1 2005 Paper 5 Question 6

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2005p5q6.pdf

(a) Explain how a parse tree representing an expression can (*i*) be converted into stack-oriented intermediate code and then (*ii*) be translated into simple machine code for a register-oriented architecture (e.g. ARM or IA32) on an instruction-by-instruction basis. Also indicate how this code might be improved to remove push-pop pairs introduced by (*ii*). Your answer need only consider expression forms encountered in the expression:

```
h(a, g(b), c) * 3 + d
```

(i) A parse tree can be converted into stack-oriented intermediate code by performing a depth-first-search and appending to the linear code when different expressions are seen.

a search? looking for what?! maybe you mean an L->R post-order tree walk?

In OCaml-esque pseudocode:

```
type parsetree =
        | Add of parsetree * parsetree
        | Mul of parsetree * parsetree
        | Var of int ref
        | Call of ('a -> 'b) * parsetree
        | Args of parsetree * parsetree
        | Integer of int

let rec gen_code parsetree file parsetree =
  match parsetree with
  | Add(x, y) -> mk_code file x; mk_code file y; file.append(Add)
  | Mul(x, y) -> mk_code file x; mk_code file y; file.append(Mul)
  | Var(x) -> file.append(Push x); file.append(Deref x)
  | Call(f, args) -> mk_code file args; file.append(Apply(f))
  | Args(hd, args) -> mk_code file hd; mk_code file args
  | Integer(i) -> file.append(Push i)
```

Why does Deref find the arg ('x') on the stack whereas Apply is parameterised on ('f')?

(ii) The simplest way to convert a stack-oriented intermediate langauge into a register-oriented intermediate language is to simulate a stack. Programs would allocate a block of memory onto the heap and then use this as a stack.

This could be implemented by using one of the registers as a stack pointer. Every time data is pushed, increment the stack pointer and decrement when it is popped.

However, this doesn't exploit temporal locality and would be inefficient in a real implementation. A better way would be to use one register as the stack pointer, but to use the rest of the registers to simulate the top of the stack. The intermediate code would simply assume an infinite number of registers and when compiled, would move registers which aren't in use out onto the stack (and when functions are called). The optimised semantics would be as follows:

but why complicate it?
This is only the intermediate form so why make it harder to manipulate later by premature optimisation?

- `ldimm ri, n`

  Set the contents of register `ri` to `n`

- `call ri, f`

  Call the function f with register `ri` as argument and return value – note arguments with multiple arguments can be implemented by *ri* being a pointer to objects on the heap or by taking a closure.

- `ld ri, x`

  Dereference `x` and load its contents into register `ri`

You haven't done a peephole optimisation to remove push-pop pairs.

See live template:

- compilation is template translation (ie .replace)
- you don't need to keep the whole program in memory
- this can be done with just-in-time compilation

- stack caches keep a trivial amount more in memory
- but if you want to optimise more, you have to look across more of the program to get more long-range optimisations

Peephole optimisation:
- remove consecutive push/pop with Mov R1 <- #3
- you have a small window on the code, make instructions within this window better
  you can't make long-range optimisation.
- the rest of the code is stored somewhere else (maybe not even generated)
- you can do this in a JIT compiler
- this peephole optimisation can get pretty optimal for maths and within 10x of optimal for general purpose code

We can replace
  neg
  expr
  sub

where expr is any expression which has the net effect of adding a single thing onto the stack

- `add ri, rj, rk`

  Add the contents of `rj` and `rk` and store them in `ri`

- `set ri, x`

  Store the contents of register `ri` in the variable `x`

  where is 'x' in memory? seems that this doesn't solve a fundamental task of compilers!

- `neg ri, rj`

  Negate the value stored in register `rj` and store its value in the register `ri`.

- `sub ri, rj, rk`

  Subtract the value stored in register `rk` from the value stored in register `rj` and store the value in register `ri`.

(b) In Java, expressions are evaluated strictly left-t-right. Consider compiling the function `f` in the following Java definition.

```
class A{
        static int a, b;
        void f() { ... <<C>> ... }
        int g(int x) { ... a++; ... }
}
```

Indicate what *both* the intermediate code *and* (improved as above) target code might be for `<<C>>` for the cases where `<<C>>` is:

(i) `b = g(7) + a;`

In the stack-oriented intermediate language:

```
Push 7
Apply g
Push a
Deref a
Add
Assign b
```

In the register-oriented intermediate language:

```
ldimm r0, 7
call r0, g
ld r1, a
add r0, r0, r1
set r0, b
```

(ii) `b = a + g(7);`

In the stack-oriented intermediate language:

```
Push a
Deref a
Push 7
Apply g
Add
Assign b
```

In the register-oriented intermediate language:

```
ld r0, a
ldimm r1, 7
call r1, g
add r0, r0, r1
set r0, b
```

(iii) `b = (-g(7)) + a;`

In the stack-oriented intermediate language:

```
Push 7
Apply g
Neg
Push a
Deref a
Add
Assign b
```

In the register-oriented intermediate language:

```
ldimm r0, 7
call r0, g
neg r0, r0
ld r1, a
add r0, r0, r1
set r0, b
```

(iv) `b = a - g(7);`

In the stack-oriented intermediate language:

```
Push a
Deref a
Push 7
Apply g
Sub
Assign b
```

In the register-oriented intermediate language:

```
ld r0, a
ldimm r1, 7
call r1, g
sub r0, r0, r1
set r0, b
```

Comment on any inherent differences in efficiency at both the intermediate code and target code levels. You're committed to some inherant inefficiency due to the source language but can break free of it with some peephole optimisations.

404

2 2001 Paper 6 Question 6

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2001p6q6.pdf

(a) Describe one possible structure (e.g ELF) of an object file. Illustrate your answer by considering the form of object file which might result from the following C program.

```
int a = 1, b = -1;
extern int g(int);
extern int c;
int f() { return g(a-b) + c; }
```

It is not necessary to consider the exact instruction sequence, just issues concerning its interaction with the object file format.

ELF consists of segments and sections. Sections are used for linking and during compilation but are not included in the final executable. Segments are included in the final executable.

The ELF file contains: Machine code at this point

- `.text` Contains blanks where labels would be

  This segment contains machine instructions. However, in the object file these are not directly executable – we need to link to variables using `.data`.

  In the above program, this would contain the machine code instructions for `f` – however none of the variables would refer to real addresses – they'd be null with the `.rel.text` containing sufficient information to relocate.

- `.data`

  This segment contains the initial values for all (writeable global) variables or static variables.

  In the above program, `.data` would be 8 bytes large and contain 1, -1 – the values of `a` and `b`.

- `.rodata`

  This segment contains the initial values for all read-only global or read-only static variables.

  The above program has no read-only data of any sort so `.rodata` would be empty.

- `.bss`

  This contains uninitialised global/static variables. However, since they're uninitialised it only contains their sizes in bytes. The compiler will initialised this much memory to zero.[1] the *linker* will?

  The above program has no uninitialised global/static variables and so `.bss` would be empty.

- `.rel.text` Use of labels is stored here

  In RISC-V, each instruction either operates on registers, constants or uses exactly one address.

  This section contains the relocation data required for the `.text` segment to be converted into proper machine code. It is a list of pairs i, j where $(i, j) \in$ `.rel.text` means "the first free operand of instruction i points to the variable stored at offset j in the `.data` segment".

  In the above program, the `.rel.text` segment would contain two entries, relating the occurrences of a and b in f to their offsets in the data segment.

  What about c? There isn't an address in the `.data` segment for `.rel.text` to point to...Same dilemma for `.sym`.

  .rel.text is not /per variable/, it's per instruction that needs relocation.
  .symtab has entries for variables defined in other translation units.

- `.rel.data`

  Some entries on the `.data` segment are initialised to pointers – since memory is not allocated in the object file this is not known when the object file is created. The `.rel.data` segment contains pairs i, j where $(i, j) \in$ `.rel.data` means "the variable at offset i `.data` is a pointer to the variable referred to at index j in `.sym`".

  No variables in the above program contain pointers, so `.rel.data` would be empty.

- `.rel.rodata`

  This performs the same function as `.rel.data` except for `.rodata`.

  No read-only variables in the above program contain pointers, so `.rel.rodata` would be empty.

- `.sym`

[1] Whether or not these variables in `.bss` are initialised to zero is apparently controversial, Windows and Unix-like systems do it; and I was not able to find a convincing example of `.bss` *not* being initialised to zero – the unconvincing examples were from the 1960s.

.symtab (what are the things you can plug into holes)
  x y (the thing whose offset is x in the strtab is stored at offset y in .data)
  z # (the thing whose offset is z in the strtab is not locally defined)

.rela.text (which holes need which things)

and .text, including external references

This is a mapping from offset in `.data` of variables to offsets in the `.strtab` – a mapping from variable names to locations.

This would contain two pairs: $(a, \ell_a), (b, \ell_b)$ where ℓ_i is the offset of variable i in the `.data` segment.

- `.strtab`

  This is a list of the names of global variables in the `.data` segment – a `/0` separated string list.

  For the above program this would be "a/0b/0c/0".

don't you need c,f,g as well?

- `.line`

  This contains a mapping from source code line to machine code instructions. This is useful when throwing exceptions and for debugging.

- `.debug`

  This contains other debugging information.

Additionally, ELF files contain an import list (containing all the variables which the program needs to run) and an export list (containing all the variables the program allows other programs to import).

(b) Describe *how* a linker takes a sequence of such programs and produces an executable file.

meaning has no further unresolved linkage

Linkers concatenate the segments to each other and resolve offsets. This produces a second object file. If this object file is closed, it can be converted into an executable.

Resolving offsets is simple. If there are two object files o_1 and o_2, where the size of `.data` in o_1 is n bytes, we would resolve offsets by increasing every offset in the `.data` segment in o_2 by n. Similarly for the `.text`, `.rodata`, `.sym` and `.strtab` segments.

Note that care must be taken to rename local variables and functions to avoid name conflicts.

Linking the object files together forms a "closed" object file (which I define as an object file which imports no variables). Closed object files can be compiled into executable files.

- Firstly, allocate memory for the `.text` segment. This is stored in the lower memory addresses. Next, allocate the, `.rodata` `.bss` (initialise the specified number of bytes to zero) then `.data` segments on top of the `.text` segment.
- Next, we must use the relocation data stored in `.rel.text`, `.rel.data` and `.rel.rodata` to fill in values which were not known pre-compile time. This includes pointer addresses, variable addresses and more – for example we don't know at compile time whether 64-byte ints must be 4-byte aligned or 8-byte aligned.

  Use the sections as specified earlier in the description of ELF.

  Iterate through `.rel.rodata` and `.rel.data` and for each pair (i, j), initialise the value of the variable stored at offset i into the respective segments with the variable pointed to at index j in `.sym`.

  Iterate through `.rel.text` and for each pair (i, j), replace the first uninitialised variable in instruction i with a pointer to the variable stored at index j in `.sym`.

any nasty cases?

(c) Compare and contrast *static* and *dynamic* linking in a system using your object file format.

Static Linking:

- Code used from other libraries is statically linked at compile time.

- This means linking is not done at runtime.
- However, code size can be substantially larger since every file needs a copy of every library it uses. Although modern static linkers are smart and only import the functions the code could use.
- Updates to libraries will not affect existing code – which can both be good or bad. For example bug fixes will not affect existing code and can lead to code with known bugs being left in deployment. Conversely, backwards-incompatible changes to libraries will not break existing code.

Dynamic Linking:

- Code from libraries is dynamically linked at runtime. or load-time
- This means only one copy of each library is needed on each computer – code is shared.
- This adds overhead at runtime. or load-time
- This can use less RAM – if another program is also using the library then they can share and don't have to load in separate copies. needs OS support for dynamic linkage!

Many modern programs use a mixture of static and dynamic linking – it's more efficient to statically load small libraries and avoid page faults; while it's significantly more efficient to dynamically load larger libraries (ie pytorch is $\sim 512GB$ – statically linking this into every program which used it would rapidly use up disk space).

Dynamic linking can be done at load-time or run-time

.exe is an object file
execve has to pull pieces together. You need a big hash table with all functions and variables for all libraries installed on the system. You have to build this index.
It's good, you can share machine code. When you launch the application, it's slow to launch.
But it will work, if the OS does it, you can arrange for libraries to use COW -- so you only need one copy in RAM -- so libc will save a lot of memory.

BUT load-time dynamic linking isn't modular...

You'd like to dynamically link-load.
Now you're trying to extend machine code after the OS has setup your process.
This is hard. This allows you to load bits of the program at runtime.
This also creates DLL hell problems: where you've got different versions of things installed.