

# CS4481

## Project 7 - Compiler

In this project you will implement a full compiler for the C- language. You can choose to start with the code formatter you wrote in project 6 or you can use the starter code provided here (it is *strongly* recommended to use the provided starter code). Then you will write Java code to generate MIPS assembly code. We will not be implementing a compiler for C- in its entirety. See the scoring section below for a list of features we will support. Additional features can be implemented for extra credit.

### A note on activation records

The provided starter code places the return value of the function as the first value in the activation record. Formal parameters, then register saves, then local variable declarations follow.

### Example 1

Consider the following C- code:

```
int add(int x, int y) {
    return x + y;
}

void main() {
    int a, b, c;
    add(3, 4);
}
```

The code would result in something like the following in memory where `r` is the return value for `add()`. `$sp` is the stack pointer when `add()` is called.

```

                                     <--- stack|
|           <--- add() --|--      main()  --|
|           ...$s0 x y r |         ... c b a |
|                                     ^
|                                     |
|                                     $sp
```

In the above example there are four symbol tables. One for `main`'s parameters, one for `main`'s compound statement, one for `add`'s parameters, and one for `add`'s compound statement. *Each*

variable offset is stored relative to its containing symbol table. So `b` has an offset of -8 and `x` has an offset of -12. Then the question becomes, how do we access `b` from inside of `add()`?

Take a look at `SymbolTable.find()`. This method looks for a variable and returns a `SymbolInfo` object. However, note that it does not necessarily return the local `SymbolInfo`, but rather, may return a new `SymbolInfo` with the address offset that takes into account where the stack pointer is.

To make this concrete, consider the example above. When inside `add()` I want to access `b`. The local offset of `b` is -8 (as if `$sp` was pointing at `main()`). But with `$sp` pointing to `add()`, we need to add the size of `main()`'s activation record to the offset. Suppose the activation record for `main()` is size 24. Then the true offset for `b` when inside `add()` is  $-8 + 24 = 16$ . This is the value that is returned by `SymbolTable.find()`.

## Example 2

Consider another example:

```
int fee() {
    int a;
    a = 3;
    {
        int b;
        {
            int c;
            c = a + 4;
            println(c);
            return c;
        }
    }
}

main() {
    fee();
}
```

In the inner-most compound statement, the stack might look something like this:

```

                                     <--- stack|
|           | | <--- fee()  --|-- main() --|
|           c | b |  $s1 $s0 a r |    ...    |
|           ^
|           |
|           $sp
```

Again, `$s1` and `$s0` are registers that are stored temporarily on the stack and `r` is the location of the return value. Note that we have two unnamed “activation records” corresponding to the two nested compound statements. Variable offsets are computed as described in the previous example. That is, the offset of `a` would be  $-8 + 16 + 4 = 12$  where -8 is the local offset of `a`, 16 is the size of the `fee()` activation record, and 4 is the size of the first unnamed activation record.

Another issue to consider is return statements. The return statement needs to know two things: where on the stack to place the return value, and where on the stack to assign `$sp`. Both of these can be accomplished by calling `SymbolTable.returnValueOffset()`. In example 2, when the `return` statement is made, the return value offset is  $-4 + 16 + 4 = 16$ . You can then determine where to restore `$sp` to by adding the size of the return value.

Note that if a function returns `void`, `SymbolTable.returnValueOffset()` may return 0 or the offset to the function's activation record base address. This shouldn't require any special handling.

## Tasks

1. Download the QtSpim simulator if you haven't already. Get familiar with the software using the assembly files you wrote for homework 4.
2. Use the starter code in the downloaded project `src` directory. **You will be running from the "Compiler.java" class, not "Formatter.java" as in project 6.**
3. You should be able to run your starter code on **data/test1.c** without errors.
4. The best approach is to add a method called `toMIPS()` to the `CminusElement` interface, much like `toCminus()`. This has been done for you in the starter code. It should be something similar to the following:

```
/**
 * Emits MIPS code.
 *
 * @param code Use this builder to emit code to the .text section of the MIPS
 * file.
 * @param data Use this builder to emit code to the .data section of the MIPS
 * file.
 * @param symbolTable Keeps track of variable and function symbols.
 * @param regAllocator Allocates registers on demand. Each C- function should
 * use fresh registers.
 * @return The result of emitting MIPS code. See comments in EvalResult.
 */
public EvalResult toMIPS(StringBuilder code, StringBuilder data,
    SymbolTable symbolTable, RegisterAllocator regAllocator);
```

You can initially implement your new method in each AST node as follows:

```
@Override
public EvalResult toMIPS(StringBuilder code, StringBuilder data,
    SymbolTable symbolTable, RegisterAllocator regAllocator) {
    builder.append("# BinaryOperator not supported");
    return EvalResult.createVoidResult();
}
```

By doing this you will always output valid MIPS code (your stub code is commented out with the `#` character).

5. The provided starter code includes implementations of `toMIPS()` for the following classes: `BoolConstant`, `ExpressionStatement`, `FunDeclaration`, `ParenExpression`, and `Program`. The implementation in `FunDeclaration` and `Program` should be particularly helpful in getting started.
6. The starter code writes MIPS code both to `stdout` and to a file called **Cminus.asm**.
7. In the starter code you will need to handle both `println` and calls to functions defined in the code. This should be handled in `Call.toMIPS()`. I recommend that you implement the `println()` function first. That way, you can use `println()` to test the rest of your code. The function `println()` takes one argument: a variable, char constant, bool constant, int constant or string constant.
8. Note that `FunDeclaration` and `CompoundStatement` both have a `SymbolTable` as a data member. The symbol table was added when the node was constructed. This will be helpful as you emit MIPS code because the symbol table not only stores symbols and

their locations in memory (as offsets from `$sp`) but also stores the activation record size. *Important:* in the starter code, entering a function enters two nested symbol tables: one for the function parameters and one for the compound statement containing the function's implementation.

9. In `toMIPS()`, you may not need to comment much of your Java code if you emit MIPS comments that explain what is going on.
10. If you do any extra credit work (see below), then include a **README.txt** file explaining what you did. Please also include a test file or two.
11. C- doesn't support function overloading. Every function must have a unique name.

## Submission

When you are done, zip your entire project into `project7.zip` and submit using Moodle.

## Scoring

1. 10% - `println`
2. 10% - Basic arithmetic using numbers
3. 10% - Variables
4. 30% - Function calls
5. 30% - Parameters
6. 10% - Coding quality and style

Extra credit:

1. 3% - `if`
2. 5% - `while`
3. 2% - char data types
4. 10% - arrays
5. 2% - relational operators
6. 2% - unary relational expressions
7. 4% - complex assignments (`+=`, `-=`, `*=`, `/=`, `++`, `--`)