CS4212 Programming Assignment 3
Bernard Teo Zhi Yi

[Note: I have submitted both the markdown (.md) file and the equivalent HTML and PDF file containing the same information. You can use either one, but the HTML version has the best formatting.]

Introduction

This submission builds on the parser from assignment 2, and is a fully-featured compiler from Jlite to ARM assembly. The assembly code can then be assembled using GCC's assembler, and then run using gem5. There are no known bugs — all Jlite programs are compiled correctly.

When optimisations are enabled, my compiler does liveness analysis on the control flow graph of each function, and builds an interference graph, on which a graph colouring algorithm is used to determine register allocation. The graph colouring algorithm has been modified to give "preferences" for certain register assignments that minimises moves, such as preferring that function parameters are placed in registers a1-a4, and a variable being returned is placed in register a1. This ultimately makes the assembly code more optimised.

The generated code is carefully prepared so that the register allocator can make use of as many registers as possible. In particular, the registers a1-a4, v1-v7, fp, and lr, may be assigned to variables (the allocator ensures that a1-a4 and lr are not used to store variables across a function call). Note that the fp register is treated as a general-purpose register in my compiler — all local variables on the stack have offsets calculated from sp, which is not affected by fp. This still conforms to the ARM calling convention, so we can still call functions from the C standard library (like `malloc`).

I also implemented the full Jlite spec, **including readln**, so programs can read from standard input. There are some test cases that use this feature.

Division is implemented with a long division routine, and works for all inputs (both positive, negative, or zero). String concatenation is also implemented with a routine that calls `malloc` and then copies the string content into the new heap memory that is allocated.

Overloaded methods were implemented in assignment 2, and they work fully in assignment 3 as well.

Content of the submission

The `src` directory contains all the files of my submission, including appropriate versions of the jflex and cup binaries.

Subdirectories:

- `src` : code that drives the compiler
- `src/tree` : classes to represent the syntax tree of the program
- `src/ir3` : classes to represent the IR3 code, code generation, register allocation, and type-checking functions
- `src/util` : some utility classes for error reporting
- `src/test` : contains tests
- `src/test/errors` : contains tests that are semantic errors

- `src/test/asg1` : contains tests from assignment 1 (you can ignore this)
- `src/test/asg2` : contains tests from assignment 2 (you can ignore this)
- `src/testcases` : symlink to `src/test` , to satisfy assignment 3 submission requirements
- `src/jflex` : contains jflex spec and jflex binaries
- `src/cup` : contains cup spec and cup binaries

Important!: Before continuing any further, you must go into the directory first:

```
cd src
```

Compile

Firstly, we need to chmod the java archives for jflex and cup, and the scripts:

```
chmod u+x cup/*.jar
chmod u+x jflex/jflex-1.8.2/lib/*.jar
chmod u+x *.sh
```

Then type `make` , like this command:

```
make
```

Then will take a while to compile. Be patient, and it should be done after a while.

Compile Jlite code

If you just want to see the ARM assembly output of my compiler, you can read this section that describes how to compile code with my compiler. If you also want to run the code, go to the next section instead.

You can use the `compile.sh` and `compile-opt.sh` scripts, which will use the specified Jlite source file from the test folder. It will compile the Jlite file, then invoke the assembler, then use gem5 to run the binary, and display the output.

For example to run the `test/default.j` file, you can do the following.

For the **unoptimised** version:

```
./compile.sh default
```

For the **optimised** version:

```
./compile-opt.sh default
```

Prerequisites for running code

To run the tests, you need to have these installed:

- the GCC ARM cross-compiler (`gcc-arm-linux-gnueabi` , the current version from the Ubuntu package manager is gcc 9, but others should also work), from the default package manager (used for the assembler only)
- gem5, installed somewhere on your machine

The GCC cross-compiler should be invokable using the `arm-linux-gnueabi-gcc` command (see the `run.sh` script for details).

You must also define the `GEM5_DIR` environment variable to the location of your gem5 installation (without trailing "/"). If not specified, the default location used is `$HOME/Documents/gem5` (see the `run.sh` script for details).

You need both prerequisites in the following sections.

Run

You can use the `run.sh` and `run-opt.sh` scripts, which will use the specified Jlite source file from the test folder. It will compile the Jlite file, then invoke the assembler, then use gem5 to run the binary, and display the output.

For example to run the `test/default.j` file, you can do the following.

For the **unoptimised** version:

```
./run.sh default
```

For the **optimised** version:

```
./run-opt.sh default
```

Note that the unoptimised version simply places all the local variables on the stack. To use the better register allocation (with liveness analysis and interference graph), you have to invoke the optimised version.

Remember that you do not add the file extension of the Jlite file to the command.

Alternatively, you may type in the command manually (see the content of `run.sh` for the exact syntax).

The difference between the unoptimised and optimised version is the `-O` flag for the compiler. The default mode is unoptimised, and you have to pass the `-O` flag to use the optimised version. The scripts will set the flag for you, so you don't need to worry about it when using `run.sh` / `run-opt.sh`.

Tests

All the tests are in `src/test`.

There are 7 tests (apart from the default).

Each test is made up of the following files, where `xxx` is the name of the test:

- `xxx.j` — Jlite source code
- `xxx.s` — expected assembly output when compiled with the **unoptimised** version
- `xxx-opt.s` — expected assembly output when compiled with the **optimised** version
- `xxx.out` — expected standard output of the compiled program (both the unoptimised and optimised versions should agree)
- `xxx.in` (optional) — standard input for the compiled program (only present for tests involving `readln()`)

To run all the tests using the **unoptimised** version, do:

```
./test.sh
```

To run all the tests using the **optimised** version, do:

```
./test-opt.sh
```

In either case, you should get an output like this:

```
Testing test/call ... Assembly matches ... Program output matches
Testing test/default ... Assembly matches ... Program output matches
Testing test/divide ... Assembly matches ... Program output matches
Testing test/list ... Assembly matches ... Program output matches
Testing test/operator ... Assembly matches ... Program output matches
Testing test/readln ... Assembly matches ... Program output matches
Testing test/recursion ... Assembly matches ... Program output matches
Testing test/sum ... Assembly matches ... Program output matches
```

The `default` test is the one that came with the assignment.

Brief explanation of each test

- `call` — tests function calls and calling convention, including spillage of excess parameters onto the stack
- `default` — the test that came with the assignment
- `divide` — tests for integer division, which uses a hand-coded long division routine
- `list` — a simple linked list library to test objects and allocation, as well as while-loops
- `operator` — exhaustively tests all unary and binary operators
- `readln` — tests the readln operation
- `recursion` — tests recursion, as well as if-statements
- `sum` — reads the input for a single integer n, and outputs the sum 1+2+...+n; it tests loops and optimisations

Brief overview of the compiler

The following is a brief overview of how the compiler works.

We start with IR3 code (that was produced from assignment 2).

The main complication in assignment 3 is register allocation, which is described in the next section.

After register allocation is done, the appropriate assembly code for each IR3 statement is emitted, which produces the ARM assembly file.

We also transform the main function because Jlite returns `Void` from the main function, but we need it to `return 0` in assembly. We do a simple transformation at IR3-level by replacing each `return` statement with `return 0` and adding `return 0` to the end of the main function.

Register allocation (+ optimisation)

Each function is compiled separately, and some global bookkeeping is used to store the string literals and labels. For each function, liveness analysis is carried out, like in the lectures. We do liveness analysis directly at the statement level (instead of basic blocks) so that we can very precisely determine if two variables interfere (i.e. have overlapping live ranges). From the liveness analysis, the interference graph is built. The interference graph is "exact" in the sense that two variables that are detected to interfere must definitely actually interfere.

The interference graph is somewhat augmented to specify that variables that live across particular operations (such as function calls) cannot use certain registers. Specifically, any variable that lives across a function call cannot be placed in a1-a4 or lr.

The interference graph also records "**preferences**" of certain variables for certain registers. For example, the first four arguments of a function will prefer to be in a1-a4 (so that we can elide a `mov` instruction where possible), and return values will prefer to be in a1. Such preferences also apply in call-expressions/statements (arguments prefer to be in a1-a4), `new Object()` statements (the variable that stores the new object prefers to be in a1), and a number of other situations. My register allocation algorithm tries to select the preferred registers whenever possible, so as to minimise `mov` instructions.

Specifically, in the graph colouring algorithm, we try to remove variables with no preference from the graph **first**, so that when in the colouring stage, those with preferences get to pick their colours before those without preferences. This ensures that we adhere to the register preferences of variables whenever possible.

(Remember that you need to compile/run the **optimised** version of the compiler to use this register allocation algorithm.)

If there are too many variables, then excess variables are spilled onto the stack. To comply with the ARM calling convention, the stack pointer is always maintained at an 8-byte alignment.

You may look at the optimised assembly samples in the test folder (`xxx-opt.s` where `xxx` is the test name) to see that redundant `mov` instructions are indeed minimised.

As an example, we have the following source code:

```
Int size() {
  Int x;
  Node curr;
  x = 0;
  curr = head;
  while (!curr.last) {
    x = x + 1;
    curr = curr.next;
  }
  return x;
}
```

The **unoptimised** assembly code is:

```
$List$size$$$List:
stmfd sp!,{a1,fp,lr}
sub sp,sp,#12
mov lr,#0
str lr,[sp,#8]
ldr lr,[sp,#12]
ldr lr,[lr,#0]
str lr,[sp,#4]
.L0:
ldr lr,[sp,#4]
ldrb lr,[lr,#8]
```

```
strb lr,[sp,#0]
ldrb fp,[sp,#0]
eor fp,fp,#1
cmp fp,#0
bne .L1
b .L2
.L1:
ldr fp,[sp,#8]
mov lr,#1
add lr,fp,lr
str lr,[sp,#8]
ldr lr,[sp,#4]
ldr lr,[lr,#0]
str lr,[sp,#4]
b .L0
.L2:
ldr a1,[sp,#8]
add sp,sp,#16
ldmfd sp!,{fp,pc}
```

The **optimised** assembly code is:

```
$List$size$$$List:
stmfd sp!,{lr}
sub sp,sp,#4
mov a2,#0
ldr a3,[a1,#0]
.L0:
ldrb a1,[a3,#8]
cmp a1,#0
beq .L1
b .L2
.L1:
add a2,a2,#1
ldr a3,[a3,#0]
b .L0
.L2:
mov a1,a2
add sp,sp,#4
ldmfd sp!,{pc}
```

As can be seen, in the unoptimised version, all the local variables are placed on the stack. However, in the optimised version, they are all placed in registers, and the registers are chosen to eliminate as many `mov` instructions as possible.

Features

Some of the more interesting features of my compiler are highlighted here, in the hope of gaining me more marks :)

Of course, my compiler does produce correct ARM assembly for all valid Jlite programs, but that should be expected from all compilers. This section is for the more interesting features.

Strings

Strings are encoded in a somewhat unique way in my compiler. A string of length n is stored in an array of $4 + n$ bytes, where the first 4 bytes encodes the string length. The string is **not** null-terminated.

For example, the string "abcdefg" is stored as a 11-byte array "\007\000\000\000abcdefg", which is not null-terminated.

Strings are stored this way so that we can determine the size of the string in constant time, which is more efficient when doing string operations. For example, we can do string concatenation in a single pass with this encoding, but not with the C-style null-terminated strings.

String concatenation

While string concatenation is a single IR3 instruction, it expands out to a moderately long sequence of instructions in ARM assembly, since we need to call `malloc` to allocate heap memory for the new string, then copy the data from the two existing strings into the new string. Such "routines" have to be written carefully and the registers that are clobbered are made known to the register allocator, so that variables that live past a string concatenation operation do not use certain registers.

String pooling

If there are multiple identical string literals in the data section of the executable program, they will be merged, so as to reduce the size of the executable and hopefully make it more cache-friendly.

Integer Division

Since we cannot use a division operation in ARM, I manually written a long division algorithm in assembly that is used whenever there is an IR3 division operation. It is similar to string concatenation, in the sense that the registers clobbered by the long division algorithm are made known to the register allocator.

This is the ARM assembly used for long division:

```
mov __A2,right                %dividend
mov __A1,left                 %divisor
mov A4,__A2,ASR #31           % -1 if negative, 0 otherwise
add A3,__A2,__A2,ASR #31      % magic to take absolute value
eors A2,A3,__A2,ASR #31       % magic to take absolute value
moveq output_reg,#0           % if divisor is zero, set output to 0, and skip to end
beq L3
eor A4,A4,__A1,ASR #31        % -1 if sign different, 0 if same
add A3,__A1,__A1,ASR #31      % magic to take absolute value
eor A1,A3,__A1,ASR #31        % magic to take absolute value
% now A1 contains nonnegative dividend, A2 contains nonnegative divisor
mov A3,#1
L1:
cmp A2,A1,LSR #1              % shift A2 and A3 to the biggest digit
movls A2,A2,LSL #1
movls A3,A3,LSL #1
bls L1
mov output_reg,A4             % set output to -1 if we need to flip the sign later, otherw
L2:
cmp A1,A2                     % actual long division happens here
subcs A1,A1,A2
addcs output_reg,output_reg,A3
movs A3,A3,LSR #1
movne A2,A2,LSR #1
```

```
bne L2
eor output_reg,output_reg,A4   % set the sign of the answer
L3:
```

The code has been carefully written to ensure that division by zero or division where the dividend or divisor is $2^{31} - 1$ or -2^{31} does not lead to an infinite loop.

Println

Println is implemented via `printf` for strings and integers, and via `puts` for booleans.
Since our strings are not C-style strings, we use printf with the length specifier, i.e. `printf(%.*s\n, length, ptr)`.

Booleans are printed as `true` or `false`, as they are more natural than `0` or `1`, and it helps to distinguish them from integers.

Readln (extension)

My compiler also implements the readln operation for strings, integers, and booleans. In all cases, it uses the POSIX `getline` function to read an entire line, and then handles IO errors appropriately as if it is the empty string.

After that, the processing depends on the type:

- For strings: we convert the C-style string into our string representation
- For integers: we use the `strtol` function to convert the C-style string to an integer
- For booleans: we check the first character, and if it is `t` or `1` then it returns `true`, otherwise it returns `false`

By using `getline`, which allocates a buffer automatically, we avoid the need to manually grow the buffer using functions like `realloc`.

Class layout (optimisation)

`Int`, `String`, and objects (pointers) take up 4 bytes when stored in a class, but `Bool` only takes up one byte. My compiler respects alignment of the non-`Bool` types, but may reorder `Bool` fields to pack the more efficiently.

To minimise the size of instances of a class, we reorder the `Bool`s to store them in adjacent memory locations, to minimise the amount of padding in the class.

Objects are allocated with `calloc` instead of `malloc`, so that all their fields are zeroed out.

If-Goto (optimisation)

If-Goto instructions are optimised to emit the proper `cmp ...` followed by `b<c> ...` sequence.

In non-optimised code, the IR3 instruction `if (a > b) goto L1;` is compiled into something roughly equivalent to `t1 = a > b; if (t1) goto L1;`:

```
cmp a,b
movgt r0,#1
movle r0,#0
cmp r0,#0
bne .L1
```

However, in with optimisation enabled, the entire IR3 instruction is converted as a whole in order to use just a single `cmp` instruction:

```
cmp a,b
bgt .L1
```

Where the second argument of the RelExp is a constant, the constant will be an immediate, e.g.:

```
cmp a,#123
bgt .L1
```

This generates more efficient ARM assembly code.

Modification of IR3

The optimisation described above works for some other operators that are not RelExps. In particular, they also work for `&&` (conjunction), `||` (disjunction), `!` (negation), and `-` (unary minus) too. I modified the IR3 specification to allow these binary and unary operators in an If-Goto IR3 instruction, so that we can emit better ARM assembly code.