CS4212 Assignment 3

zhiayang

Contents

1 Overview

This compiler implementation conforms to the specified Jlite language. It parses a conforming Jlite source file, performs typechecking, generates intermediate representation (IR) in the form of IR3, and then outputs 32-bit ARM assembly which can be compiled by `gcc`.

The full Jlite specification has been implemented, including the following 'bonus' features:

1. string concatenation
2. `println` and `readln` for strings, integers, and booleans
3. integer division support (via long division)

Register allocation is done via liveness analysis and graph-colouring, including a 'pre-colouring' phase to encode a variable's preferred register. A number of optimisations have also been implemented; these are further described below.

The compiler has 3 main phases:

i. Lexing and parsing
ii. Typechecking and IR generation
iii. ARM assembly generation

Optimisation steps are interspersed where appropriate. The first two parts are just a more organised re-stating of the previous submissions, so feel free to skim/skip them.

There aren't too many 'design decisions' here; the obvious ones are:

i. representing IR3 functions with basic blocks
ii. using SSA form for temporaries

Both were done with the intention of allowing for optimisations to be implemented easily; they appear to have achieved their goals. The other implementation choices were just what were natural and/or forced.

This report turned out a lot longer than I expected, sorry about that; the old stuff from the previous submissions are at the very end, and the interesting new stuff up front.

2 Code Generation

Code generation entails turning IR3 into ARM assembly. For some semblance of sanity, the assembly is not represented in a purely textual form, but encoded into a simple datastructure. This allows the compiler to abstract certain things away (like ensuring the constant operand is always second), and match instructions for optimisation in a less hacky way.

One small step is also to fix the layout for classes. All fields are 4 bytes except for booleans, which are 1 byte. To preserve alignment, boolean fields are packed at the back of the class. The total size of the class is then aligned (upwards) to a multiple of 4 bytes.

2.1 IR3 Lowering

Before that though, the compiler *massages* the IR3 into a form that is more palatable to the code generator, through a *lowering* step. This step is always performed regardless of optimisations, though it is done *after* the IR3 optimisation pass. The following lowering steps are performed:

Multiplication by a Constant

For some reason, the inferior instruction encoding on ARM doesn't let '`mul`' encode a constant operand. For this reason, IR3 multiplies with a constant operand will have the constant pulled out:

```
_t1 = _t2 * 3;      // old

_c0 = 3;
_t1 = _t2 * _c0;    // new
```

A 'strength reduction' optimisation is possible here (eg. to replace the multiply with a series of adds, shifts, and subtracts — or even the basic one of replacing multiplies by a power-of-two into shifts), but they were not done because I didn't want to.

There is apparently an entire research field on the *Single Constant Multiplication* problem, but my brain was not creased enough to understand the papers.

Strings & Large Constants

Next, 'large' constants are also extracted into a temporary, and the assembler is used to load a value from memory. The ARM instruction encoding only allows for 8-bit constants and a 4-bit shift. However, I decided to just limit the values to $(-256, 256)$ (inclusive). Constants outside that range will be transformed:

```
_t1 = _t2 * 420;    // old

_c0 = 420;
_t1 = _t2 * _c0;    // new
```

The reason for pulling this out is to let the register allocator (which will be discussed below) do its job and give '`_c0`' its own register. The code generator will then emit this instruction:

```
ldr a1, =#420
```

(assuming the allocated register was '`a1`'). A similar thing is done for strings, but always, since strings can only be loaded in this form (they can't be an immediate).

This is represented not as a normal IR3 statement, but by special 'pseudo-statements', which you can find in `cgpseudo.py`.

Dot-op Assignments

Assignments and dot-op (ie. field) assignments can contain an IR3 expression as their right-side operand. For regular assignments this isn't a problem, but for field assignments, I chose to (again) let the register allocator do its job by pulling the right side out into a temporary variable so it can get its own register.

This is because, even though the right-side can (almost always) be encoded in one instruction, it cannot also have a memory operand, so we would need another register either way.

Function Calls

Surprisingly, function calls also have to be lowered. The problem is when there are too many arguments, exceeding the number of allocatable registers. The problem is that all those variables need to be available *on the same statement*, which is impossible if they all need to be in registers.

However, only the first *four* need to be in registers; the rest are passed on the stack. The trick, then, is to let the existing spilling mechanism handle 'swapping' variables in and out of registers; we simply need to split up the function call over multiple statements so the register allocator can work.

Another pseudo-op, '`StoreFunctionStackArg`', is introduced. The code generation for this statement is responsible for storing the variable to the correct stack slot for the upcoming function call. After lowering, a particularly bad call might look something like this:

```
spill a;
spill this;
spill c;
spill b;
stack_arg(4): d;
stack_arg(5): e;
stack_arg(6): f;
stack_arg(7): g;
stack_arg(8): h;
stack_arg(9): i;
restore j;
stack_arg(10): j;
restore k;
stack_arg(11): k;
restore b;
restore c;
restore this;
restore a;
_t0 = _J3Foo_1xiiiiiiiiiiiE(this, a, b, c, d, e, f, g, h, i, j, k, 12);
```

By splitting the work of the function call across several statements, the allocator is able to insert spills and loads between them to make allocation succeed.

Unfortunately, while this is an elegant solution in design, the implementation is a lot... uglier. This bug was discovered around 3 hours ago, as I write this report on *Saturday morning*. The problem arises because the function call itself is responsible for saving and restoring `a1-a4`. Some finesse and hackery was needed to allow the call to *backpatch* the offsets correctly after it is known.

If this was built-in from the beginning, a better implementation would probably have been found. That being said, this implementation has been 'battle-tested' up to 27 arguments, so I am reasonably confident in it.

Phi Nodes

Lastly, as mentioned above, the IR3 generator creates phi nodes for the merge block of branches; there is not really a machine-instruction for that, so it is lowered simply by rewriting the source assignments to set the merge value instead.

As an example, this would be the code before lowering, with the phi node:

```
L1:
    _t0 = true;
    goto merge;
L2:
    _t1 = false;
    goto merge;
merge:
    _t2 = phi(_t0, _t1);
```

and after lowering:

```
L1:
    _t2 = true;
    goto merge;
L2:
    _t2 = false;
    goto merge;
merge:
```

So this lowering step breaks the SSA form, but the optimisations that rely on SSA have already been performed, so it is not a problem.

2.2 Register Allocation

Next, the heart of the compiler is the register allocator. This is a liveness-analysing, graph-colouring allocator. The registers `a1 - a4, v1 - v5, fp` are made available for use. The frame pointer is not used, so it is used as a general-purpose register, which does not break the calling convention for external functions.

2.2.1 Liveness Analysis

Liveness analysis is performed using a 'backwards may' — predecessor/union — dataflow algorithm. It is an iterative queuing algorithm that enqueues predecessors if the `IN` of a statement changes.

Analysis is performed on a statement level instead of a basic-block level for finer granularity in register assignments. The generated code is better, and there's less bookkeeping to do anyway. A dummy statement is inserted at the beginning of each function that serves as the first definition for all locals and parameters.

To extract the live ranges, the `IN` of each statement is collected, and the mapping is 'inversed', so to speak — to get a map of variable names to statement numbers.

2.2.2 Graph Colouring

To build the interference graph, the live range for every pair of variables is intersected, and non-empty sets indicate an interference. The graph itself is represented with an adjacency list, though this is not so important.

Before colouring, the 'preferences' of variables are recorded; incoming parameters will prefer to be in `a1 - a4`, and outgoing arguments (ie. variables used in a call) will also prefer those spots. Since a variable might be used in a call more than once (and in different positions), each variable will have a 'ranking' of its preferred registers, ordered by the number of times it prefers it.

Next, the graph is coloured using the simplify-select-spill-retry algorithm, except that variables are not immediately spilled if they can't be simplified. They are removed anyway, and only when they *actually* can't be coloured, are they spilled.

Colouring is straightforward; if the variable has preferences, they are taken into account, though it is not guaranteed. This means that this is, strictly speaking, not a *pre-colouring* method, just a preferred colouring. Otherwise, any free register is used.

2.2.3 Register Spilling

To spill registers, we introduce two pseudo-statements, `SpillVariable` and `RestoreVariable`. As the names suggest, these are responsible for spilling and restoring variables. A spill is treated as a use, and a restore is treated as a define.

Some edge-casing is done so that, on the dummy entry, the 5th parameter onwards is not spilled (since it already lives on the stack), and locals are also not spilled (since they have indeterminate values to begin with). This prevents redundant stores to stack.

2.2.4 Scratch Registers

In prior iterations of the code generator/register allocator, 2 registers were reserved as 'scratch' registers, for the purposes of loading spilled variables from memory, and for dealing with strings and large integers. However, this was a pain to deal with, since it introduced a bunch of problems:

i. two operands of the same instruction can't use the same scratch register

ii. fewer registers were available for use

iii. if the destination *and* both operands were spilled, then 3 scratch registers were required

In the end, this was not a great solution; I then introduced the *lowering* step to fix this. As mentioned briefly above, constants are pulled out to 'let the register allocator do its job'. This essentially means letting that new temporary get its own formally-allocated register. Even if it's spilled, the generated spill code means that it *will* get a register for the duration of its use (which is very short).

This concept also extends to the spill and restore pseudo ops; a register is nicely allocated for them. I think this is the intended implementation for a graph-colouring register allocator, but I didn't think of it at first. Sadge.

Either way, this means that no scratch registers are required, and the code generator can be *sure* that, when a value is requested at a statement, it will always be available in a register.

2.3 Assembly Generation

There isn't really much to discuss here that is unique for this implementation. The generator simply traverses each IR3 statement in the function and generates the corresponding ARM assembly.

For each function, a 'god object' is created which holds all the state required for generating that function, namely the register assignments, stack spill locations, and list of instructions. Additionally, some global state is also used to maintain the class layouts. These are in `cgstate.py`.

One hiccup is that C compilers will insert an implicit `return 0` at the end of '`main`'; I did not want to rewrite all the return statements, so the user-defined '`main`' is renamed and wrapped in a real main function that simply sets '`a1`' to 0 on exit so a nice exit code is returned.

Argument number 5 and above are passed on the stack; the code generator also follows this convention for obvious reasons. A benefit of this is that there is a free space to spill these arguments if needed — just write to the caller stack frame.

The ARM calling convention (AAPCS) states that the callee is free to write to the caller-owned area for stack-passed variables, and we do just that instead of increasing our own frame size.

Some implementation details will be discussed below.

2.3.1 Instruction Representation

In the beginning, emitted assembly was simply stored as a list of lines; no backpatching was required (the assembler handles any of that), but it was error-prone and not very elegant. To make implementing optimisations easier (as opposed to doing some kind of textual string match...), ARM instructions are now encoded as a datastructure.

This allows the use of symbolic constants for registers and instructions:

```
fs.emit(cgarm.store_multiple(cgarm.SP.post_incr(), [ cgarm.V1, cgarm.PC ]))
```

Furthermore, an interesting feature is that instructions can be *annotated* with comments (which can be disabled with a flag). This allowed for an easier time debugging the generated assembly, since the corresponding IR3 statement can be annotated on an instruction.

This annotation feature is also used to show the register assignments and spills for each function at its head, again to ease debugging. They are just comments, so there is no performance penalty.

2.3.2 Println & Readln

These are implemented for all the supported types as given in the IR3 specification — integers, booleans, and strings. Printing them is straightforward; the latter two simply use '`puts`', while the former uses '`printf`'. Conditional moves (stores) are used to eliminate branching for printing booleans.

For reading, '`scanf`' is used for integers and booleans. Integers behave how one would expect, but I chose to let the following represent a 'true' value for booleans: any string starting with '`1`', '`t`', or '`T`' will be considered true (regardless of the remaining characters).

For all other inputs, they are considered false. For '`readln`' specifically, each variant (for each type) is implemented as a separate function. This way, no special care is needed to handle clobbers across its boundary, just normal function-calling code.

2.3.3 Integer Division

Like '`readln`', this is implemented as a separate routine. It is a very simple algorithm that simply does repeated subtraction and returns the count. It handles a zero divisor as well as negative operands correctly, though there might be other edge cases.

The implementation is quite simple:

```
__divide_int:
    @ takes two args: (dividend, divisor) and returns the quotient.
    stmfd sp!, {{v1, v2, v3, v4, v5, fp, lr}}
    cmp a2, #0              @ check if we're dividing by 0. if so, just quit.
    beq .exit
    movs v4, a1, asr #31    @ sign bit (1 if negative)
    rsbne a1, a1, #0        @ negate if the sign bit was set (ie. abs)
    movs v5, a2, asr #31    @ also sign bit
    rsbne a2, a2, #0        @ negate if the sign bit was set (ie. abs)
    mov v3, #0              @ store the quotient
.L1:
    subs a1, a1, a2         @ check if we're done
    blt .done
    add v3, v3, #1
    b .L1
```

```
.done:
    mov a1, v3
    eors v1, v4, v5         @ check if the sign bits are different
    rsbne a1, a1, #0        @ negate if so
.exit:
    ldmfd sp!, {{v1, v2, v3, v4, v5, fp, pc}}
```

2.3.4 Function Calls

Function calls needed a little special consideration, apart from the mess with passing parameters on the stack. As the calling convention dictates, `a1` - `a4` are clobbered on a call boundary, so they need to be saved, but clearly it would be suboptimal to always save and restore 4 registers on every call boundary.

The solution here was to invert the variable live ranges, and map it with the assignments; this gets us a 'register live range', or ranges (statements) where each *register* is live. Then, in deciding whether or not one of the 'a' registers should be saved, we check if it is live in any of the *successors* to the call statement.

This greatly reduces the amount of unnecessary stack traffic, which is good. This 'optimisation' is always performed, since it is technically part of the register allocation machinery.

However, I have yet to counter the situation where (for example, 'a1') is not spilled, and it is used before and after a series of function calls; this results in the following (simplified) code:

```
stmfd sp!, {a1}         @ caller-save
bl printf(PLT)
ldmfd sp!, {a1}         @ caller-restore
stmfd sp!, {a1}         @ caller-save
bl printf(PLT)
ldmfd sp!, {a1}         @ caller-restore
stmfd sp!, {a1}         @ caller-save
bl printf(PLT)
ldmfd sp!, {a1}         @ caller-restore
@ finally somewhere here, a1 is used
mov v1, a1
```

An assembly optimisation pass (discussed below) can reduce it a little to this:

```
stmfd sp!, {a1}
bl printf(PLT)
ldmfd sp, {a1}
bl printf(PLT)
ldmfd sp, {a1}
bl printf(PLT)
ldmfd sp!, {a1}
mov v1, a1
```

However, there are still a lot of unnecessary stack loads which cannot be eliminated by a simple analysis like this. I think mitigating this would require dataflow at the *register* level (instead of variable-level), which I am not keen to do.

2.3.5 Strings

Strings are implemented as a pointer + length, where the length is stored in the first 4 bytes on the heap, before the string data itself. The string data *is* null-terminated, so we can trivially pass it to C functions.

However, having the length allows string concatenation to be performed in a single pass, which is good for efficiency. As for strings in the program itself, they are interned, so a given string is only emitted once per program.

I briefly mention this in the typechecking section below, but I extended IR3 to allow string comparisons with '==' and '`!=`'. Comparison is performed with '`strcmp`', and null string objects are distinct from empty strings.

2.3.6 Memory Management

There is none. Classes are allocated with '`calloc`', which ensures that they are zero-initialised according to the Jlite specification. Memory is *not* freed, since it is impossible to determine when an object should be freed without implementing some sort of garbage collector.

Memory is never explicitly freed, but the OS will handle that when the program terminates anyway.

Somewhat related to memory, but not really — as mentioned above, the compiler makes use of the 'constant pool' to store large constants inline with the code, eg. `ldr a1, =#12345`. The issue is that the assembler appears to 'pool' (haha) these constants near the end of the generated executable (maybe in the data section?).

Apparently, the maximum offset is at most 4 kB; if functions get too large, or there are too many of them, the assembler complains. The mitigation is to insert `.ltorg` in between functions, so the assembler can choose to place constants there as well.

3 Optimisations

Quite a number of optimisations were implemented in this compiler. I am *reasonably* sure that all of them are correctness-preserving. Probably.

There are two main groups: IR3 optimisations and assembly optimisations. The optimisations described here are turned off by default unless the optimisation flags is used.

Note that all other operations described above (lowering, AST constant folding, register allocator etc.) are *always* performed, regardless of the optimisation setting.

I don't want to paste a whole bunch of assembly here, so the testcases can be used to see the effectiveness of these optimisations.

3.1 IR3 Optimisations

Most of the work was done here, since it is at a higher level than raw assembly. They are all implemented in `iropt.py`. At a high level, the optimiser runs each optimisation pass on a function, and, if any of them made a change, the loop is repeated.

```
while(changed)
{
    changed = false;
    changed |= pass1();
    changed |= pass2();
    // ...
}
```

Each group of optimisations will be described below.

3.1.1 Basic Block Pruning

This group contains 3 optimisations:

i. eliminating double jumps

ii. removing unreachable blocks

iii. removing unreachable statements

The first is simple — when generating branches, there can arise cases where the only statement in a block is an unconditional jump to another block; this pass rewrites its predecessors to jump directly to the target. Since the passes are run multiple times, this can eliminate an arbitrarily long chain of jumps.

The second builds on the first; after retargeting jumps, the 'middle' block will be unreachable; this pass removes those blocks (with care taken to update the predecessor list of its successors as well).

The third is for eliminating this case:

```
L1:
    // statements...
    goto L2;
    goto L3;
```

which can occur after conditional branches with constant predicates are reduced to unconditional branches. This pass simply removes the second branch. Again, this is capable of removing an arbitrary number of unreachable statements since it is run to convergence.

3.1.2 Temporary Cleanup

This next group contains two passes:

i. eliminating redundant temporary stores
ii. removing unused variables

The first pass is to deal with a quirk of the typechecker/IR3 generator; it will often generate code like this:

```
_t3 = _t2 + 69;
x = _t3;
```

If these two statements are in the same block (with any number of statements between, though this is usually 0), then it will be rewritten to assign to 'x' directly. This can be done because temporaries are in SSA form; we know that '`_t3`' will never be assigned again, so it is safe to perform this optimisation.

In the second pass, unused variables are eliminated. A variable is considered unused if it is only assigned to, but is never used by any statement. After the first pass above, '`_t3`' is left as-is, since it will be removed in this pass.

Some care is taken in case the assignment is a function call (ie. the return value is not used); in this situation, the call is transformed into a statement, and the variable is discarded.

3.1.3 Dataflow Optimisations

This next group consists of three passes:

i. (global) common subexpression elimination
ii. constant propagation
iii. copy propagation

A generic function was used to abstract the dataflow algorithm for all of these passes, which wasn't that hard since all 3 use the 'forward must' variant.

In the first pass, CSE is performed on the entire function, making it a global CSE. Since IR3 statements are at most binary, the common subexpression is at most binary. In isolation then, it might not make sense to implement CSE, since the cost of 1 arithmetic instruction is not high. However, copy propagation allows the simplification to keep going.

The second and third passes perform constant and copy propagation respectively. They are very similar, but perform fundamentally different optimisations. I won't go into too much detail since they are covered in class.

3.1.4 Constant Evaluation

The last pass is constant evaluation, which, as the name suggests, evaluates constants. After (possibly multiple rounds) of the previous passes, including GCSE and constant propagation, there maybe be IR3 expressions all-constant operands. This pass simply evalutes them at compile time.

An additional note is that conditional branches are replaced with unconditional ones (or removed) if their condition is constant; combined with the block-based optimisations, dead-code elimination is essentially done, implicitly.

3.2 Assembly Optimisations

Compared to the IR3 optimisations the assembly ones are relatively trivial, and are just meant to counter some of the quirks of the code generator. All of them work on a 'sliding window' basis, and just look at a subset of instructions at a time.

One case to note is that labels are present in the instruction stream; this acts as a natural 'barrier' to prevent cross-block optimisations.

3.2.1 Redundant Loads & Stores

These passes handle the following cases:

i. When there are two consecutive loads or stores from the same address to the same register, one of them can be eliminated

ii. When there is a load followed by a store to/from the same address to/from the same register, the store can be eliminated

These mostly arise from preserving caller-saved registers across a function call, and from spilling or restoring variables. Note that in both cases, if any operand has a side effect (ie. a post-increment '`!`' — we don't generate the other form), it cannot be eliminated.

3.2.2 Redundant Arithmetic

To simplify the code generation, the compiler doesn't check whether the source and destination of a `mov` are identical; this optimisation eliminates cases like `mov a1, a1`, as well as `add a1, a1, #0`, among others.

3.2.3 Branch Optimisation

The last pass optimises conditional branches. In unoptimised form, a conditional branch takes 5 instructions, due to the way the IR3 was lowered:

```
cmp v2, v1                  @ _t3 = k == _c20;
moveq v1, #1
movne v1, #0
cmp v1, #0                  @ if (_t3) goto .L1;
bne ._J3Foo_3fooiiiiiE_L1
```

By applying some pattern-matching on the sliding-window view, we are able to optimise this to eliminate the conditional moves and branch directly on the flags register:

```
cmp v2, v1
beq ._J3Foo_3foooiiiiiE_L1
```

Lastly, for the following case:

```
    b .L1
.L1:
    mov a1, a2, #69;
```

The unconditional branch is redundant, and can be eliminated.

4 Project Organisation

This is a brief overview of how the project is laid out.

4.1 Source Tree

First, the sources themselves:

```
test.py                 # test runner
compile.py              # main driver
src/                    # main source folder
    util/               # utility functions (errors, StringView, etc)
    ast.py              # Jlite AST
    cgannotate.py       # annotations
    cgarm.py            # abstraction for ARM assembly
    cgliveness.py       # liveness analysis (dataflow)
    cglower.py          # IR3 lowering passes
    cgopt.py            # assembly optimisations
    cgpseudo.py         # new pseudo-ops for IR3 (phi, spill/restore, etc.)
    cgreg.py            # register allocator
    cgstate.py          # codegen state classes
    codegen.py          # actual codegen functions
    ir3.py              # IR3 ast
    iropt.py            # IR3 optimisations (all the good stuff)
    lexer.py            # Jlite lexer
    parser.py           # Jlite parser
    simp.py             # AST constant folding
    typecheck.py        # typechecker and IR3 generator
```

4.2 Tests

To aid testing, a test runner script was written; it calls out to `compile.py`, GCC, as well as the gem5[1] simulator, to run the tests.

It is capable of capturing stdout (as well as feeding stdin, to test `readln`) and checking it against the expected output, both for optimised and non-optimised runs.

A brief description of the tests:

1. **`simple`**
   tests various basic functionalities; nothing too interesting
2. **`calls`**
   tests function calls up to 27 arguments
3. **`cse`**
   tests common subexpression elimination, as well as constant and copy propagation
4. **`short_circuiting`**
   tests short-circuiting behaviour
5. **`recursion`**
   tests recursive calls
6. **`fibonacci`**
   iterative implementation of fibonacci up to (almost) 2^{31}
7. **`support`**
   tests the division and string comparison routines
8. **`readln`**
   tests `readln`

[1] `https://www.gem5.org`

5 Prior Work

These parts discuss (in a little more detail) the previous work done for assignments 1 and 2. There aren't many changes (but there are some), so it isn't critical.

5.1 Lexing & Parsing

The discussion here will be somewhat brief.

5.1.1 Lexing

Lexing is straightforward, using a greedy, string-matching approach. To avoid copies (which I now realise is pointless given the costs of the subsequent steps...), a '`StringView`' is used, which tries to prevent copying strings and instead uses indices to perform operations.

During lexing, location information is generated for each token, which is then carried over to AST nodes and IR3 nodes; this allows accurate error reporting in all phases of the compilation.

5.1.2 Parsing

The parser uses a recursive-descent, Pratt/precedence climbing parsing algorithm. The Jlite grammar is not unambiguous without lookahead, in particular for dotops and method definitions vs. field declarations.

This is solved by the following mechanisms:

Dot-ops

To correctly parse dotops, the parser uses a token of lookahead to detect a period ('`.`') after an identifier or method call, and recursively parses the 'atom chain' (my terminology). Note that there was a bug in the submission for Assignments 1 and 2 which has been fixed in this submission (`1 + new A().m()` would parse incorrectly as `(1 + new A()).m()`).

Method vs Field

To disambiguate (for example) '`Int a;`' from '`Int a(...)`', the parser simply uses a token of lookahead and defers constructing the AST node.

Function Calls

Function (method... same difference) calls can be both statements and expressions, but in Jlite expressions are not statements. The AST represents a call as an expression, with a separate '`ExprStmt`' type that wraps an expression into a statement.

5.2 Typechecking

As the name implies, this phase performs typechecking on the parsed program AST. IR3 generation actually happens concurrently (ie. IR3 is emitted immediately after a statement is typechecked), but it will be discussed separately for organisational reasons.

Typechecking is quite straightforward as well, give that Jlite is a very simply-typed language, without subtyping, (parametric) polymorphism, or any other funny ideas. The given typechecking rules are followed as specified.

Since there is no (apparent) ordering for each component, the typechecker first visits all classes, and for each class, visits all its methods. This ensures that any method can call any other method independent of declaration order.

Each method is then typechecked, which involves typechecking each statement, which involves typechecking their constituent expressions... and so on.

5.2.1 Constant Folding

Before typechecking begins, the AST is traversed to simplify any trivially-simplifiable constant operations, namely unary expressions with a constant operand, and binary expressions with both operands being constants.

This is distinct from the constant propagation/folding step in the optimisation pass. It is always performed (cannot be disabled), and is very weak, since it does not perform propagation across a statement boundary.

5.2.2 Method Overloading

With the promise of bonus marks, method overloading (ad-hoc polymorphism) was implemented. Again, with the simplicity of Jlite, this was quite simple to implement. A method with a given name can have multiple definitions, as long as their signatures are different. Methods can be overloaded on arity, type, or both.

At each callsite, matching methods are searched by comparing the 'virtual signature' of the argument types, and the declared signature of the parameter types of each overload. Note that only parameters are considered, and not the return type.

If there is not exactly 1 'solution' (ie. matching function), an error is reported. For ambiguous calls, the locations of each matching candidate are helpfully printed as well:

```
overload_5.j:8:19: error: ambiguous call to function: 2 overloads match
   |
 8 |     new Foo().kekw(null);
   |               ^
overload_5.j:14:5: note: here
    |
```

```
 14 |     String kekw(String a)
    |     ^
overload_5.j:20:5: note: here
    |
 20 |     String kekw(Foo a)
    |     ^
```

There is a bugfix here — in part 2, it would always return the first match, even if multiple functions match (ie. an ambiguous call). This has been fixed.

Ambiguity arises from '`null`' (as demonstrated above), since it can match any object type. This is the only case of ambiguity, which is quite easy to handle.

5.2.3 Handling Null

Nulls (ie. the literal '`null`') are handled by giving them a distinct type, '`$NullObject`', which cannot be named by the user. This type is 'compatible' with any object type, and so you can assign null to strings and objects, as intended.

5.2.4 Non-void Returns

To avoid undefined behaviour (the Jlite specification does not define this, so it is quite literally undefined), the typechecker also ensures that non-void functions return a value along all control flow paths.

This is done simply via a recursive traversal of the AST; function bodies must either contain a return statement at the top level, or else contain an if statement where both branches (recurisvely) return in all control paths.

5.2.5 Shadowing

The behaviour (ie. whether shadowing is allowed) is also not specified; it is implemented here. Local variables shadow class fields and method parameters, and parameters shadow class fields. Since there are only these 3 fixed scopes (variables must be declared at the start like it's C in 1989), implementation was not difficult.

5.2.6 Miscellaneous

It was rather strange that strings could be concatenated but not compared, and booleans could also not be compared. The compiler generates a superset of IR3 that allows these to exist; by extension, they need to be typechecked correctly.

These behave as one would expect; currently only '`==`' and '`!=`' are extended to strings and booleans.

5.3 IR Generation

As mentioned above, IR3 generation is actually intertwined with typechecking; each statement's checking function returns a list of IR3 statements, as well as a list of additional temporary variables to create.

The IR3 program is also organised as a rough kind of AST, except it is a very shallow tree. There is no attempt to generate optimised code here, but rather care was taken to *limit* the kinds of IR3 generated.

Notably, RelConds are not placed inside the condition of a conditional branch; instead, conditional branches are only predicated on variables; this means that the following is always generated:

```
_t1 = x > 69;
if(_t1) goto foo;
```

This was done mostly for convenience; an if-condition from the AST can contain an arbitrarily complex expression, so it is easier to always generate a temporary for it.

5.3.1 SSA Form

One important point is that the compiler actually emits static-single-assignment (SSA) form IR3 for temporaries. That is, a temporary variable is only assigned once (this does not hold for named variables).

This was not explicitly done at first, but rather a product of how the IR3 was generated. Later, after it was decided to actually make it SSA, phi nodes had to be added due to the logical AND/OR operators.

Phi nodes act as an assignment to a (temporary) variable, with a list of variables from predecessor blocks that it should take values from. This ensures that the variable normally used to hold the result of the expression is only assigned once, and the final result is a phi over all the possibilities.

The IR3 generated by this compiler will include phi nodes for '&&' and '||' expressions, so its output is not strictly conforming IR3.

5.3.2 Short Circuit Evaluation

This behaviour was also not specified; since all prior material seems to assume that short-circuiting is present, it was implemented. As noted above, phi nodes are required to generate these correctly in SSA form.

No trickery with backpatching was required; with the power of modern technology, we can hold statements and labels in memory, inserting them at arbitrary points to generate correct code.

Short-circuiting is not always performed; if the right side of the expression has no side effects (ie. is not a function call), then no short-circuiting is performed, and both operands are always evaluated.

5.3.3 Name Mangling

Since methods can be overloaded, they must be name-mangled. The mangling form is somewhat inspired by the Itanium C++ ABI, but there really isn't anything concrete here. IR3 only knows about mangled names.

5.3.4 Basic Block Conversion

After all IR3 statements are generated each function is converted into basic block form. This is greatly helped by the IR3 generator itself — implicit fallthroughs are never generated, so blocks can easily be delineated by a label and a branch instruction.

Since IR3 does not have the classic two-target conditional branch, the generator always inserts an unconditional branch immediately after a conditional one, even if the target label is on the next line.

The compiler also generates labels in topological order (just by its nature), so a sorting step was not necessary. The predecessors of each block are also computed and stored for later optimisation steps.

An IR3 function is represented internally as a list of basic blocks, where each basic block is a list of statements.

5.3.5 IR3 Verification

Lastly, a small verification step is performed; this simply ensures that all basic blocks have either a return or an unconditional branch as their last statement (ie. they do not fallthrough), and that all temporaries are only assigned once (SSA-form).

This was mostly for my own sanity.