prone.

Examples of other missing functionality include pointer dereferencing, multi-dimensional arrays, structures, enumerations, defining new types, variable initialization within a declaration, loops other than `for` loops, `switch` statements, and bitwise operations. Anything not explicitly mentioned as a type of AST node in the AST section of this document is simply not included in the language, and you need not worry about implementing it. Of course, you won't be able to use such functionality in C programs on which you want to use your compiler, either.

Code Generation

Your code must generate LC-3 assembly for the `main` subroutine and print it to `stdout`. As you know, a C function such as `main` consists of variable declarations and statements. Variable declarations will be translated for you into a symbol table that your code must use when generating assembly code for each of the statements. The stack frame will already be set up for you, and will be torn down when your code finishes. Be sure to execute the stack frame teardown—do **not** insert RET instructions directly into the code that you generate.

This machine problem is not intended to require substantial LC-3 programming on your part. Each of the recursive components that you write should be no more complex than implementing a single step of systematic decomposition, and you may in fact want to review the LC-3 implementation of the conditional and iterative decompositions that we covered early in the course.

In addition to statements, much of the work involved in code generation involves computing the results of expressions. To simplify your task, you are required to use the stack to compute all expressions. As you may recall, any expression can be computed using a stack and some basic rules. To "execute" a literal operand such as a number is seen, push the operand onto the stack. To "execute" an operator, pop operands for the operator from the stack, apply the operator to those operands, and then push the operation result back onto the stack.

Although the LC-3 code generated by this approach will be lengthy and inefficient, use of a stack combines nicely with recursion to allow a straightforward implementation of most of this MP. For example, the code for an addition operator (`AST220_OP_ADD`, as described later) performs the following sequence of operations:

generate code to produce the first operand of the add	(recursive call)
generate code to produce the second operand of the add	(recursive call)
pop the second operand from the stack	(print LC-3 instructions)
pop the first operand from the stack	(print LC-3 instructions)
add the two together	(print an `ADD` instruction)
push the result onto the stack	(print LC-3 instructions)

The LC-3 instructions that you produce must be aware of the LC-3 register conventions as well as any assumptions that you make in code that can call it recursively. Consider, for example, what might happen if we did not use a stack-based approach for expressions: a given binary operator might produce its first operand and put it into R0, then generate code to produce its second operand. However, the code needed for the second operand must avoid overwriting the first operand in R0. What if the same operator is used by the second operand, though (*e.g.*, 1+(2+3))? Use of a stack avoids such complexity.

The register conventions for LC-3 are as follows:

R0-R3	available for your code's use
R4	global data pointer
R5	`main`'s stack frame pointer
R6	stack pointer
R7	used by JSR/JSRR

Note also that the assembly routines provided for you do change certain register values when they are called (details are in the next section).

The C Library

In order to avoid your having to implement functionality in LC-3, we have written and provided you with a small library of functions to support you when generating the assembly code. The library includes four interfaces that can be used directly from C and another three interfaces intended for your use in implementing basic arithmetic. The calls visible in C are as follows:

```
int printf (const char* format, ...); /* assembly label PRINTF */
int rand ();                          /* assembly label RAND   */
int scanf (const char* format, ...);  /* assembly label SCANF  */
void srand (int new_seed);            /* assembly label SRAND  */
```

These calls have C-style interfaces: arguments must be put onto the stack in the correct order, and the return value will be on the stack. Except for R6 (decremented to store the return value) and R7 (changed by JSR), no register values are changed by these subroutines. As described in class, the caller is responsible for removing arguments from the stack after the call to any of these functions returns.

The `printf` and `scanf` routines work in the standard way, but support only a handful of escapes:

`%d`	print/scan an argument as a decimal integer
`%%`	print/scan a single % character
`\n`	print/scan a line feed
`\\`	print/scan a single \ character

The `printf` routine returns the number of characters printed. The `scanf` routine returns the number of integers converted. The `rand` routine returns a pseudo-random number between 0 and $2^{15}-1$, and the `srand` routine sets the seed for the pseudo-random number generator used by `rand`.

The arithmetic functions defined for you have register-based interfaces. All three are binary operations on R0 and R1, and all three return the result of the operation in R0. Except for R0 (the return value) and R7 (changed by JSR), no register values are changed by these subroutines. The routines are as follows:

DIVIDE	divide R0 by R1, round towards 0, and store the result in R0
MODULUS	calculate R0 % R1 (C definition) and store the result in R0
MULTIPLY	multiply R0 by R1 and store the result in R0

All seven of these library routines use the stack for storing callee-saved registers.

Abstract Syntax Trees

The abstract syntax tree (AST) is one form of intermediate representation used as interface between the front-end and the back-end of a compiler. Recall that the front-end of a compiler translates from a high-level language (*e.g.*, C) into an intermediate representation, while the back-end translates the intermediate representation into assembly code. Your job in this assignment is to write an LC-3 back-end for an intermediate representation based on ASTs.

An AST is a pointer-based data structure consisting of nodes that represent statements and expressions in the original code. The structure used in your compiler is the following:

```
typedef struct ast220_t ast220_t;
struct ast220_t {
    ast220_type_t         type;   /* type of AST node                  */
    int32_t               value;  /* a number (PUSH_INT, DEBUG_MARKER)  */
    char*                 name;   /* a string (PUSH_STR, VARIABLE)      */
    ast220_builtin_func_t fnum;   /* function number (FUNC_CALL)        */
    ast220_t*             test;   /* test condition (IF_STMT, FOR_STMT) */
    ast220_t*             left;   /* left child/first operand           */
    ast220_t*             middle; /* middle child (FOR_STMT)            */
    ast220_t*             right;  /* right child/second operand         */
    ast220_t*             next;   /* next AST node                      */
};
```

The `type` field of an AST node (the `ast220_t` structure) specifies what kind of node is represented. For example, one AST node might represent an `if` statement, while a second node represents a variable reference, and a third node represents an addition operation. **In general, the type of the AST node determines the meaning (or lack thereof) of the structure's other fields.** The exception to this rule is the `next` field, which is used to link together AST nodes representing sequential statements in a block of code as well as sequential arguments in a function call.

Two general classes of AST nodes are defined for your compiler: statements and expressions. The subset of C supported by the compiler is minimal, thus only five statement types are possible:

AST220_FOR_STMT	a `for` statement	
	`test`	test condition
	`left`	initialization expression (an AST220_POP_STACK node, or NULL)
	`middle`	loop body (statements linked by `next`)
	`right`	update expression (an AST220_POP_STACK node, or NULL)
AST220_IF_STMT	an `if` statement	
	`test`	test condition
	`left`	then code block (statements linked by `next`)
	`right`	else code block (statements linked by `next`)
AST220_RETURN_STMT	a `return` statement (don't forget to branch to the stack frame teardown code provided to you)	
	`left`	expression to be returned
AST220_POP_STACK	an expression, with result discarded (*e.g.*, `a=b;`)	
	`left`	expression to produce and then pop (discard)
AST220_DEBUG_MARKER	a debugging marker	
	`value`	marker value

Fields not defined in the table above have no meaning and should not be assumed to hold any particular value (*e.g.*, NULL). The only statement type that might not be immediately clear to you from your experience with C is the `AST220_DEBUG_MARKER`, which is simply a debugging aid designed to help you with your coding. If you add a statement of the form `DEBUG(20);` to your C program, an AST node of type `AST220_DEBUG_MARKER` will appear in its place (with `value` equal to 20), which in turn enables generation of an assembly comment marking that point in the assembly output (using a routine already written for you).

Expression nodes include all operators supported by our subset of C as well as nodes for integer and string constants, variable references, and so on. The simplest expressions include constants, variable references, and function calls, for which AST types are shown in the table below. In general, the code generated (recursively) for an expression AST node should leave the result of the expression on top of the stack.

AST220_PUSH_INT	an integer constant	
	`value`	the number itself (fits into 16-bit 2's complement)
AST220_PUSH_STR	a string constant	
	`name`	the string, including quotation marks
AST220_VARIABLE	a variable reference	
	`name`	variable name (look up in symbol table)
	`left`	array index expression (for array variables; NULL for scalar variables)
AST220_FUNC_CALL	a function call; be sure to pop off arguments, but leave the return value on top of stack	
	`fnum`	C library function identifier; one of: AST220_PRINTF AST220_RAND AST220_SCANF AST220_SRAND
	`left`	argument list (expressions **from right to left** linked by `next`)

The `AST220_VARIABLE` node is special in the sense that it is used both for expressions that use the value as well as the address of the variable. In a general expression, you should assume that the value of the variable must be found and pushed onto the stack. The address of the variable plays a role whenever the `AST220_VARIABLE` node is the child of one of the types in the table at the top of the next page. In these cases, you should process the variable directly and not treat it as a general expression (*i.e.*, do not recurse to your routine for generating the code for an arbitrary expression).

AST220_GET_ADDRESS	the address of a variable `left` the variable, an AST220_VARIABLE node; note that array variables will also have an expression for the array index
AST220_OP_ASSIGN	an assignment; copy the expression into the variable, but do not pop the expression result off of the stack `left` the variable, an AST220_VARIABLE node `right` the expression
AST220_OP_PRE_INCR AST220_OP_PRE_DECR AST220_OP_POST_INCR AST220_OP_POST_DECR	pre- and post-increment and decrement `left` the variable, an AST220_VARIABLE node

The remaining AST nodes correspond to unary and binary operators for arithmetic, logical operations, and comparisons. Remember that assembly code subroutines for multiplication, division, and modulus are provided for you.

AST220_OP_NEGATE	negate an expression `left` the operand (an expression)
AST220_OP_ADD AST220_OP_SUB AST220_OP_MULT AST220_OP_DIV AST220_OP_MOD	binary arithmetic operations `left` the first operand (an expression) `right` the second operand (an expression)
AST220_OP_LOG_NOT	logical NOT; result is 0 or 1 `left` the operand (an expression)
AST220_OP_LOG_AND AST220_OP_LOG_OR	binary logic operations; result is 0 or 1; don't forget to perform shortcutting `left` the first operand (an expression) `right` the second operand (an expression)
AST220_CMP_NE AST220_CMP_LESS AST220_CMP_LE AST220_CMP_EQ AST220_CMP_GE AST220_CMP_GREATER	comparison/relation operations; result is 0 or 1 `left` the variable, an AST220_VARIABLE node `right` the expression

Implementation of LC-3 code generation for comparisons is provided to you as a building block. The implementation provided makes the following simplifying assumption: **for the purpose of all comparisons, the resulting code ignores 2's complement overflow.** As you know, the LC-3 ISA supports comparisons only with 0. Thus it is natural to transform comparisons by subtracting the right side from the left side and comparing the result with 0. This approach **does not work** when the representation overflows. For example, is $A < B$ true when $A = 0\text{x}4000$ and $B = 0\text{x}8000$? Clearly not: B is negative! On the other hand, $A - B = 0\text{x}4000 - 0\text{x}8000 = 0\text{xC}000 < 0$, which is true.

The Symbol Table

A symbol table will be constructed and provided for you. As described previously, AST nodes of type `AST220_VARIABLE` use the `name` field to specify the variable name referenced. In the case of an array variable, the node's `left` field will hold an expression to calculate the array index. The parser checks that an index expression is always present for an array variable and is never present for a scalar (non-array) variable. Two symbol table routines are of interest to you:

```
void symtab_dump ();
symtab_entry_t* symtab_lookup (const char* vname);
```

The `symtab_dump` routine is called for you to place a human-readable copy of the symbol table into the assembly code output, but you can call it again if you find it convenient to make additional copies. However, these extra copies should be removed before you turn in your code.

The `symtab_lookup` routine finds the symbol table entry associated with a given variable name. We disallow variable shadowing (using the same name for both a local and a global variable). The symbol table entry is defined as follows:

```
typedef struct symtab_entry_t symtab_entry_t;
struct symtab_entry_t {
    char* name;      /* variable name (case sensitive)              */
    int   array_len; /* size of array (0 = not an array)            */
    int   is_global; /* global variable (1) or local to main (0)    */
    int   offset;    /* offset with respect to appropriate register */
};
```

When you generate LC-3 instructions to operate on a variable, you must use the `is_global` and `offset` fields of the variable's symbol table entry to locate it. As you may recall from class and/or the text, global variables are relative to R4, the global data pointer, and local variables are relative to R5, `main`'s frame pointer.

You may assume that the symbol table offsets for all variables fall in the range $[-16, 15]$**.** Large arrays may extend beyond this interval, but the first element of any array will always fall within it when we test your MP.

Unique Labels

A facility for creating and using unique labels in the assembly code that you generate is provided for you:

```
ece220_label_t* label_create ();
char* label_value (ece220_label_t* label);
```

When you need a label, declare a variable of type `ece220_label_t*` and obtain a new label by calling `label_create`. To print the label, call `label_value` to obtain a string representation from the label pointer.

Getting Started

After reading this document completely, go into your `MP11H` directory and type `make` to compile everything into an executable called `c220`. At this point, the `c220` compiler will produce valid assembly output from a valid C program, but the assembly code implementation of the `main` subroutine will not actually do anything other than stack frame setup and teardown.

Your code goes into the `mp11.c` file. Specifically, you must fill in the function

```
void MP11_generate_code (ast220_t* prog);
```

Note that you **may not modify** any files other than `mp11.c`. We will not use any other files from you in testing your code.

Existing Code Generation

Some of the work needed to generate code has been done for you, and the `mp11.c` file provided to you already contains a significant amount of the necessary structure. What's there is about 700 lines, most of which is prototypes and comments.

Code optimization is not a goal for this assignment, and thus any reasonably-sized piece of C code is likely to break the limits on LC-3 branch and JSR offsets. To avoid assembler failures, you may only use the branch instruction when you know that the branch offset is in range, and you must always use JSRR rather than JSR. For the branches, a routine to print an appropriate instruction sequence for a "branch" with arbitrary offset (*i.e.*, a JMP instruction) is provided to you:

```
void gen_long_branch (br_type_t type, ece220_label_t* label);
```

Branch types are defined near the start of the `mp11.c` file.

Also included in `mp11.c` are a couple of big switch statements to direct recursive calls for generic statements (`gen_statement`) and expressions (`gen_expression`). These switch statements direct each type of AST node to a small function to generate the code for that type of node. You will have to write the code for most of these smaller functions, but some (the six comparison operations) have been provided for you as an example. The debug marker function is also included.

First Half

You must complete all tasks in both halves by the deadline. The separation is meant to only to guide you in how to approach the problem. We suggest that you do so in the order listed here, which will allow you to check your progress.

- `MP11_generate_code`: Start by writing the main entry point for your code. Until you've written this function, none of the other functions in `mp11.c` will be called. The `ast` argument is a linked list of AST nodes (linked by the `next` field) representing the statements in the C program's `main` function. All you have to do for now is generate code for each one.
- integer constants: Next, write code to push a constant integer on to the stack. Here, you'll need to play the same trick as you'll see in the long branch code provided to you. That is, put the .FILL with the constant into the code, then branch around it. Ugly, but this approach guarantees that your LD instruction has a valid offset.
- `return` statement: Implement the `return` statement. The stack frame is already set up for you, so you can just look up the relative offset in the book or your notes. Keep in mind that you'll need to branch to the stack teardown after you've set the return value. Add code to `MP11_generate_code` to put the label in the right place, and keep track of the label in a file-scope variable.

  Once you have reached this point, you can compile, assemble, and simulate a program consisting of a return statement. The code that calls main also prints the return value; try it out and make sure that your code works so far.

- negation: Negation is one of the simplest operations. Take the value off the top of the stack, negate it, and put it back. To test, `return -(42);`—the parentheses are necessary; try it without them to see why.

- addition and subtraction: Two more fairly simple stack operations are next. Addition and subtraction take two values, perform the appropriate operation, and push the result back onto the stack. Be sure to get the order right for subtraction. You can test by using the return statement, *e.g.*, `return 10+(40-8);`.

- string constants: Handle string constants in the same way that you handled integer constants. Remember that the address of the first character of the string should be pushed.

- function calls: Function calls will take a bit of work. First, note that the arguments are given to you as they appear from right to left in the C code. Once you've pushed them all, use a JSRR (again, see the long branch code provided) to call the library, then clean up the arguments, copying the return value to be the only thing that you leave on the stack.

  Now you can try the classic first C program: `printf ("Hello, world!\n");`

- scalar variable reference: Write the code to read a scalar variable and push it onto the stack. You'll need to look the variable up in the symbol table to know where to find it. For now, you may assume that the variable is not an array (or get a headstart on the second checkpoint by handling both kinds).

- scalar variable assignment: Now write the code to handle an assignment to a scalar variable. Again, for now you may assume that you don't need to handle assignments to array elements.

- pop stack statement: A C expression followed by a semicolon is a statement (which is not as useless as it may sound, since "a=10" is an expression). The expression leaves a value on the stack, which must be discarded to finish the statement. All you need to do is pop the stack.

Once you have completed these tasks, your compiler will be able to handle assignments and return statements using expressions involving addition, subtraction, negation, and scalar variables. You can also print out results using calls to `printf`. Try a few different programs to check that things work properly.

Second Half

You must also complete the following tasks, although we suggest that you get the first half working before undertaking these tasks.

- multiplication, division, and modulus: Start with a few more binary operators. For these, rather than implementing the operations yourself, simply call (using JSRR) to the C library provided to you.

- array variable reference: Now go back to your variable reference code and handle accesses to array variables. Remember that the offset in the symbol table is to the start of the array, so you'll need to add in whatever index is provided in the C code (it's an expression that must be evaluated).

- array variable assignment: Again, go back to your assignment code and handle assignments to array variables.

  If you haven't already done so, try compiling some code that uses your new operators as well as arrays to check that you are generating code correctly. Be sure to try both local and global variables.

- increment and decrement: These operators come in four varieties and require you to read a variable (possibly an array entry), modify it, write it back, and return a result on the stack. Be careful and test them all before moving on.

- for stmt: If you don't remember how the `for` loop works, you may want to look it up in the book or your notes. Note that the initialization and update AST nodes are statements in our compiler so that you don't have to pop the results from the stack in `gen_for_statement`. The test condition is a regular expression, since you'll have to test it after producing it. The body is a linked list of statements. Put them all in the right order, remember that anything non-zero is true in C, and use long branches in case the loop body is long.
- get variable address: For the `scanf` call, you'll need to be able to pass variable addresses. Write the function that finds the address of a variable, be it a scalar or an array element, and pushes it onto the stack.
- if statement: As with the `for` statement, you may want to look up the LC-3 implementation of the conditional decomposition before implementing this statement. The test is an expression, and the then and else blocks are linked lists of statements, either of which may be empty (but you don't need to optimize your code in such cases).
- logical operations: The logical NOT, OR, and AND operations are somewhat more complicated than the operators that you have implemented already. First, they always result in 0 or 1, and you will have to make sure that you generate code to place the right value on the stack. Next, both OR and AND require shortcutting, which means that the second expression is only evaluated if the first allows it to change the outcome of the logical operation. For example, if the first operand of OR is true, the second operand is not evaluated.

Grading

Details to be determined, but roughly 85% on functionality, and 15% on comments and documentation.