CS4212 2019 Project Assignment 3 Details

The design of the parser and the type checker are in `Assignment 1.txt` and `Assignment 2.pdf` for completeness sake.

Compilation

Make sure `javac` is available.

1. Inside the source folder, execute `make` .
2. To compile without optimisations, do `make compile file=/path/to/file.j` .
3. The compiled assembly will be in `/path/to/file.j.s` .
4. To compile with optimisations, do `make compileOpt file=/path/to/file.j` instead.

Testing

5 sample programs are located inside `tests/pa3` , along with their compiled outputs. Each test file has comments beside each relevant line to describe what I'm trying to test, so check that the output matches the comment.

I have tried to covered all the features of jLite: all operations like arithmetic and boolean are tested in `basics.j` , overloaded functions are tested in `overload.j` , method calls are tested liberally in all files, but especially in `basics.j` with more than 4 arguments, and in `recursion.j` to test recursive functions like fibonacci and the ackermann functions.

Field access is tested inside `objects_in_objects.j` , where I also test that objects are also allowed as fields in other objets.

Infinite loops are also possible in `infinite_loop.j` , where the string "loop" will get printed until gem5 or the user decides to terminate.

Register Allocation

Not really proud of this. I did it the easy way: assign all variables a place on the stack, then retrieve and reassign them as necessary.

For example, the jLite:

```
a = b + c + d;
```

would translate into IR3 something like:

```
t1 = b + c;
a = t1 + d;
```

and then would get compiled to ARM like:

```
# t1 = b + c:
ldr v2, [address of b]
ldr v3, [address of c]
add v1, v2, v3
str v1, [address of t1]
# a = t1 + d:
ldr v2, [address of t1]
ldr v3, [address of d]
add v1, v2, v3
str v1, [address of ta]
```

Basically, load the two operands into `v2` and `v3`, operate on them putting the result into `v1`, then store it back into the stack.

Printing

We first have some default formatters for strings and integers.
Then, for strings, whenever a string literal is encountered we simply put an entry for it at the top of the ARM code like:

```
.data
.int_format:
        .asciz "%i\n"
.string_format:
        .asciz "%s\n"
.string1:
    .asciz "Hello, world!"
...
.text
```

Then the line

```
println("Hello world!");
```

would become:

```
ldr v1, =.string1
ldr a1, =.string_format
ldr a2, v1
bl printf(BLT)
```

Handling `null`

We simply just ignore it. If someone tries to access the field of a `null` object, undefined behaviour will occur.

Method calls

The first argument to a method is always a reference to the object, to allow for member access.

Then, following the ARM calling convention, the arguments are placed in registers `a1` to `a4`, with any remaining arguments stored on the stack.

Subsequently, the method can retrieve the needed arguments from the registers/stack as needed.

Objects

Objects are stored on the heap. They are initialised with `new Classname()` , and then the method `_Znwj` is invoked with the size of the class, and that amount of space is allocated. Each field then has a unique offset to that address, so to access that field we just need to do `[obj_address, #field_offset]` to get the desired memory location.

Booleans

Since there is no boolean type in ARM, we represent them with integers, `1` for `true` and `0` for false. Accordingly, we can use the instructions `and` and `orr` for `&&` and `||` respectively. To print booleans, we check to see if it's `1` or `0` , then print out `true` or `false` respectively.

Static calculation optimisations

Some statements, such as:

```
a = 2 * 3;
```

do not need to be evaluated in ARM since the result is always the same.
Thus, we compute these results beforehand, and transform them into

```
a = 6;
```

to save instructions.

Peephole optimisations

Some examples:

Store and reload

```
str r1, [add]
ldr r1, [add]
```

optimises to become

```
str r1, [add]
```

Operate then move

```
op v1, v2, v3
mov v2, v1
```

optimises to become

```
op v2, v2, v3
```

We can be sure that v1 is not needed later, as v1, v2 and v3 are only used as temporaries.