# CS4212 Programming Assignment 3 Report

JULIUS PUTRA TANU SETIAJI (A0149787E)

<div align="center">CONTENTS</div>

## 1 GENERAL COMPILATION FLOW

Besides IR3, I defined my own machine-specific intermediate representation (IR) for use in my compiler, called Lower. With reference to GCC, IR3 is similar to GIMPLE, while Lower is similar to RTL.

My compiler can be divided into several parts:

(1) Lexer
(2) Parser
(3) Typechecker
(4) IR3 Codegen
(5) Lower Codegen
(6) Optional optimisation passes
(7) Register Allocation
(8) ARM code generation
(9) Optional peephole optimisation passes

### 1.1 Lexer and Parser

Lexer and parser were done as part of Programming Assignment 1.

Lexing is done using the lexer generator JFlex. It was relatively straightforward as the tokens in JLite is a subset of the tokens in Java, and JFlex provided a sample Java lexer. The one difference between Java tokens are JLite tokens is the liberty that I took to disallow representing the empty string as `""`, and instead forcing the use of `null` to represent empty string.

Parsing is done using the LALR parser generator CUP. This part is more complicated as the provided grammar is ambiguous, due to the type-specificity of each operator, the overloading of the operator + to represent both arithmetic addition and string concatenation, and the associativity ambiguity of the string concatenation operator. Thus, I specified a less strict grammar rules, and verified the abstract syntax tree during construction in order to make sure it follows the full JLite grammar rules.

The generated abstract syntax tree is then passed on to the typechecker.

## 1.2   Typechecker and IR3 Codegen

Typechecker and IR3 code generation were done as part of Programming Assignment 1.

Typechecking is performed on the abstract syntax tree recursively, and the types are attached to the abstract syntax tree. Generating IR3 from the abstract syntax tree generated is quite trivial through recursion and with the type information yielded from the typechecker.

The generated IR3 code is then passed on to the Lower codegen.

## 1.3   Lower Codegen

Lower is my machine-specific where I perform all the passes on. Unlike Jlite and IR3, Lower does not have any notion of expression, only statements. These statements are designed to very closely match ARM instructions, while still maintaining IR3 semantics as much as possible. Lower also allows statements to make use of real physical hardware ARM registers, besides variables. The struct `Addressable` in Lower is designed to be the representation of a "register", both real physical hardware registers as well as pseudo-registers (which program variables are).

There are many types of statements in Lower:

- `BINARY`, representing instructions in ARM whose operands are one register and one Operand2 (either Immediate or Register without barrel-shifter operation).
- `BINARY_BIT`, representing instructions in ARM whose operands are one register and one Register with barrel-shifter operation.
- `BIT`, representing barrel-shifter instructions in ARM.
- `BRANCH_LINK`, representing the BL instruction.
- `CMP`, representing an if-goto instruction (CMP followed by conditional B instructions).
- `FIELD_ACCESS`, representing a field access (LDR instruction).
- `FIELD_ASSIGN`, representing a field assignment (STR instruction).
- `GOTO`, representing unconditional branching (unconditional B instruction).
- `IMMEDIATE`, representing assignment into a register from an immediate value (either MOV instruction with an immediate operand2, or LDR from a constant pool, as is the case for literal strings).
- `LABEL`, representing a label.
- `LOAD_STACK_ARG`, representing LDR instruction from the stack for a function argument.
- `LOAD_LARGE_IMM`, representing assignment into a register from the constant pool for a large immediate value.
- `LDR_SPILL`, representing LDR instruction for a spilled variable.
- `STR_SPILL`, representing STR instruction for a spilled variable.

- MOV, representing assignment from one register into another (MOV instruction with two register operands).
- RETURN, representing returning from the current function (either BX LR instruction or LDMFD SP!, {..., PC}).
- PUSH_PAD_STACK, representing SUB SP, SP, #4 to pad the stack by 4 bytes to maintain 8-byte alignment as required by the ARM calling convention.
- PUSH_STACK, representing PUSH pseudo-instruction (STR Rs, [SP, #-4]!) to push a register into the stack, which also updates the SP register.
- POP_STACK, representing popping the stack (ADD SP, SP, #<number of bytes>).
- REG_BINARY, representing instructions in ARM whose operands are 2 registers (MUL and SDIV).
- REVERSE_SUBTRACT, representing the RSB instruction whose operand2 is either an Immediate value or Register without barrel-shifter operation.
- REVERSE_SUBTRACT_BIT, representing the RSB instruction whose operand2 is a Register with barrel-shifter operation.
- UNARY, representing either the MVN instruction or NEG pseudo-instruction (RSB Rd, Rs, #0).

There could be some IR3 operations that do not fulfill the requirements of any Lower statement, e.g. a = b * 3;. The pass that lowers IR3 to Lower will create new temporaries as required. The purpose of Lower is to make ARM code generation trivial given register allocation information.

new statements, println statements readln statements, and string concatenation operations are also lowered into either a call into a helper function, or the corresponding libc functions:

- **new** statements are implemented by calling malloc with the correct number of bytes based on the number of fields of the class. The types of the fields do not matter as all types in JLite are uniformly 32 bits (4 bytes):
  - String is represented as a pointer to the first element of a null-terminated array of characters. Since ARM is an ISA for a 32-bit machine, pointer lengths are 32-bits.
  - Int is represented as a 32-bit integer as we are using a 32-bit machine.
  - Bool is represented as an Int whose value is 1 for true, and 0 for false.
  - Classes are represented as a pointer, which in a 32-bit machine is 32-bits long.
- **println** statements are implemented by:
  - String: calling puts(string);.
  - Int: calling printf("%d", integer);.
  - Bool: calling helper function println_bool(**bool**);.
- **readln** statements are implemented by:
  - Int and Bool: calling readln_int_bool(); and using the result.
  - String: calling getline_without_newline(); and using the result.
- **String concatenation** is implemented by using strlen to find out the lengths of the two string, then malloc to allocate a buffer for the resulting string, then strcpy from the first string to the resulting string, followed by strcat from the second string to the resulting string.

Note that since we do not have a garbage collector, currently memory allocated for strings and classes are just leaked when they go out of scope.

Helper function println_bool(**bool**); prints either true or false depending on the boolean value. The implementation in C and ARM:

```
1  void println_bool(int a)
2  {
```

```
3    puts(a == 1 ? "true" : "false");
4  }
```

```
1  println_bool:
2  LDR R2, .TRUE
3  LDR R1, .FALSE
4  CMP R0, #1
5  MOVEQ R1, R2
6  MOV R0, R1
7  b puts
```

Helper function `readln_int_bool()`; gets one whole new line and read an integer from it. The implementation in C and ARM:

```
1  int readln_int_bool()
2  {
3    int a;
4    char* result = NULL;
5    size_t n = 0;
6    getline(&result, &n, stdin);
7    sscanf(result, "%d", &a);
8    free(result);
9    return a;
10  }
```

```
1  readln_int_bool:
2  STMFD SP!, {R4, LR}
3  SUB SP, SP, #16
4  MOV R0, #0
5  STR R0, [SP, #8]
6  STR R0, [SP, #4]
7  LDR R0, .Lstdin
8  LDR R2, [R0]
9  ADD R0, SP, #8
10  ADD R1, SP, #4
11  BL getline
12  LDR R0, [SP, #8]
13  LDR R1, .PERCENTD
14  ADD R2, SP, #12
15  BL sscanf
16  LDR R0, [SP, #8]
17  BL free
18  LDR R0, [SP, #12]
19  ADD SP, SP, #16
20  LDMFD SP!, {R4, PC}
```

Helper function `getline_without_newline()`; gets one whole new line and removes the newline from it. The implementation in C and ARM:

```
1  char* getline_without_newline()
2  {
3    char* result = NULL;
4    size_t n = 0;
5    ssize_t len = getline(&result, &n, stdin);
6    result[len - 1] = 0;
```

```
7    return realloc(result, len - 1);
8  }
1  getline_without_newline:
2  STMFD SP!, {R4, LR}
3  SUB SP, SP, #8
4  MOV R4, #0
5  STR R4, [SP]
6  STR R4, [SP, #4]
7  MOV R0, SP
8  ADD R1, R0, #4
9  LDR R3, .Lstdin
10 LDR R2, [R3]
11 BL getline
12 SUB R1, R0, #1
13 LDR R0, [SP]
14 STRB R4, [R0, R1]
15 BL realloc
16 ADD SP, SP, #8
17 LDMFD SP!, {R4, PC}
```

### 1.4 Optional Optimisation Passes

There are a couple of optimisations that I have implemented on the Lower IR. I will expound on this in a later section.

### 1.5 Register Allocation

Register allocation is performed on the Lower IR based on graph colouring of the interference graph, created using liveness dataflow analysis. Performing graph colouring based on the Lower IR rather than IR3 allows for more optimal register allocation, such as using R0 to R3, as well as R12 and LR (as long as LR is saved) for variables that are not live across function calls. R12 is not safe to use across function calls as it is used as Intra-Procedure-call scratch register.
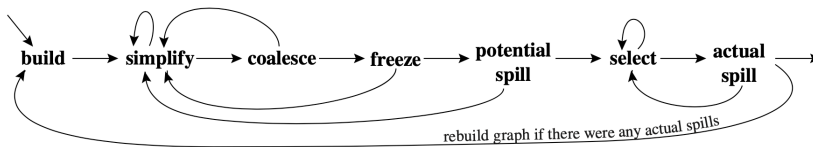


Fig. 1. Chaitin-Briggs-George-Appel register allocation algorithm based on graph-colouring

The register allocation method used is Chaitin-Briggs-George-Appel algorithm based on Kempe's graph-colouring algorithm. The steps are as can be seen from Figure 1:

- **Build**: construct the interference graph, and categorise each pseudo-register as either move-related or non-move-related.
- **Simplify**: one at a time, remove non-move-related nodes of low-degree from the graph, similar to Kempe's graph-colouring algorithm.
- **Coalesce**: perform conservative coalescing on the reduced graph obtained in the simplification phase.

- **Freeze**: if neither simplify nor coalesce applies, freeze all the low-degree move-related nodes – this means that they are considered to be non-move-related, which enables more simplication. After this, simplify and coalesce are resumed
- **Potential Spill**: If there are no low-degree nodes, select a significant-degree node for potential spilling and push it on the stack.
- **Select**: Pop the entire stack, assigning colours if possible.
- **Actual spill**: If there is a node for which no colour can be applied during select, it is an actual spill. The program is rewritten with the spill, and the whole process is started from scratch.

Spilling priority is determined by the spilling priority score, where lower scores are more likely to be spilled. The heuristic used is score = (number of defs and uses) ÷ degree. This is because we want to spill variables that are least frequently defined and used, as well as those that interferes with many other variables. Temporaries created in IR3/Lower is given higher scores than user-defined variables, and spilled temporaries are given even higher scores then temporaries created in IR3/Lower.

Note that some operands of Lower statements are actual physical hardware registers, which means that they are precoloured in terms of the graph colouring problem. Using conservative coalescing, these registers can be coalesced with other pseudo-registers, resulting in a more optimal register allocation. Registers R0 to R3 are also used in the calling convention. Since they are caller-saved, their contents are clobbered after a function call. This can be indicated in the interference graph by making these registers interfere with one another, as well as with variables that are live across function calls. Care should also be taken to ensure short live-range of physical hardware registers, as they cannot be spilled into memory. As such, the standard convention call is represented in Lower using `MOV` and `IMMEDIATE` statements to move the parameters into registers R0 to R3, as well if need be pushing the rest of the parameters into stack, then a `BRANCH_LINK` statement, followed by a `POP_STACK` statement if there were parameters on the stack, and finally, if need be, a `MOV` statement to move the result of the function call to a variable if the result is used.

For example, take the following JLite statements (with the class signature in the comment):

```
1  /* class Func {
2   *   Func f(Int, Int, Int, Int, Int, Int);
3   *   Void g();
4   * }
5   */
6  Func a; Int b; Int c; Int d; Int e; Int f;
7  a.f(b, 5, c, d, e, f).g();
```

Lowered to IR3, the equivalent statements are:

```
1  Func _t1;
2  _t1 = %Func_0(a, b, 5, c, d, e, f);
3  %Func_1(_t1);
```

Lowered to Lower, the equivalent statements are:

```
1  R0 <- a;     // MOV
2  R1 <- b;     // MOV
3  R2 = 5;      // IMMEDIATE
4  R3 <- c;     // MOV
5  PUSH (PAD);  // PUSH_STACK_PAD
6  PUSH f;      // PUSH_STACK
7  PUSH e;      // PUSH_STACK
8  PUSH d;      // PUSH_STACK
```

```
9    CALL Func_0; // BRANCH_LINK
10   POP 4;       // POP_STACK
11   _t1 <- R0;   // MOV
12   R0 <- _t1;   // MOV
13   CALL Func_1; // BRANCH_LINK
```

Assuming that a is assigned to register R4, b to register R1, c to register R5, d to register R6, e to register R7, f to register R8, g to register R0, then the generated ARM code is:

```
1    MOV R0, R4
2    MOV R2, #5
3    MOV R3, R5
4    SUB SP, SP, #4
5    STR R8, [SP, #4]!
6    STR R7, [SP, #4]!
7    STR R6, [SP, #4]!
8    BL Func_0
9    ADD SP, SP, #16
10   BL Func_1
```

Observe that MOV to the same registers are omitted. With the liveness information used to construct the interference graph, the register allocator is able to tell, e.g. that variables b and _t1 do not live across function calls. Thus, the register allocator is able to make a more optimal choice of assigning b to R1, given that it is the second parameter to Func_0, while _t1 is assigned to R0 as it is the first parameter of Func_1.

## 1.6   ARM Code Generation

With register allocation from the register allocator, ARM code generation from Lower IR is relatively trivial, as most types of Lower statements correspond with exactly one ARM instruction, except for CMP, and comparison operators in BINARY and BINARY_BIT.

## 1.7   Optional Peephole Optimisation Passes

There are a couple of optimisations that I have implemented on the ARM code generated. I will expound on this in a later section.

## 2   LOWER IR OPTIMISATIONS

A couple of optimisations are performed on the Lower IR. These optimisations are run continuously until 2 runs produce the same Lower IR.

## 2.1   Algebraic Identities

We make use of the following algebraic identities:

- $x + 0 = 0 + x = x$ and $x - 0 = x$.
- $x \land true = x$
- $x \land false = false$
- $x \lor true = true$
- $x \lor false = x$

## 2.2   Deadcode Elimination

There are a couple of dead code types that are detected in this pass:

- **Basic Block**: basic blocks with in-degrees = 0 except for the entry node is removed.
- **Uses**: statements that defines dead variables are removed.

- **Dead Gotos**: delete goto statemens pointing to a label that follows the goto statement directly, i.e. GOTO L2; L2:, and rather use the fallthrough behaviour.
- **Mov to itself**: delete mov statements with the same source and destination.
- **Remove useless labels**: remove labels that are not the destination of any goto or if-goto statements.
- **Remove double goto**: remove labels whose content is another goto statement, and ensuring all gotos to the removed label go to the destination of the goto statement in the removed label.
- **Remove label return**: remove labels whose only content is return, in favour of replacing the original goto to the removed labels to return directly.

## 2.3 Barrel-shifter Operand2

Replace the sequence of BIT followed by BINARY with BINARY_BIT, and BIT followed by REVERSE_SUBTRACT with REVERSE_SUBTRACT_BIT. If the destination of the BIT statement is not used anymore anywhere else, then the **Deadcode Elimination** pass will pick it up to remove the BIT statements.

## 2.4 Constant Propagation and Constant Folding

Using reaching definition dataflow analysis, resolve the value of variables into a constant. If all operands in a statement is identified, then perform constant folding: do the operation during compile time instead and turn the statement into an IMMEDIATE statement with the result of the operation.

This works wonderfully on the provided test_booleans.j and test_ops.j sample input files provided, and we are left only with print statements containing the results.

There are 2 additional passes performed on the resulting constant-propagated and constant-folded program:

- **Handling large immediate**: operand2 immediate values are limited to 8-bit values rotated by an even number of bits. Any immediate outside of that range must be loaded from a constant pool. This is handled by replacing those with LOAD_LARGE_IMM Lower statement.
- **Inlining of println_bool**: if the parameter to println_bool is constant-folded and known at compile-time, then inline the helper function to either puts("true"); or puts("false");.

## 3 ARM CODE PEEPHOLE OPTIMISATIONS

There are a couple of peephole optimisations that can be performed on the resulting ARM code.

## 3.1 Algebraic Identities

*3.1.1 Addition and subtraction with zero.* Replace ADD R0, R1, #0 or SUB R0, R1, #0 with MOV R0, R1.

*3.1.2 And with true.* Replace AND R0, R1, #1 with MOV R0, R1.

*3.1.3 And with false.* Replace AND R0, R1, #0 with MOV R0, #0.

*3.1.4 Or with true.* Replace ORR R0, R1, #1 with MOV R0, #1.

*3.1.5 Or with false.* Replace ORR R0, R1, #0 with MOV R0, R1.

## 3.2 Useless MOV

Replace:

```
1   MOV R5, R0
2   MOV R0, R5
```

with:

```
1   MOV R5, R0
```

Note that we cannot remove `MOV R5, R0` entirely as the value of R5 might be used later on, for example, to ensure a variable survives a function call.

### 3.3   Useless LDR

Replace:

```
1   STR R0, [SP]
2   LDR R0, [SP]
```

with:

```
1   STR R0, [SP]
```

Note that we cannot remove `STR R0, [SP]` entirely as that value in memory might be loaded later on.

### 3.4   Remove MOV to itself

Delete:

```
1   MOV R0, R0
```

### 3.5   Consecutive MOV of immediate values to the same register

Replace:

```
1   MOV R0, #5
2   MOV R0, #1
```

with:

```
1   MOV R0, #1
```

## 4   ADDITIONAL NOTES

Possible future work includes:

- Sub-expression elimination, which is probably best done on the IR3 level.
- Better spilling heuristic by changing the weight of the number of defs and uses inside a loop as opposed to outside a loop, for example $score = ((\text{number of defs and uses outside loops}) + 10 \times (\text{number of defs and uses inside loops})) \div degree$.
- Converting into Single Static Assignment form to unlock further optimisation, for example with sparse conditional constant propagation.
- Performing