Principia Softwarica: The ARM Linker `51`

OCaml edition

version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Rob Pike and Yoann Padioleau

March 10, 2026

Contents

Chapter 1

Introduction

The goal of this book is to explain with full details the source code of a linker.

1.1 Motivations

Why a linker? Because I think you are a better programmer if you fully understand how things work under the hood, and the linker is the final piece of any development toolchain, the one generating finally the executable.

There are very few books about linkers, as opposed to a myriad of books on compilers or kernels. Linkers are usually not even studied during the computer science curriculum. This is a pity because they are central. Indeed, they make the link (no pun intended) between the compiler and the kernel: the linker generates from object files executables that the kernel will load. Without a linker there is no program to execute.

Here are a few questions I hope this book will answer:

- What is the format of object files?
- What is the format of executables? What is the format of machine instructions?
- What is the format of libraries?
- How are object files and libraries combined together and patched to form the final executable?
- What is the difference between a linker and a loader?
- What is the memory image of a program? How does it relate to the executable file? How does it relate to the original source code?
- What debugging information contains executables? How source-level debuggers can find which C source code corresponds to a specific binary instruction?

1.2 The Plan 9 ARM linker: `5l`

I will explain in this book the code of the Plan 9 ARM linker `5l`[1], which is about 8150 lines of code (LOC). `5l` is written entirely in C.

Like for most books in Principia Softwarica, I chose a Plan 9 program because those programs are simple, small, elegant, open source, and they form together a coherent set. The `'5'` comes from the Plan 9 convention to name architectures with a single character (`'0'` is MIPS, `'5'` is ARM, `'8'` is x86, etc), and the `'l'` means linker.

[1] See http://plan9.bell-labs.com/magic/man2html/1/8l, which despite its name covers also `5l`.

Like for the other Principia Softwarica books covering the development toolchain, I chose the ARM architecture [Sea01] variant, in this case of the Plan 9 linker `5l`, and not for instance the x86 variant `8l`, for reasons of simplicity. Indeed RISC machines are far simpler than CISC machines. Moreover, the availability under Plan 9 of an ARM emulator, `5i`, helps to understand the semantics of the machine instructions generated by `5l`.

The Plan 9 approach to linking is a bit different than in other operating systems. The Plan 9 linker is responsible for generating executables, but also for generating the machine code from the object files. Indeed, the object files generated by `5a` and `5c` are ARM-specific, but they do not contain machine code. Instead, under Plan 9 an object file is essentially the serialized form of the abstract syntax tree of an assembly source. The linker `5l` generates the actual machine code. I think though that it is actually a better design because it leads to less code in total and also to simpler code. The Plan 9 linker does also some dead code elimination which reduces the size of the binaries. This is partly to compensate for the lack of dynamic shared libraries in Plan 9.

1.3 Other linkers

Here are a few linkers that I considered for this book, but which I ultimately discarded:

- The original UNIX linker, called `ld` (for link edit), was creating executables for the DEC PDP11 architecture. `ld` started as an assembly program in UNIX V2[2] and finished as a C program in UNIX V7[3]. `ld` contains 1300 LOC, which is smaller than `5l`. This is partly because `5l` contains also the code to generate machine code (a job usually done in the assembler). However, `ld` targets an obsolete architecture (the PDP11).

- The GNU Linker (also called `ld`), part of the `binutils` package[4], is probably the most used open source linker. It is using the Binary File Descriptor (BFD) library to read and write object files using different formats. It supports many architectures (ARM, x86, etc) as well as many executable and object file formats (ELF, `a.out`, etc). It is fairly big though: 30 000 LOC for `ld/` and 500 000 LOC for `bfd/`. Even the ARM-specific file `bfd/elf32-arm.c` has already 17 000 LOC.

- Gold [Tay08] is a faster multi-threaded ELF linker originally developed at Google. It is now part of the `binutils` package. It supports also many architectures (ARM, x86, etc). It is unfortunately also fairly big: 153 000 LOC. Again, even the ARM-specific file `gold/arm.cc` has already 13 000 LOC.

- The LLVM Linker `lld`[5] aims to remove the current dependency to the host linker (e.g., the GNU linker under Linux) in the LLVM infrastructure. Its goal is also to be more modular and extensible than `ld`. It supports also many architectures and executable formats. It is smaller than `ld` and Gold, but still fairly large: 71 000 LOC.

- LD86[6] is an x86 16-bit and 32-bit linker, part of Bruce Evans' C compiler (BCC). It is an historical linker used to compile old versions of Minix and Linux. It is fairly small: 6100 LOC, which is smaller than `5l`. But, to be fair, `5l` does also machine code generation, which is done instead by AS86 for LD86. `5a`+`5l` is made of 12 000 LOC, which is smaller than AS86+LD86 at 18 600 LOC. Moreover, the instruction format in the ARM is far simpler to understand than the one in the x86, which makes it a better candidate for Principia Softwarica.

Figure 1.1 presents a timeline of major linkers. I think `5l` represents the best compromise for this book: it implements the essential features of a linker, for an architecture that is still relevant today (the ARM), while still having a small and understandable codebase (8150 LOC).

[2] http://minnie.tuhs.org/cgi-bin/utree.pl?file=V2/cmd/ld1.s
[3] http://minnie.tuhs.org/cgi-bin/utree.pl?file=V7/usr/src/cmd/ld.c
[4] https://www.gnu.org/software/binutils/
[5] https://lld.llvm.org/
[6] http://v3.sk/~lkundrak/dev86/

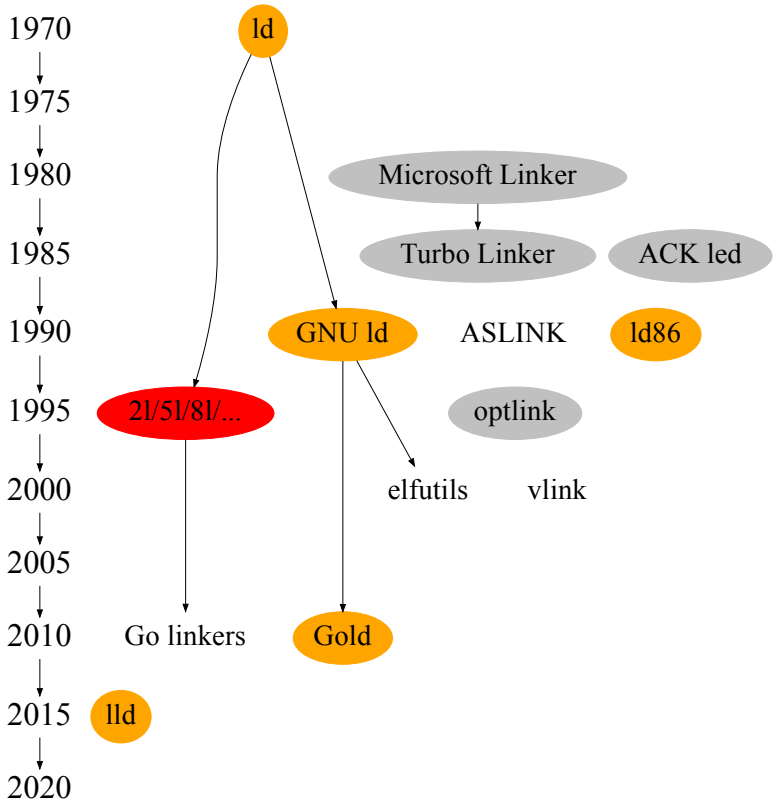

Figure 1.1: Linkers timeline

`5l` is obviously not as used as the GNU linker, but it is still a production-quality linker. It was used to link all Plan 9 programs at Bell Labs and it is still used in the toolchain of the Go programming language[7]. Because Go was originally conceived and used at Google, some of Google's services are linked by a Plan 9 linker.

1.4 Getting started

To play with `5l`, you will first need to install the Plan 9 fork used in Principia Softwarica (see http://www.principia-softwarica.org/wiki/Install). Once installed, you can test `5l` under Plan 9 with the following commands:

```
1   $ cd /tests/5l
2   $ 5a hello.s && 5a world.s
3   $ 5l hello.5 world.5 -o hello
4   $ ./hello
5   hello world
6   $
```

The command in Line 2 assembles the simple `hello.s` and `world.s` ARM assembly programs and generates the `hello.5` and `world.5` ARM object files. Line 3 then links the object files together and generates the final ARM binary executable `hello`. Line 4, which assumes you are under an ARM machine (e.g., a Raspberry Pi[8]), launchs the program.

[7] https://golang.org/cmd/link
[8] https://www.raspberrypi.org/

Note that it is easy under Plan 9 to cross compile from another architecture: you can use the same commands, `5a`, `5l`, etc. To play with `5l` under an x86 machine you just need after the linking step to use the ARM emulator `5i` instead:

```
...
4   $ 5i hello
...
```

See the EMULATOR book [Pad15b] for more information on `5i`.

1.5 Requirements

Because most of this book is made of C source code, you will need a good knowledge of the C programming language [KR88] to understand it. In fact, for Chapter 8, where you will see the code that generates ARM instructions, you will need a very good knowledge of C. Indeed, the code does lots of bit manipulations and uses many C idioms.

There are very few books explaining how a linker works. I can cite *Linkers and Loaders* [Lev99] and one chapter of *Computer Systems: A Programmer's Perspective* [BO10]. Thus, I assume most programmers do not really know how a linker works, which is why, in addition to explaining the code of `5l`, I will also explain most of the concepts related to linking in this book. This is a bit unusual in our Principia Softwarica series. Indeed, I usually assume the reader has read books introducing at least the concepts and theories underlying the programs I present.

Reading the ASSEMBLER book [Pad15a] is a requirement to understand this book. Indeed, many data structures introduced (and fully explained) in the ASSEMBLER book [Pad15a] are similar to data structures used by `5l`. This is normal because the assembler generates what the linker consumes. Those data structures will be presented only quickly in this book. The same is true for the object file format described at length in the ASSEMBLER book [Pad15a].

It is not necessary to know the ARM architecture to understand most of this book. For the machine code generation part though, in Chapter 8, I highly recommend to print the excellent colorful ARM reference card `http://re-eject.gbadev.org/files/armref.pdf`. It will help you to visually understand the binary format of the instructions. This card is very helpful to understand the code which does many bit manipulations to generate different parts of an ARM instruction.

1.6 About this document

This document is a *literate program* [Knu92]. It derives from a set of files processed by a tool, `syncweb` [Pad09], generating either this book or the actual source code of the program. The code and its documentation are thus strongly connected.

1.7 Copyright

1.8 Acknowledgments

I would like to acknowledge of course the author of `5l`, Rob Pike, who wrote in some sense most of this book.

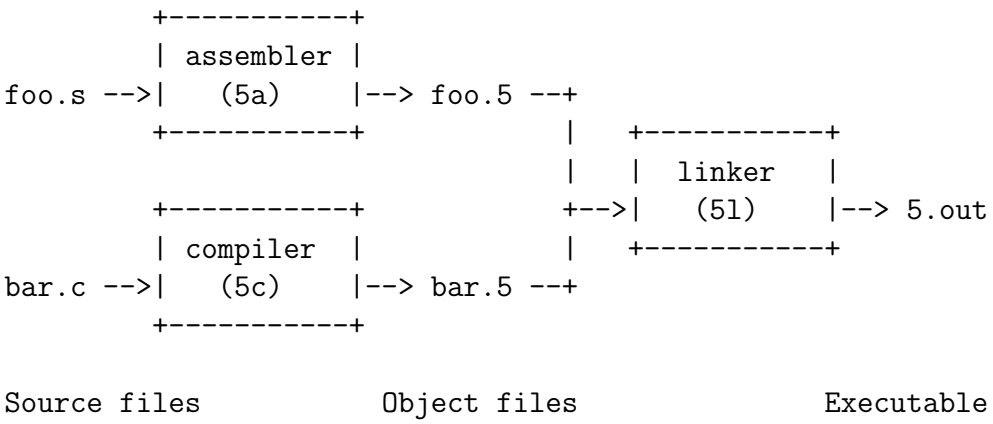

Figure 2.1: Linker inputs and output.

Chapter 2

Overview

Before showing the source code of `5l` in the following chapters, I first give an overview in this chapter of the general principles of a linker. I also describe quickly the format of the object files generated by `5a` and `5c` and used as inputs by `5l`, as well as the format of the executables generated by `5l`. I also define terms, explain how the code is organized, and more generally give the background necessary to understand the code I will show later.

2.1 Linker principles

A *linker* is a program which takes as input multiple object files and combine them to form an executable as shown in Figure 2.1.

An *object file* is really the simplest form of a *module*. It packs code, data, and information about exported and imported entities. Object files are generated by assemblers and compilers.

An *executable* is a file containing machine instructions as well as data. The code and data are stored in different parts of the file called *sections* as shown in Figure 2.2. The size and location of those sections are stored at the beginning of the file in a part called the *header*. There are different kinds of headers corresponding to different kinds of *executable formats*. In this book I will focus on the `a.out` format which is very simple. Section 11.5 will present other formats (including the popular ELF format used in Linux).

Thanks to the header, a part of the kernel called the *loader* can know where the code and data are stored in the executable file as well as its entry point. The loader, triggered by the `exec()` system call (see the KERNEL book [Pad14]), can then load (copy) in memory the different sections of the executable file and start executing its code.

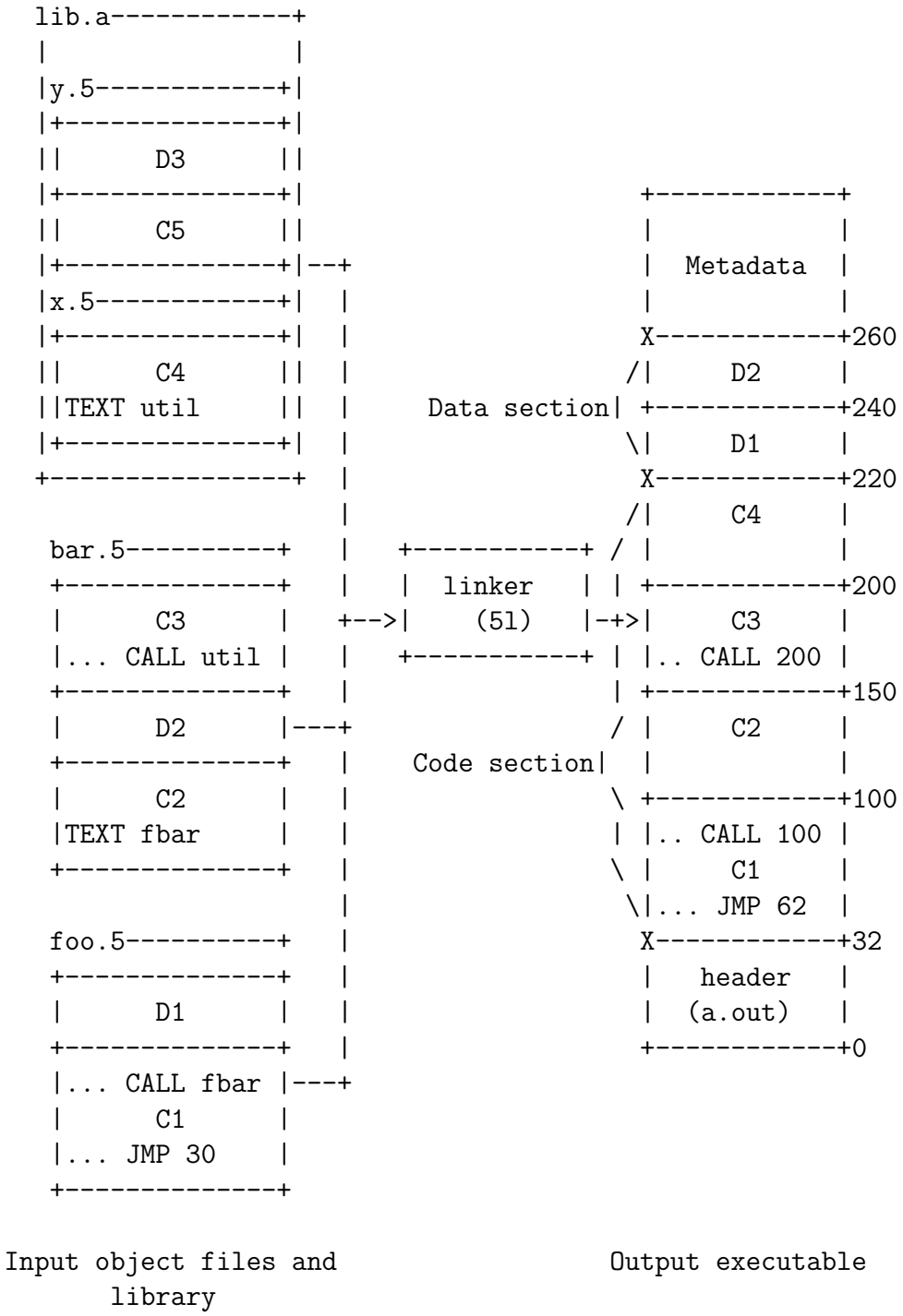

Figure 2.2: Linking process overview.

Figure 2.2 gives an overview of the linking process. The linker first *concatenates* the code parts of the different object files together to form the *code section* (the code parts `C1` to `C4`), also known as the *text section*. It does the same for the data to form the *data section* (`D1` and `D2`). Then, it links the *uses* of symbols in those code (or data) parts, e.g., the call to `fbar` in `C1`, to their *definitions* in possibly other code (or data) parts, here the definition of `fbar` in `C2`.

2.1.1 Separate compilation

One of the main operation of a linker is to concatenate parts of different files together. In fact, the program `cat` can be seen as a very primitive linker. Instead of using a linker, you could use `cat` to concatenate multiple source programs together and compile/assemble the resulting single (big) file[1]. As programs grow larger, this approach becomes inconvenient though. Linkers and object files enable *separate compilation* which in the long term saves lots of compilation time.

2.1.2 Symbol resolution

Linking the uses of symbols to their definitions is also called *resolving* the references to external symbols. The linker can do so because it has access to all the code (and data). Once all the code (and data) parts are concatenated together, the final addresses of the different entities can be known. So, in Figure 2.2 the call to `fbar` in `C1` can be transformed in a machine instruction to go to the address 100 where the code of `fbar` resides (assuming `fbar` was the first procedure in `bar.5` and so `C2`).

2.1.3 Relocation

Another important operation done by the linker is called *relocation*. Remember from the ASSEMBLER book [Pad15a] that `5a` can resolve the use of labels in jump instructions as both the definitions and uses of those labels must be in the same file. The instructions generated for those jumps assume though that the code of the program was loaded at the memory address 0. For instance, to jump to a label `foo` which happens to be defined just before the 30th instruction in a program `foo.s`, the instruction `JMP 30` will be generated in the object file as shown in Figure 2.2 (see `foo.5`). Once the code of object files are concatenated together and put after the header, the linker must *relocate* the jump instructions to take into account the new *memory address origin* where the object code containing the jump will be loaded. For our previous example, the linker generated the relocated instruction `JMP 62` in Figure 2.2 as `C1` is after the executable header which uses 32 bytes.

2.1.4 Disk image versus memory image

Linkers and loaders are strongly connected, just like assemblers and linkers. Indeed, one produces what the other consumes. In fact, early linkers were also called loaders as they were responsible for linking and loading in memory a program. It is important to note though, that the *disk image* of an executable generated by the linker does not necessarily match exactly the *memory image* of the program when loaded in memory by the loader. An offset in the file does not necessarily correspond to the same offset in memory, even though I assumed this was the case in Figure 2.2.

Under Plan 9, the header and code section are actually not loaded at the memory address 0 but instead after the first page at 4096 (4K). Indeed, the first page in the virtual address space of a program is reserved by the kernel and marked specially to generate faults when accessed. This helps for instance to track null pointer bugs. So, the instructions `CALL fbar` and `JMP 30` in Figure 2.2 would be actually transformed respectively in the resolved `CALL 4196` and relocated `JMP 4158` instructions with `5l` under Plan 9.

2.1.5 Libraries

Object files can be concatenated together to form *libraries* using a tool called `ar` (for archive). The linker can also take as input a set of libraries as shown in Figure 2.2 with `lib.a`. Those libraries are convenient for programmers because they just have to remember the name of one file, e.g., `libc.a`, instead of a possible long list of object files.

[1] Assuming the compiler/assembler could from one source file generate directly an executable.

Moreover, the linker can also take care of including only the object files in the library that matters, that is the object files containing code or data referenced by the other object files passed on the command line. For instance, in Figure 2.2 the linker decided to include in the executable the code of `x.5` (`C4`), which is part of `lib.a`, because it contains the definition of a procedure `util()` which is mentioned in the object file `bar.5`. It did not include the code from `y.5` though because such code would be anyway *dead code* in the executable. Such code would waste disk and memory space.

2.1.6 Static and dynamic linking

The linking I described until now is completely *static*. All the symbol references must be known at *link-time* in which case they can be fully resolved to generate an executable. This requires that all the object files (or libraries) containing those symbols get passed to the linker on the command line.

Another popular form of linking is called *dynamic linking*. Dynamic linking blurs the line between the responsibilities of the linker and loader. Indeed, with dynamic linking an executable can still contain unresolved references to external symbols; the loader at *load-time* must, before loading the program in memory, link additional objects.

Plan 9 opted mainly for static linking and so `5l` is mainly a static linker. Static linking is far simpler than dynamic linking and is arguably also better in many respects. I will delay the discussions on dynamic linking to Section 11.1.

For more information on the principles of linkers and loaders I recommend to read *Linkers and Loaders* [Lev99] which is entirely dedicated to the topic.

2.2 `5l` command-line interface

The command line interface of `5l` is pretty simple:

```
$ 5l
usage: 5l [-options] objects
$ 5l foo.5 bar.5 lib.a
$ ./5.out
```

Given the set of object files `foo.5` and `bar.5`, and a library `lib.a`, `5l` outputs an executable file called `5.out`. You can change this default behaviour by using the `-o <outfile>` option.

The default executable format is `a.out`, a classic UNIX format, but this can be changed with the `-H<num>` option (`H` for header) as explained in Section 4.1.2. You can also change the entry point of the program with `-E<funcname>` which by default is `_main` as explained in Section 4.1.3. Other options related to debugging will be described later in Appendix A.

Executables generated by C compilers in UNIX-like operating systems, e.g., `gcc` under Linux, are usually called by default `a.out`. However, under Plan 9, for the same reason ARM object files use the `.5` filename extension and not `.o`, ARM executables are called `5.out` not `a.out`. This is more convenient in an operating system supporting multiple architectures at the same time. That way, you can have in the same directory an ARM executable `5.out` and an x86 executable `8.out` without any name conflict.

Another important linker option, `-T<num>`, allows to change the memory address origin of the code section (`T` for text section). In Plan 9 the default value for this address is 4128. Indeed, the loader in the kernel loads executables after the first page (4096), and includes the header which is using 32 bytes, so the text section will start at the memory address 4128. The machine code generated for the jumps and calls must then assume the code will be loaded at 4128. You can also change the memory address origin of the data section as explained in Section 4.1.2. Those options are almost never used by programmers, but they are necessary for producing special executables such as kernels or boot loaders as you will see in the KERNEL book [Pad14]. Indeed, those

binaries will be loaded (by the bootstrapping process or firmware) at special memory addresses (e.g., 0x80000000 for the ARM Plan 9 kernel, or 0x7c00 for an x86 bootloader).

Another set of options are used to manage libraries, which are really files encapsulating a set of object files, and will be introduced later in Chapter 6.

2.3 `hello.5` and `world.5`

In this section, I adapt the `helloworld.s` example of the ASSEMBLER book [Pad15a] to illustrate the linking process with a concrete example. The goal is also to learn how to use the debugging options of `5l` as well as tools such as `nm`.

2.3.1 The source files

I split the original source file `helloworld.s` in two files `hello2.s` and `world.s`. I also use the C library functions `fprint()` and `exits()` instead of using directly the `PWRITE` and `EXITS` system calls:

⟨linkers/5l/tests/hello2.s 15a⟩≡
```
  TEXT main(SB), $8
          /* fprint(1,&hello) */
          MOVW $1, R0
          MOVW $hello(SB), R1
          MOVW R1, 8(R13)
          BL fprint(SB)
          /* exit(0) */
          MOVW $0, R0
          BL exits(SB)
```

⟨linkers/5l/tests/world.s 15b⟩≡
```
  GLOBL   hello(SB), $12
  DATA    hello+0(SB)/8, $"hello wo"
  DATA    hello+8(SB)/4, $"rld\n"
```

If you do not understand the code, or the calling conventions, read the ASSEMBLER book [Pad15a]. Symbol definitions and uses are now spread over different files: the `hello` global is defined in `world.s` but used in `hello2.s`. Moreover, the `fprint()` and `exits()` functions are defined in source files of the C library but used in `hello2.s`.

2.3.2 The linking command

To compile the previous program do:

```
$ cd /tests/5l
$ 5a hello2.s -o hello.5
$ 5a world.s
$ 5l hello.5 world.5 /arm/lib/libc.a -o hello
$ ./hello
hello world
```

The main difference with the commands shown in Section 1.4 is the use of the C library, compiled here for the ARM architecture: `/arm/lib/libc.a`. You will see in chapter 6 other ways to link libraries.

2.3.3 Inspecting objects with nm

You can use the tool nm to get the set of symbols defined or referenced in object files (this set is also called the name list):

```
$ nm hello.5
 U exits
 U fprint
 U hello
 T main

$ nm world.5
D hello
```

nm can also be used on libraries:

```
$ nm /arm/lib/libc.a | grep fprint
...
fprint.5: T fprint
...
$ nm /arm/lib/libc.a | grep exits
...
atexit.5:         T exits
...
```

U stands for undefined, T for text (a defined procedure), and D for data (a defined global). From the previous commands you can see that hello.5 has three undefined symbol references, including hello which is a global data defined in world.5. The job of the linker is then to link those definitions to their uses.

There are other kinds of symbols and so other single letter codes used by nm. Those codes as well as the source of nm will be described fully later in Appendix D.1.

2.3.4 Dumping objects with 5l -W

Another way, more complete, to inspect the content of object files is to use the debugging option -W of 5l. The effect of the option is to "dump" the instructions of the object files passed on the command line:

```
1   $ 5l -W hello.5 world.5 /arm/lib/libc.a
2   ...
3    ANAME main
4   (1) TEXT {main}00000+0(SB),$8
5   (4) MOVW $1,R0
6    ANAME hello
7   (5) MOVW ${hello}00000+0(SB),R1
8   (5) MOVW R1,8(R13)
9    ANAME fprint
10  (6) BL ,{fprint}00000+0(SB)
11  (9) MOVW $0,R0
12   ANAME exits
13  (9) BL ,{exits}00000+0(SB)
14  (10) END ,
15  ...
```

```
16   ANAME hello
17  (1) GLOBL {hello}00000+0(SB),$12
18  (2) DATA {hello}0000c+0(SB)/8,$"hell"
19  (3) DATA {hello}0000c+8(SB)/4,$"rld\n"
20  (4) END ,
21  ...
22      ANAME _main
23  (8) TEXT {_main}00000+0(SB),R1,$802
4      ANAME setR12
25  (10)MOVW ${setR12}00000+0(SB),R12
26      ANAME _tos
27  (11)MOVW R0,{_tos}00000+0(SB)
28  ...
29  (21)BL ,{main}00000+0(SB)
```

The first two fragments show the instructions of `hello.5`, from Line 3 to Line 14, and `world.5`, from Line 16 to Line 20. The output is very similar to the assembly sources of Section 2.3.1. This is normal since under Plan 9 an object file is essentially the serialized form of the abstract syntax tree of an assembly source. See the ASSEMBLER book [Pad15a] if you do not understand the opcodes or pseudo-opcodes such as `ANAME`.

Symbol references are enclosed in braces, e.g., `main` Line 4, followed by the *symbol value* in hexadecimal, e.g., `00000` Line 4. The symbol value corresponds normally to the resolved memory address of the symbol. But, almost all those values and addresses are null when using `-W` because the option dumps instructions while the object files are read into memory, at the beginning, and so when the linker has not yet resolved any of those symbols.

Once a global has been declared, e.g., `hello` Line 17, the value of its symbol is then (ab)used to store its size, here `0000c` (12) Line 18. Later you will see that the symbol value will contain the resolved address of the global.

The last fragment, starting at Line 22, shows the instructions of a procedure called `_main`. The linker `5l` is looking by default for such a procedure for the entry point of the executables it generates (unless the `-E` option is used), which is why it is included. The reason for a default called `_main`, and not `main`, even though the entry point of C programs is `main`, is because `_main()` is normally a procedure defined in the C library. This procedure, written in assembly, does some core initializations and then calls `main()`, as shown by the last dumped instruction Line 29. `5l` is usually called to link C programs, and the C compiler `5c` assumes a few core initializations has been done before the call to `main()`, hence the choice of `_main` as the default entry point.

For more information on `5l -W` see Appendix A.3.

2.3.5 Debugging machine code generation with `5l -a`

You can also display the generated machine code by using the debugging option `-a` of `5l`:

```
1   $ 5l -a hello.5 world.5 /arm/lib/libc.a
2
3   00001020:           (1) TEXT {main}01020+0(SB),$8
4   00001020: e52de00c (1) MOVW.W R14,-12(R13)
5   00001024: e3a00001 (4) MOVW $1,R0
6   00001028: e59f1884 (5) MOVW ${hello}00014+0(SB),R1
7   0000102c: e58d1008 (5) MOVW R1,8(R13)
8   00001030: eb0000bb (6) BL ,{fprint}01324(BRANCH)
9   00001034: e3a00000 (9) MOVW $0,R0
10  00001038: eb00009a (9) BL ,{exits}012a8(BRANCH)
```

```
11
12 0000103c:          (8) TEXT {_main}0103c+0(SB),R1,$80
13 0000103c: e52de054 (8) MOVW.W R14,-84(R13)
14 00001040: e59fc870 (10) MOVW ${setR12}00ffc+0(SB),R12
15 00001044: e50c0fd0 (11) MOVW R0,{_tos}0002c+0(SB)
16 ...
17 00001068: ebffffec (21) BL ,{main}01020(BRANCH)
18 ...
19
20 000012a8:          (915) TEXT {exits}012a8+0(SB),R1,$16
21 000012a8: e52de014 (915) MOVW.W R14,-20(R13)
22 ...
23
24 00001324:          (874) TEXT {fprint}01324+0(SB),R0,$20
25 00001324: e52de018 (874) MOVW.W R14,-24(R13)
26 ...
27
28 00006b50: e49df03c (890) MOVW.P 60(R13),R15
```

The first column contains the memory address in hexadecimal where the code will be loaded (e.g., `01020` Line 4). Then comes the 4 bytes in hexadecimal of the generated ARM instruction (e.g., `e52de00c`). Indeed ARM uses fixed-length instructions of 4 bytes. The third column contains the (global) line number in parenthesis of the instruction in the original source file (assembly or C)[2]. Finally, the last column displays the disassembled instruction. As opposed to the previous section, the symbol values are not null anymore and contain now the resolved memory address of the symbol, e.g., `01324` Line 8 for the `fprint` symbol. The output contains also the `TEXT` pseudo-instructions to better see the procedure boundaries, e.g., Line 3 and Line 12.

The first instruction Line 4 starts at `01020` which is equal to 4128. Indeed, as said previously in Section 2.2, the kernel loads executables after the first page (4096) and includes the `a.out` header which is using 32 bytes. Thus, the code starts at the memory address 4128.

Note that even though it looks like there are two instructions at `01020`, with Line 3 and Line 4, the first instruction (`TEXT ...`) is a pseudo-instruction which got transformed by the linker in the ARM instruction Line 4 (`MOVW.M ...`). Indeed, `TEXT` is an assembly directive introducing a symbol, here `main`, and symbols are resolved in concrete addresses by the linker. The generated machine code contains only concrete addresses, no symbols (except for debugging purposes as explained in Chapter 9). So, the reference to `main` Line 17 is resolved to the concrete address `01020`. In the same way, the calls to `fprint` and `exits` Line 8 and 10 are fully resolved in respectively the memory addresses `01324` (Line 25) and `012a8` (Line 21).

The symbol value of `hello` Line 6 is `00014` which seems incorrect. Indeed, data should be stored after the code section and so the resolved address of `hello` should be beyond `06b50` (the address of the last instruction Line 28). But, the symbol value for data symbols contains the resolved address of the symbol as *an offset to the start of the data section*. The rational for this decision will be explained later in Chapter 7. Thus, the final address of `hello` is `014 + INITDAT` where `INITDAT` is the address where starts the data section.

For more information on `5l -a` see Appendix A.4.

2.3.6 Inspecting executables with `nm`

`nm` can also be used on executables. In that case `nm` not only displays the list of symbols, it also displays their addresses (in hexadecimal) when loaded in memory. Here it is used with the `-n` option to sort by addresses:

```
1 $ nm -n 5.out
```

[2] See Section 9.3.1 for more information about line numbers.

```
2     1020 T main
3     103c T _main
4     ...
5     110c T _div
6     1168 T _mod
7     ...
8     12a8 T exits
9     ...
10    1324 T fprint
11    ...
12    6b58 T etext
13    7000 d onexlock
14    7000 D bdata
15    7010 D argv0
16    7014 D hello
17    7020 d _exitstr
18    ...
19    7940 D edata
20    7b44 B onex
21    7c50 B end
22    7ffc D setR12
```

You find the same resolved memory addresses we saw in the previous section with `5l -a`, e.g., `12a8` for `exits` Line 8, or `1324` for `fprint` Line 10.

You can also now see the addresses of the data symbols. `hello` Line 16 is at the address `7014`. The data section starts at `7000` as indicated by the *special symbol* `bdata` (begin data) Line 14. So, `INITDAT` is `7000` which confirms that the address of `hello` is indeed `14 + INITDAT` (`7014`) as mentioned in the previous section.

The symbol after `hello` is `_exitstr` Line 17 at address `7020`. This confirms that `hello` is using 12 bytes (`0x7020 - 0x7014 = 0xc = 12`) as said in the source of `world.s` in Section 2.3.1.

The linker defines a set of special symbols: `etext` (end text) Line 12, `bdata` (begin data) Line 14, `edata` (end data) Line 19, and finally `end` Line 21. Those symbols are used only for their addresses and allow a form of *reflection* on the structure of the executable and its sections as explained in Section 7.6.

Another special symbol, `setR12` Line 22 plays a complex role in `5l`. This symbol is notably used by `_main` as indicated by the output of `5l -W` and `5l -a` you saw in the previous sections. Its role and its relationship with the pseudo-register `SB` will be explained later in Chapter 7.

2.4 Input object format

The object file is now the input, as opposed to the ASSEMBLER book [Pad15a] where it was the output. The format of ARM object files is actually fully described in the ASSEMBLER book [Pad15a] so I will not repeat those explanations here. Section **??** contains a figure summarizing this format though, which will be useful to understand the code of Chapter 5 which loads object files.

Section 2.3.4 contains the dump of a few object files and shows, as I said previously, that an object file is essentially the serialized form of the abstract syntax tree of an assembly source.

2.5 Output executable format: `a.out`

Plan 9 is using the very simple `a.out` executable format. So, `5l` by default generates executables in this format. This can be changed with the `-H` option. Other formats will be described in Section 11.5

The file `include/exec/a.out.h` contains a formal description of the `a.out` header:

⟨*struct* `Exec` 20⟩≡

```
// a.out header
struct Exec
{
    long magic;  /* magic number */

    long text;   /* size of text segment */
    long data;   /* size of initialized data */
    long bss;    /* size of uninitialized data */

    long syms;   /* size of symbol table */

    // virtual address in [UTZERO+sizeof(Exec)..UTZERO+sizeof(Exec)+text]
    long entry;   /* entry point */

    long _unused;
    // see a.out.h man page explaining how to compute the line of a PC
    long pcsz;  /* size of pc/line number table */
};
```

Figure 2.3 illustrates the format of `a.out` executables. An `a.out` header is made of 8 `long`s (32 bytes) used as follows:

- The first four bytes in `Exec.magic` contains a *magic number* identifying the file as an `a.out` executable as well as an ARM executable. This magic number is recognized by tools such as `file` and by the kernel loader which will load only files having this magic number *signature*.
- Then comes the size of the text and data sections. The format of the ARM machine instructions in the text section will be described in Chapter 8.
- `Exec.bss` contains the size of the BSS section which corresponds to uninitialized data. The actual values for those data is not defined in the executable, as opposed to the data section, but the loader must still reserve space in memory for those data. The following C code shows which section will be used depending on the declaration style of a variable:

  ```
  int global = 1; // Data section
  int another;    // BSS section
  ```

- `Exec.syms` and the executable symbol table will be discussed in Chapter 9.
- `Exec.entry` contains the memory address of the entry point of the program. This address is used by the kernel loader to start the program. It is usually the address of the `_main` procedure of the C library (unless the `-E` option is used). So, for the `hello` executable of the previous section, the value of the entry point is `0x103c`. The entry point can also be the address of the `_mainp` procedure if profiling is enabled as explained in Chapter 10.
- Finally `Exec.pcsz` and the line table will be discussed in Chapter 9.

Note that the `Exec` structure is actually not used by the code of `5l` for *writing* the header. Instead, the function `asmb()`, which you will see in Section 4.5, outputs directly the bytes of the header with a series of calls to the `lput()` (for output `long`) utility function. The `Exec` structure is used though in programs which are *reading* the header of executables. There are many programs which need to understand the format of executables, e.g., the debugger `db`, the profiler, and small utilities like `nm`. Under Plan 9, all those programs use a common library called `libmach` which defines `Exec` as well as many other data structures and functions. Appendix D discusses the `libmach` interface and the code of utilities such as `nm`, but the code of `libmach` itself will be shown in the DEBUGGER book [Pad16].

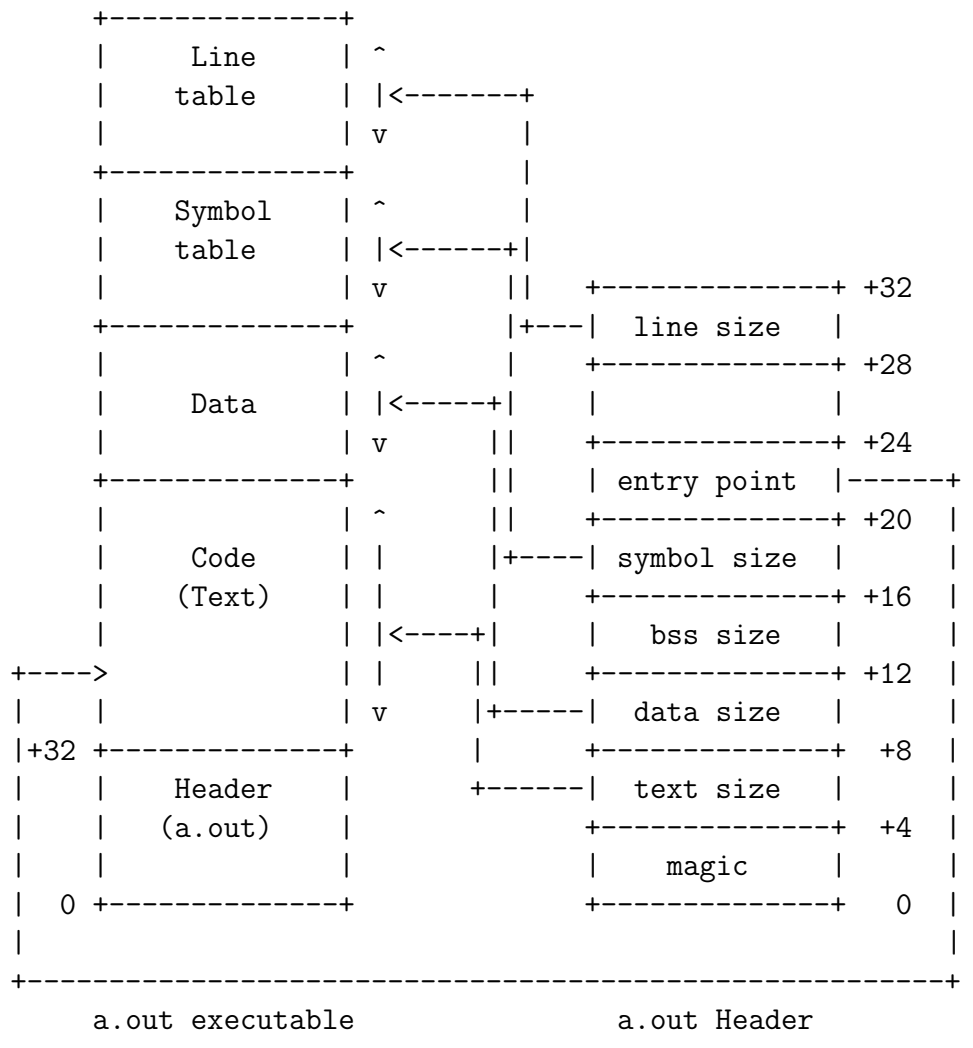

Figure 2.3: `a.out` executable format.

2.6 Code organization

2.7 Software architecture

2.8 Book structure

Chapter 3

Core Data Structures

3.1 Location and other basic types

⟨*type* Types.loc 23a⟩≡ (85e)
```
(* a single line number is not enough anymore, we need also the filename *)
type loc = Fpath.t * A.loc
```

⟨*type* Types.byte 23b⟩≡ (85e)
```
(* 8 bits *)
type byte = char
```

⟨*type* Types.word 23c⟩≡ (85e)
```
(* 32 bits *)
type word = int
```

⟨*type* Types.addr 23d⟩≡ (85e)
```
(* 32 bits *)
type addr = int
```

⟨*type* Types.offset 23e⟩≡ (85e)
```
(* 32 bits *)
type offset = int
```

3.2 Symbols and hash tables

⟨*type* Types.symbol 23f⟩≡ (85e)
```
type symbol = string * scope
```

⟨*type* Types.scope 23g⟩≡ (85e)
```
    and scope =
      | Public
      | Private of int (* represents a unique filename, less: use filename? *)
```

⟨*function* Types.s_of_symbol 23h⟩≡ (85e)
```
let s_of_symbol (s, scope) =
  (* less: could print the object filename instead *)
  s ^ (match scope with Public -> "" | Private _ -> "<>")
```

⟨*function* Types.symbol_of_global 23i⟩≡ (85e)
```
(* assert not Some -1 ! should have been set during loading! *)
let symbol_of_global (e : A.global) : symbol =
  e.name, (match e.priv with None -> Public | Some i -> Private i)
```

3.3 Sections

3.3.1 The virtual PC world

⟨*type* Types.virt_pc 24a⟩≡ (85e)
```
(* increments by 1. Used as index in some 'code array' or 'node array' *)
type virt_pc = A.virt_pc
```

⟨*type* Types.section 24b⟩≡ (85e)
```
(* before layout *)
type section =
  | SText of virt_pc
  | SData of int (* size *)
  | SXref (* undefined, alt: section option and use None *)
```

⟨*type* Types.value 24c⟩≡ (85e)
```
type value = {
  mutable section: section;
  sig_: signature option;
}
```

⟨*type* Types.signature 24d⟩≡ (85e)
```
(* the filename is for safe linking error report *)
type signature = int (* todo: * Common.filename *)
```

⟨*type* Types.symbol_table 24e⟩≡ (85e)
```
type symbol_table = (symbol, value) Hashtbl.t
```

3.3.2 The real PC world

⟨*type* Types.real_pc 24f⟩≡ (85e)
```
(* increments by 4 for ARM *)
type real_pc = int
```

⟨*type* Types.section2 24g⟩≡ (85e)
```
(* after layout *)
type section2 =
  | SText2 of real_pc
  (* offset to start of data section for ARM *)
  | SData2 of offset * data_kind
```

⟨*type* Types.data_kind 24h⟩≡ (85e)
```
  and data_kind = Data | Bss
```

⟨*type* Types.value2 24i⟩≡ (85e)
```
type value2 = section2
```

⟨*type* Types.symbol_table2 24j⟩≡ (85e)
```
type symbol_table2 = (symbol, value2) Hashtbl.t
```

3.4 Object files and programs

3.5 List of instructions

3.5.1 Code instructions

⟨*type* Types.code 24k⟩≡ (85e)
```
type 'instr code = ('instr code_bis * loc)
```

⟨*type* Types.code_bis 25a⟩≡ (85e)
```
(* a subset of Ast_asm5.line (no GLOBL/DATA, no LabelDef/LineDirective) *)
and 'instr code_bis =
  | TEXT of A.global * A.attributes * int
  | WORD of A.ximm
  | V of A.virtual_instr
  | I of 'instr
```

3.5.2 Data instructions

⟨*type* Types.data 25b⟩≡ (85e)
```
(* remember that GLOBL information is stored in symbol table  *)
type data =
  | DATA of A.global * A.offset * int * A.ximm
```

3.6 Code graph

⟨*type* Types.node 25c⟩≡ (85e)
```
type 'instr node = {
  (* can be altered during rewriting *)
  mutable instr: 'instr code_bis;
  mutable next: 'instr node option;
  (* for branching instructions and also for instructions using the pool *)
  mutable branch: 'instr node option;

  (* set after layout_text (set to -1 initially) *)
  mutable real_pc: real_pc;

  n_loc: loc;
}
```

⟨*type* Types.code_graph 25d⟩≡ (85e)
```
type 'instr code_graph = 'instr node (* the first node *)
```

⟨*type* Types5.instr 25e⟩≡ (87b)
```
type instr = Ast_asm5.instr_with_cond Types.code_bis
[@@deriving show]
```

⟨*type* Types5.node 25f⟩≡ (87b)
```
type node = Ast_asm5.instr_with_cond Types.node
[@@deriving show]
```

⟨*type* Types5.code_graph 25g⟩≡ (87b)
```
type code_graph = Ast_asm5.instr_with_cond Types.code_graph
[@@deriving show]
```

3.7 Executable header

⟨*signature* A_out.header_size 25h⟩≡ (87c)
```
val header_size: int
```

⟨*constant* A_out.header_size 25i⟩≡ (88a)
```
let header_size = 32
```

3.8 Linker configuration

⟨*type* Exec_file.linker_config 26a⟩≡ (90b)
```
type linker_config = {
  header_type: header_type;
  arch: Arch.t;
  header_size: int;

  init_text: addr;
  init_round: int;
  init_data: addr option;

  (* less: could be (string, addr) Common.either too *)
  entry_point: string;
}
```

⟨*type* Exec_file.header_type 26b⟩≡ (90b)
```
type header_type =
  | A_out (* Plan9 *)
  | Elf (* Linux *)
  (* TODO: | Elf64 | MachO | PE *)
```

⟨*type* Exec_file.addr 26c⟩≡ (90b)
```
type addr = int
```

⟨*type* Exec_file.sections_size 26d⟩≡ (90b)
```
type sections_size = {
  text_size: int;
  data_size: int;
  bss_size: int;
  (* TODO? symbol_size? more? *)
}
```

Chapter 4

Main Functions

4.1 main()

⟨*toplevel* Main._1 27a⟩≡ (83d)
```
let _ =
  Cap.main (fun (caps : Cap.all_caps) ->
     let argv = CapSys.argv caps in
     Exit.exit caps
        (Exit.catch (fun () ->
            CLI.main caps argv))
  )
```

⟨*type* CLI.caps 27b⟩≡ (74)
```
(* Need:
 * - open_in but should be only for argv derived files
 * - open_out for -o exec file or 5.out
 *)
type caps = < Cap.open_in; Cap.open_out >
```

⟨*signature* CLI.main 27c⟩≡ (74a)
```
(* entry point (can also raise Exit.ExitCode) *)
val main: <caps; ..> ->
  string array -> Exit.t
```

⟨*function* CLI.main 27d⟩≡ (74b)
```
let main (caps : <caps; ..>) (argv : string array) : Exit.t =

  let arch =
    match Filename.basename argv.(0) with
    | "o5l" -> Arch.Arm
    | "ovl" -> Arch.Mips
    | s -> failwith (spf "arch could not detected from argv0 %s" s)
  in

  let thechar = Arch.thechar arch in
  let thestring = Arch.thestring arch in
  let thebin = spf "%c.out" thechar in
  let usage =
    spf "usage: %s [-options] objects" argv.(0)
  in

  let infiles = ref [] in
  let outfile = ref (Fpath.v thebin) in

  (* LATER: detect type depending on current host *)
```

```
let header_type = ref "elf" in

let level = ref (Some Logs.Warning) in
(* for debugging *)
let backtrace = ref false in

let options = [
  "-o", Arg.String (fun s -> outfile := Fpath.v s),
  spf " <file> output file (default is %s)" !!(!outfile);

  "-H", Arg.Set_string header_type,
  spf " <str> executable (header) format (default is %s)" !header_type;
  (* less: Arg support 0x1000 integer syntax? *)
  "-T", Arg.Int (fun i -> init_text := Some i),
  " <addr> start of text section";
  "-R", Arg.Int (fun i -> init_round := Some i),
  " <int> page boundary";
  "-D", Arg.Int (fun i -> init_data := Some i),
  " <addr> start of data section";

  (* less: support integer value instead of string too? *)
  "-E", Arg.Set_string init_entry,
  spf " <str> entry point (default is %s)" !init_entry;

  (* pad: I added that *)
  "-v", Arg.Unit (fun () -> level := Some Logs.Info),
   " verbose mode";
  "-verbose", Arg.Unit (fun () -> level := Some Logs.Info),
  " verbose mode";
  "-debug", Arg.Unit (fun () -> level := Some Logs.Debug),
  " guess what";
  "-quiet", Arg.Unit (fun () -> level := None),
  " ";

  (* pad: I added that *)
  "-backtrace", Arg.Set backtrace,
  " dump the backtrace after an error";

  "-debug_layout", Arg.Set Flags.debug_layout,
  " debug layout code";
  "-debug_gen", Arg.Set Flags.debug_gen,
  " debug code generation";
] |> Arg.align
in
(try
  Arg.parse_argv argv options
    (fun f -> infiles := Fpath.v f::!infiles) usage;
with
| Arg.Bad msg -> UConsole.eprint msg; raise (Exit.ExitCode 2)
| Arg.Help msg -> UConsole.print msg; raise (Exit.ExitCode 0)
);
Logs_.setup !level ();
Logs.info (fun m -> m "linker ran from %s with arch %s"
      (Sys.getcwd()) thestring);

(match List.rev !infiles with
| [] ->
    Arg.usage options usage;
    Exit.Code 1
| xs ->
```

```
      let config : Exec_file.linker_config =
          config_of_header_type_and_flags arch !header_type
      in
      try
        (* the main call *)
        !outfile |> FS.with_open_out caps (fun chan ->
          link caps arch config xs chan
        );
        (* TODO: set exec bit on outfile *)
        Exit.OK
      with exn ->
       if !backtrace
       then raise exn
       else
         (match exn with
         | Failure s ->
             Logs.err (fun m -> m "%s" s);
             Exit.Code 1
         (* not sure this exn is currently thrown but just in case *)
         | Location_cpp.Error (s, loc) ->
             (* TODO: actually we should pass locs! *)
             let (file, line) = Location_cpp.final_loc_of_loc loc in
             Logs.err (fun m -> m "%s:%d %s" !!file line s);
             Exit.Code 1
         | _ -> raise exn
         )
  )
```

4.1.1 Arguments processing

4.1.2 Executable format choice: `5l -H`

⟨*constant* CLI.init_text 29a⟩≡ (74b)
```
let init_text  = ref None
```

⟨*constant* CLI.init_round 29b⟩≡ (74b)
```
let init_round = ref None
```

⟨*constant* CLI.init_data 29c⟩≡ (74b)
```
let init_data  = ref None
```

4.1.3 Executable entry point: `5l -E`

⟨*constant* CLI.init_entry 29d⟩≡ (74b)
```
(* note that this is not "main"; we give the opportunity to libc _main
 * to do a few things before calling user's main()
 *)
let init_entry = ref "_main"
```

4.1.4 Computing the linker configuration

⟨*function* CLI.config_of_header_type 29e⟩≡ (74b)
```
let config_of_header_type_and_flags (arch : Arch.t) (header_type : string) : Exec_file.linker_config =
  (* sanity checks *)
  (match !init_data, !init_round with
  | Some x, Some y -> failwith (spf "-D%d is ignored because of -R%d" x y)
  | _ -> ()
```

```
  );
  match header_type with
  | "a.out" | "a.out_plan9" ->
      let header_size = A_out.header_size in
      Exec_file.{
        header_type = Exec_file.A_out;
        arch;
        header_size;
        init_text  =
        (match !init_text  with
          | Some x -> x
          | None -> 4096 + header_size
        );
        init_data = !init_data;
        init_round = (match !init_round with Some x -> x | None -> 4096);
        entry_point = !init_entry;
      }

  | "elf" | "elf_linux" ->
      let header_size = Elf.header_size in
      Exec_file.{
        header_type = Exec_file.Elf;
        arch;
        header_size;
        init_text  =
        (match !init_text  with
          | Some x -> x
          | None ->
              (match arch with
              | Arm -> 0x8000
              | Mips -> 0x400000
              | _ ->
                failwith (spf "arch not supported yet: %s" (Arch.thestring arch))
              ) + header_size
        );
        init_data = !init_data;
        init_round = (match !init_round with Some x -> x | None -> 4096);
        entry_point = !init_entry;
      }
  | s -> failwith (spf "unknown -H option, format not handled: %s" s)
```

4.2 `link()`

⟨*signature* CLI.link 30a⟩≡ (74a)

```
  (* main algorithm; works by side effect on outfile *)
  val link: < Cap.open_in; ..> ->
    Arch.t -> Exec_file.linker_config -> Fpath.t list (* files *) ->
    Chan.o (* outfile *) ->
    unit
```

⟨*function* CLI.link 30b⟩≡ (74b)

```
  let link (caps : < Cap.open_in; ..> ) (arch: Arch.t) (config : Exec_file.linker_config) (files : Fpath.t list)
    match arch with
    | Arch.Arm ->
       link5 caps config files chan
    | Arch.Mips ->
       linkv caps config files chan
    | _ -> failwith (spf "TODO: arch not supported yet: %s" (Arch.thestring arch))
```

⟨*function* CLI.link5 31a⟩≡ (74b)

```
(* will modify chan as a side effect *)
let link5 (caps : < Cap.open_in; ..> ) (config : Exec_file.linker_config) (files : Fpath.t list) (chan : Chan.o
  let arch : Ast_asm5.instr_with_cond Arch_linker.t = {
    Arch_linker.branch_opd_of_instr = Ast_asm5.branch_opd_of_instr;
    Arch_linker.visit_globals_instr = Ast_asm5.visit_globals_instr;
  }
  in
  let (code, data, symbols) = Load.load caps files arch in

  (* mark at least as SXref the entry point *)
  T.lookup (config.entry_point, T.Public) None symbols |> ignore;

  let graph = Resolve.build_graph arch.branch_opd_of_instr symbols code in
  let graph = Rewrite5.rewrite graph in

  let symbols2, (data_size, bss_size) =
    Layout.layout_data symbols data in
  Layout.xdefine symbols2 symbols ("setR12" , T.Public) (T.SData2 (0, T.Data));

  (* can only check for undefined symbols after layout_data which
   * can xdefine new symbols (e.g., etext)
   *)
  Check.check symbols;

  let symbols2, graph(* why modify that??*), text_size =
    Layout5.layout_text symbols2 config.init_text graph in

  let sizes : Exec_file.sections_size =
    Exec_file.{ text_size; data_size; bss_size }
  in
  let init_data =
    match config.init_data with
    | None -> Int_.rnd (text_size + config.init_text) config.init_round
    | Some x -> x
  in
  let config = { config with Exec_file.init_data = Some init_data } in
  Logs.info (fun m -> m "final config is %s"
        (Exec_file.show_linker_config config));

  let instrs = Codegen5.gen symbols2 config graph in
  let datas  = Datagen.gen symbols2 init_data sizes data in
  Execgen.gen config sizes instrs datas symbols2 chan
```

4.3 Loading the objects and libraries: `load()`

⟨*signature* Load.load 31b⟩≡ (83a)

```
(* Load all the object (and library) files and split in code vs data.
 * Will also relocate branching instructions.
 * less: return also LineDirective info per file.
 *)
val load:
  <Cap.open_in; .. > ->
  Fpath.t list ->
  'instr Arch_linker.t ->
  'instr Types.code array * Types.data list * Types.symbol_table
```

⟨*function* Load.load 31c⟩≡ (83c)

```
(* load() performs a few things:
 * - load objects (of course), and libraries (which are essentially objects)
 * - split and concatenate in code vs data all objects,
 * - relocate absolute jumps,
 * - "name" entities by assigning a unique name to every entities
 *   (handling private symbols),
 * - visit all entities (defs and uses) and add them in symbol table
 *
 * alt: take as a parameter (xs : Chan.i list);
 *)
let load (caps : < Cap.open_in; ..>) (xs : Fpath.t list) (arch: 'instr Arch_linker.t) : 'instr T.code array * T

  (* values to return *)
  let code = ref [] in
  let data = ref [] in
  let h = Hashtbl.create 101 in

  let pc : Types.virt_pc ref = ref 0 in
  let idfile = ref 0 in

  let process_obj (obj : 'instr Object_file.t) =
    let file = Fpath.v "TODO" in
    let prog = obj.prog in
    let ipc : Types.virt_pc = !pc in
    incr idfile;


    (* naming and populating symbol table h *)
    prog |> A.visit_globals_program arch.visit_globals_instr
        (fun x -> process_global x h !idfile);

    let (ps, _locs) = prog in
    (* split and concatenate in code vs data, relocate branches,
     * and add definitions in symbol table h.
     *)
    ps |> List.iter (fun (p, line) ->
      match p with
      | A.Pseudo pseudo ->
          (match pseudo with
          | A.TEXT (global, attrs, size) ->
              (* less: set curtext for better error managment *)
              let v = T.lookup_global global h in
              (match v.T.section with
              | T.SXref -> v.T.section <- T.SText !pc;
              | _ -> failwith (spf "redefinition of %s" global.name)
              );
              (* less: adjust autosize? *)
              code |> Stack_.push (T.TEXT (global, attrs, size), (file, line));
              incr pc;
          | A.WORD v ->
              code |> Stack_.push (T.WORD v, (file, line));
              incr pc;

          | A.GLOBL (global, _attrs, size) ->
              let v = T.lookup_global global h in
              (match v.T.section with
              | T.SXref -> v.T.section <- T.SData size;
              | _ -> failwith (spf "redefinition of %s" global.name)
              );
          | A.DATA (global, offset, size, v) ->
```

```
            data |> Stack_.push (T.DATA (global, offset, size, v))
        )

    | A.Virtual instr ->
        code |> Stack_.push (T.V instr, (file, line));
        incr pc;
    | A.Instr instr ->

        let relocate_branch opd =
          match !opd with
          | A.SymbolJump _ | A.IndirectJump _ -> ()
          | A.Relative _ | A.LabelUse _ ->
              raise (Impossible "Relative or LabelUse resolved by assembler")
          | A.Absolute i -> opd := A.Absolute (i + ipc)
        in
        arch.branch_opd_of_instr instr |> Option.iter relocate_branch;
        code |> Stack_.push (T.I instr, (file, line));
        incr pc;

    | A.LabelDef _ -> failwith (spf "label definition in object")
  );
in

(* TODO: split in obj file vs libfile and process libfile at the end
 * and leverage SYMDEF/ranlib index to optimize
*)
xs |> List.iter (fun file ->
  match () with
  | _ when Library_file.is_lib_filename file ->
       let (objs : 'instr Library_file.t)  =
          file |> FS.with_open_in caps Library_file.load in
       (* TODO: filter only the one needed because contain entities
        * used in the object files
        *)
       objs |> List.iter (fun obj -> process_obj obj)
  | _ when Object_file.is_obj_filename file ->
    (* object loading is so much easier in ocaml :) *)
    let (obj : 'instr Object_file.t) =
          file |> FS.with_open_in caps Object_file.load in
    process_obj obj
   (* less: could check valid AST, range of registers, shift values, etc *)

  | _ -> failwith (spf "file %s does not appear to be an obj or lib file"
          !!file)
);

Array.of_list (List.rev !code), List.rev !data, h
```

4.4 Resolving symbols, computing addresses

4.5 Generating the executable: `Execgen.gen()`

⟨*signature* `Execgen.gen` 33⟩≡ (82a)

```
val gen:
  Exec_file.linker_config -> Exec_file.sections_size ->
  Types.word list (* code *) -> Types.byte array (* data *) ->
  Types.symbol_table2 (* for finding entry point *) ->
```

```
    Chan.o ->
    unit
```

⟨*function* Execgen.gen 34a⟩≡ (82b)

```
  let gen (config : Exec_file.linker_config) (sizes : Exec_file.sections_size) (cs : T.word list) (ds : T.byte ar
    let entry_name : string = config.entry_point in
    let entry_addr : T.real_pc =
      try
        let v = Hashtbl.find symbols2 (entry_name, T.Public) in
        (match v with
        | T.SText2 pc  -> pc
        | _ -> failwith (spf "entry not TEXT: %s" entry_name)
        )
      (* normally impossible if propagated correctly, see main.ml *)
      with Not_found ->
        (* less: 5l does instead default to INITTEXT *)
       failwith (spf "entry not found: %s" entry_name)
    in
    let format = config.header_type in
    Logs.info (fun m -> m "saving executable in %s" (Chan.destination chan));

    match format with
    | Exec_file.A_out ->
        (* Header *)
        A_out.write_header config.arch sizes entry_addr chan.oc;

        (* Text section *)
        cs |> List.iter (Endian.Little.output_32 chan.oc);

        (* Data section (no seek to a page boundary; disk image != memory image) *)
        ds |> Array.iter (output_char chan.oc);

        (* todo: symbol table, program counter line table *)
        ()

    | Exec_file.Elf ->
        (* Headers (ELF header + program headers) *)
        let (offset_disk_text, offset_disk_data) =
          Elf.write_headers config sizes entry_addr chan.oc
        in

        (* bugfix: important seek! we are using Int_.rnd in CLI.ml for
         * header_size and so after the program header we might need
         * some padding, hence this seek.
         *)
        seek_out chan.oc offset_disk_text; (* = config.header_size *)
        (* Text section *)
        cs |> List.iter (Endian.Little.output_32 chan.oc);

        (* Data section *)
        seek_out chan.oc offset_disk_data;
        ds |> Array.iter (output_char chan.oc);

        ()
```

4.5.1 Header

⟨*signature* A_out.write_header 34b⟩≡ (87c)

```
  val write_header:
```

```
    Arch.t ->
    Exec_file.sections_size -> int (* entry_addr *) -> out_channel -> unit
```

⟨*function* A_out.write_header 35a⟩≡ (88a)

```
  (* entry point *)
  let write_header (arch : Arch.t) (sizes : Exec_file.sections_size) (entry_addr : int) (chan : out_channel) : un

    (* a.out uses big-endian integers even on low-endian architectures *)
    let output_32 = Endian.Big.output_32 in

    let magic =
      match arch with
      | Arch.Arm -> 0x647 (* Plan9 ARM magic *)
      | Arch.Mips -> failwith "TODO: A_out magic for Mips"
      | Arch.Riscv | Arch.Riscv64 | Arch.X86 | Arch.Amd64 | Arch.Arm64 ->
          failwith (spf "arch not supported yet: %s" (Arch.to_string arch))
    in

    output_32 chan magic;
    output_32 chan sizes.text_size;
    output_32 chan sizes.data_size;
    output_32 chan sizes.bss_size;
    output_32 chan 0; (* TODO: symbol_size, for now stripped *)
    output_32 chan entry_addr;
    output_32 chan 0; (* ?? *)
    output_32 chan 0; (* pc_size *)
    ()
```

4.5.2 Text section

4.5.3 Data section

⟨*signature* Datagen.gen 35b⟩≡ (81a)

```
  (* uses only sizes.data_size *)
  val gen:
    Types.symbol_table2 -> Types.addr (* init_data *) -> Exec_file.sections_size ->
    Types.data list ->
    Types.byte array
```

⟨*function* Datagen.gen 35c⟩≡ (81b)

```
  let gen (symbols2 : T.symbol_table2) (init_data : T.addr) (sizes : Exec_file.sections_size) (ds : T.data list)
    let arr = Array.make sizes.data_size (Char.chr 0) in

    ds |> List.iter (fun d ->
      let T.DATA (global, offset2, size_slice, v) = d in
      let info = Hashtbl.find symbols2 (T.symbol_of_global global) in
      match info with
      | T.SData2 (offset, T.Data) ->
          let base = offset + offset2 in
          (match v with
          | A.Int _ | A.Float _ -> raise Todo
          | A.String s ->
              for i = 0 to size_slice -1 do
                arr.(base + i) <- s.[i]
              done
          | A.Address (A.Global (global2,_offsetTODO)) ->
              let info2 = Hashtbl.find symbols2 (T.symbol_of_global global2) in
              let _i =
                match info2 with
```

```
              | T.SText2 real_pc -> real_pc
              | T.SData2 (offset, _kind) -> init_data + offset
            in
            raise Todo
        | (A.Address (A.Local _ | A.Param _)) -> raise Todo
        )
    | T.SData2 (_, T.Bss) -> raise (Impossible "layout_data missed a DATA")
    | T.SText2 _ -> raise (Impossible "layout_data did this check")
  );
  arr
```

4.5.4 Symbol and line table sections

4.6 Checking for unresolved symbols: `check()`

⟨*signature* `Check.check` 36a⟩≡ (73c)

```
(* Make sure there is no reference to undefined symbols.
 * raise Failure in case of error.
 *)
val check:
  Types.symbol_table -> unit
```

⟨*function* `Check.check` 36b⟩≡ (73d)

```
let check h =
  h |> Hashtbl.iter (fun symb v ->
    match v.section with
    | SXref -> failwith (spf "%s: not defined" (T.s_of_symbol symb))
    | SText _ | SData _ -> ()
  )
```

Chapter 5

Loading Objects

5.1 `load()`

5.2 A global and local program counter: `pc` and `ipc`

Chapter 6

Loading Libraries

6.1 Archive library format: `.a`

⟨*type* `Library_file.t` 38a⟩≡ (90)
```
(* An archive (.a) is really essentially just a list of objects, which in Plan 9
 * are just a list of serialized assembly ASTs
 *)
type 'instr t = 'instr Object_file.t list
```

⟨*signature* `Library_file.load` 38b⟩≡ (90c)
```
(* may raise Object_file.WrongVersion *)
val load:  Chan.i -> 'instr t
```

⟨*signature* `Library_file.save` 38c⟩≡ (90c)
```
val save : 'instr t -> Chan.o -> unit
```

⟨*signature* `Library_file.is_lib_filename` 38d⟩≡ (90c)
```
(* look whether finishes in .oa[5vi] or .oa *)
val is_lib_filename: Fpath.t -> bool
```

⟨*function* `Library_file.save` 38e⟩≡ (90d)
```
let save (x : 'instr t) (chan : Chan.o) : unit =
  Logs.info (fun m -> m "Saving library in %s" (Chan.destination chan));
  output_value chan.oc (Object_file.version, x)
```

⟨*function* `Library_file.load` 38f⟩≡ (90d)
```
let load (chan : Chan.i) : 'instr t =
  Logs.info (fun m -> m "Loading library %s" (Chan.origin chan));
  let (ver, x) = input_value chan.ic in
  if ver <> Object_file.version
  then raise Object_file.WrongVersion
  else x
```

⟨*function* `Library_file.is_lib_filename` 38g⟩≡ (90d)
```
let is_lib_filename (file : Fpath.t) : bool =
  !!file =~ ".*\\.oa[5vi]?$"
```

6.2 Loading libraries manually: `5l libxxx.a`

6.3 Loading libraries semi automatically: `5l -lxxx`

6.3.1 Library search path

6.3.2 `5l -L`

6.3.3 `5l -lxxx`

6.4 Loading libraries automagically: `#pragma lib "libxxx.a"`

Chapter 7

Resolving

7.1 Issues in symbol resolution

7.1.1 Virtual program counter versus real code address

7.1.2 Data address and code size mutual dependency

7.2 Building the code instructions graph: `build_graph()`

⟨*signature* `Resolve.build_graph` 40a⟩≡ (83e)

```
(* !! will modify the code to resolve SymbolJump so take care!!
 * raise Failure in case of error.
*)
val build_graph:
  ('instr -> Ast_asm.branch_operand option) ->
  Types.symbol_table -> 'instr Types.code array -> 'instr Types.code_graph
```

⟨*function* `Resolve.build_graph` 40b⟩≡ (84a)

```
let build_graph branch_opd_of_instr (symbols : T.symbol_table) (xs : 'instr T.code array) : 'instr T.code_graph
  let len = Array.length xs in

  (* stricter: does not make sense to allow empty programs *)
  if len = 0
  then failwith "empty program";

  (* graph initialization *)
  let nodes : 'intr T.node array = xs |> Array.map (fun (instr, loc) ->
    { T.instr = instr; next = None; branch = None; n_loc = loc; real_pc = -1 }
  )
  in

  (* set the next fields *)
  nodes |> Array.iteri (fun i n ->
    if i+1 < len
    then n.next <- Some nodes.(i+1)
  );

  (* set the branch fields *)
  nodes |> Array.iter (fun n ->
    match n.instr with
    | T.TEXT _ | T.WORD _ -> ()
    | T.V (A.RET | A.NOP) -> ()
    | T.I instr ->
        let resolve_branch_operand opd =
```

```
          match !opd with
          | A.IndirectJump _ -> None
          | A.Relative _ | A.LabelUse _ ->
              raise (Impossible "Relative and LabelUse resolved by assembler")
          | A.SymbolJump x ->
              (* resolve branching to symbols *)
              (match (T.lookup_global x symbols).section with
              | T.SText virt_pc ->
                  opd := A.Absolute virt_pc;
                  Some virt_pc
              | T.SXref -> raise (Impossible "SXRef raised by Check.check")
              (* stricter: 5l converts them to SText 0 to avoid reporting
               * multiple times the same error but we fail early instead.
               *)
              | T.SData _ -> failwith "branching to a data symbol"
              )
          | A.Absolute virt_pc -> Some virt_pc
        in
        let adjust_virt_pc (virt_pc : T.virt_pc) =
          if virt_pc < len
          then n.branch <- Some nodes.(virt_pc)
          else failwith (spf "branch out of range %d at %s" virt_pc
                         (T.s_of_loc n.n_loc))
        in
        branch_opd_of_instr instr |> Option.iter (fun opd ->
            resolve_branch_operand opd |> Option.iter adjust_virt_pc)
    );
    nodes.(0)
```

7.2.1 Resolving branch instructions using symbols

7.2.2 Finding instruction at `pc`

7.3 Virtual opcodes rewriting: `rewrite()`

⟨*signature* `Rewrite5.rewrite` 41a⟩≡ (84b)

```
  (* Mostly TEXT/RET rewrite depending whether a function is a "leaf".
   * !!actually works by side effect on graph so take care!!
   * may raise Failure in case of error.
  *)
  val rewrite:
    Types5.code_graph -> Types5.code_graph
```

⟨*function* `Rewrite5.rewrite` 41b⟩≡ (84d)

```
  (* less: rewrite when profiling flag -p *)
  let rewrite (cg : T5.code_graph) : T5.code_graph =

    let is_leaf : A.global Hashtbl_.set = Hashtbl_.create () in

    (* step1: mark is leaf and delete NOPs *)
    cg |> T.iter_with_env (fun (curtext, prev_no_nop) n ->
      match n.T.instr with
      | T.TEXT (ent, _attrs, _size) ->
          Hashtbl.add is_leaf ent true;
          (Some ent, Some n)
      | T.WORD _ -> (curtext, Some n)
      | T.V vinstr ->
          let env =
```

```
        match vinstr with
        (* remove the NOP *)
        | A.NOP ->
            prev_no_nop |> Option.iter (fun prev ->
              prev.T.next <- n.T.next;
            );
            (curtext, prev_no_nop)
        | A.RET -> (curtext, Some n)
      in
      (* NOP and RET should not have branch set *)
      n.branch |> Option.iter (fun _n2 ->
        raise (Impossible "branch should not be set on virtual instr")
      );
      env

  | T.I (instr, _condXXX) ->
      let env =
        match instr with
        | A5.BL _ ->
            curtext |> Option.iter (fun p -> Hashtbl.remove is_leaf p);
            (curtext, Some n)
        | _ -> (curtext, Some n)
      in
      n.branch |> Option.iter (fun n2 ->
        match n2.instr with
        | T.V A.NOP -> n.branch <- Rewrite.find_first_no_nop_node n2.next
        | _ -> ()
      );
      env
) (None, None);

(* step2: transform *)
cg |> T.iter_with_env (fun autosize_opt n ->
  match n.instr with
  | T.TEXT (global, attrs, size) ->
      (* sanity checks *)
      if size mod 4 <> 0
      then failwith (spf "size of locals should be a multiple of 4 for %s"
                      (A.s_of_global global));
      if size < 0
      then failwith "TODO: handle size local -4";

      let autosize_opt =
        if size == 0 && Hashtbl.mem is_leaf global
        then begin
           Logs.debug (fun m -> m "found a leaf procedure without locals: %s"
                        (A.s_of_global global));
           None
        end
        (* + 4 extra space for saving rLINK *)
        else Some (size + 4)
      in
      autosize_opt |> Option.iter (fun autosize ->
        (* for layout text we need to set the final autosize so that
         * size_of_instruction can get passed autosize and can correctly
         * handle instructions using (FP) (the frame pointer).
         *)
        n.instr <- T.TEXT (global, attrs, autosize);
        (* decrement SP and save rLINK in one operation:
         *   MOVW.W R14, -autosize(SP)
```

```
          *)
         let n1 = {
           instr = T.I (A5.MOVE (A.Word, Some A5.WriteAddressBase,
                                 A5.Imsr (A5.Reg A5.rLINK),
                                 A5.Indirect (A5.rSP, -autosize)), A5.AL);
           next = n.next;
           branch = None;
           n_loc = n.n_loc;
           real_pc = -1;
         }
         in
         n.next <- Some n1;
       );
       autosize_opt

   | T.WORD _ -> autosize_opt
   | T.V A.RET ->
       n.instr <- T.I
         ((match autosize_opt with
          (* B (R14) *)
          | None -> A5.B (ref (A.IndirectJump (A5.rLINK)))
          (* increment SP and restore rPC in one operation:
           *    MOVW.P autosize(SP), PC
           *)
          | Some autosize -> A5.MOVE (A.Word, Some A5.PostOffsetWrite,
                                      A5.Indirect (A5.rSP, autosize),
                                      A5.Imsr (A5.Reg A5.rPC))
         ), A5.AL);
       autosize_opt
    | T.V A.NOP -> raise (Impossible "NOP was removed in step1")

    | T.I (
          ( A5.RFE | A5.Arith _ | A5.ArithF _ | A5.MOVE _
          | A5.SWAP _ | A5.B _ | A5.BL _ | A5.Cmp _ | A5.CmpF _ | A5.Bxx _
          | A5.SWI _
          )
          , _) ->
       autosize_opt
) None;

(* works by side effect, still return first node *)
cg
```

7.3.1 Leaf procedure optimisation

7.3.2 ATEXT patching

7.3.3 ARET rewriting

7.3.4 ANOP stripping

7.4 Laying out data: layout_data()

⟨*signature* Layout5.layout_data 43a⟩≡ (82c)

⟨*function* Layout5.layout_data 43b⟩≡ (82d)

7.5 Laying out code: `layout_text()`

⟨*signature* Layout5.layout_text 44a⟩≡ (82c)

```
(* Returns symbol_table2 with SText2 entries populated.
 * Returns also nodes in code_graph with their real_pc field set.
 * Returns also text_size.
 * !! works by side effect on code_graph and symbol_table2, so take care !!
 *)
val layout_text:
  Types.symbol_table2 -> Types.real_pc (* INITTEXT *) -> Types5.code_graph ->
  Types.symbol_table2 * Types5.code_graph * int
```

⟨*function* Layout5.layout_text 44b⟩≡ (82d)

```
let layout_text (symbols2 : T.symbol_table2) (init_text : T.real_pc) (cg : T5.code_graph) : T.symbol_table2 * T

  let pc : T.real_pc ref = ref init_text in
  (* less: could be a None, to be more precise, to detect use of local/param
   * outside a procedure. But anyway at frontier of objects we
   * are considered in TEXT of preceding obj which does not make
   * much sense (we should do this kind of check in check.ml though).
   *)
  let autosize = ref 0 in
  let literal_pools = ref [] in

  cg |> T.iter (fun n ->
    n.real_pc <- !pc;

    let size, poolopt =
      Codegen5.size_of_instruction
          Codegen.{syms = symbols2; autosize = !autosize} n
    in
    if size = 0
    then
      (match n.instr with
      | T.TEXT (global, _, size) ->
          (* remember that rewrite5 has adjusted autosize correctly *)
          autosize := size;
          (* Useful to find pc of entry point and to get the address of a
           * procedure, e.g. in WORD $foo(SB)
           *)
          Hashtbl.add symbols2 (T.symbol_of_global global) (T.SText2 !pc);
      | _ -> failwith (spf "zero-width instruction at %s"
                       (T.s_of_loc n.n_loc))
      );
    poolopt |> Option.iter (fun pool ->
      match pool with
      | Codegen.LPOOL -> Logs.err (fun m -> m "TODO: LPOOL")
      | Codegen.PoolOperand imm_or_ximm ->
          let instr = T.WORD imm_or_ximm in
          (* less: check if already present in literal_pools *)
          let node = Types.{ instr = instr; next = None; branch = None;
                             real_pc = -1;
                             n_loc = n.n_loc } in
          if node.branch <> None
          then raise (Impossible "attaching literal to branching instruction");

          n.branch <- Some node;
          literal_pools |> Stack_.push node;

  );
```

```
      pc := !pc + size;

      (* flush pool *)
      (* todo: complex condition when possible out of offset range *)
      if n.next = None && !literal_pools <> [] then begin
        (* extend cg, and so the cg |> T5.iter, on the fly! *)
        let rec aux (prev : Ast_asm5.instr_with_cond Types.node) xs =
          match xs with
          | [] -> ()
          | x::xs ->
              (* cg grows *)
              prev.next <- Some x;
              aux x xs
        in
        aux n !literal_pools;
        literal_pools := [];
      end;

    );
    if !Flags.debug_layout then begin
      cg |> T.iter (fun (n : Ast_asm5.instr_with_cond Types.node) ->
        Logs.app (fun m -> m  "%d: %s" n.real_pc (T5.show_instr n.instr));
        n.branch |> Option.iter (fun (n : Ast_asm5.instr_with_cond Types.node) ->
          Logs.app (fun m -> m " -> branch: %d" n.real_pc)
        )
      );
    end;

    let final_text = Int_.rnd !pc 8 in
    let textsize = final_text - init_text in
    Hashtbl.replace symbols2 ("etext", T.Public) (T.SText2 final_text);

    symbols2, cg, textsize
```

⟨*signature* `Codegen5.size_of_instruction` 45a⟩≡ (76b)

```
  (* This is used for the code layout. *)
  val size_of_instruction:
    Codegen.env -> Types5.node -> int (* a multiple of 4 *) * Codegen.pool option
```

7.6 Defining special symbols: `etext`, `edata`, and `end`

⟨*function* `Layout5.xdefine` 45b⟩≡ (82d)

7.7 Mutual recursion in layout and `SB/R12`

Chapter 8

ARM Machine Code Generation

8.1 ARM instruction format

8.2 Additional data structures

⟨*type* Codegen5.action 46a⟩≡ (79d)

⟨*type* Codegen5.mem_opcode 46b⟩≡ (79d)
```
type mem_opcode = LDR | STR
```

8.3 Codegen5.gen()

⟨*signature* Codegen5.gen 46c⟩≡ (76b)
```
(* uses only config.init_text and for sanity checking only *)
val gen:
  Types.symbol_table2 -> Exec_file.linker_config -> Types5.code_graph ->
  Types.word list
```

⟨*function* Codegen5.gen 46d⟩≡ (79d)
```
let gen (symbols2 : T.symbol_table2) (config : Exec_file.linker_config) (cg : T5.code_graph) : T.word list =

  let res = ref [] in
  let autosize = ref 0 in

  (* just for sanity checking *)
  let pc = ref config.init_text in

  cg |> T.iter (fun n ->

    let {size; binary; pool = _ }  =
        rules { Codegen.syms = symbols2; autosize = !autosize }
        config.init_data n
    in
    let instrs = binary () in

    if n.real_pc <> !pc
    then raise (Impossible "Phase error, layout inconsistent with codegen");
    if List.length instrs * 4 <> size
    then raise (Impossible (spf "size of rule does not match #instrs at %s"
                              (T.s_of_loc n.n_loc)));

    let xs : Bits.int32 list = instrs |> List.map Assoc.sort_by_val_highfirst in
```

```
      if !Flags.debug_gen
      then begin
        Logs.app (fun m -> m "%s -->" (T5.show_instr n.instr));
        xs |> List.iter (fun x ->
          let w = int_of_bits n x in
          Logs.app (fun m -> m "%s (0x%x)" (Dumper.dump x) w);
        );
        Logs.app (fun m -> m ".");
      end;

      let xs = xs |> List.map (fun x -> int_of_bits n x) in
      res |> Stack_.push xs;

      pc := !pc + size;
      (match n.instr with
      (* after the resolve phase the size of a TEXT is the final autosize *)
      | T.TEXT (_, _, size) -> autosize := size;
      | _ -> ()
      );
    );

    !res |> List.rev |> List.flatten
```

⟨*function* Codegen5.error 47a⟩≡ (79d)

```
  let error node s =
    failwith (spf "%s at %s on %s" s
                (T.s_of_loc node.n_loc)
                (T5.show_instr node.instr)
    )
```

8.4 Codegen5.rules()

⟨*function* Codegen5.rules 47b⟩≡ (79d)

```
  (* conventions (matches the one used (inconsistently) in 51):
   * - rf = register from (called Rm in refcard)
   * - rt = register to   (called Rd in refcard)
   * - r  = register middle (called Rn in refcard)
   *)
  let rules (env : Codegen.env) (init_data : addr option) (node : 'a T.node) =
    match node.instr with
    (* ------------------------------------------------------------------------ *)
    (* Virtual *)
    (* ------------------------------------------------------------------------ *)
     | T.V (A.RET | A.NOP) ->
        raise (Impossible "rewrite should have transformed RET/NOP")

    (* ------------------------------------------------------------------------ *)
    (* Pseudo *)
    (* ------------------------------------------------------------------------ *)
    (* TEXT instructions were kept just for better error reporting localisation *)
    | T.TEXT (_, _, _) ->
        { size = 0; pool = None; binary = (fun () -> []) }

    | T.WORD x ->
        { size = 4; pool = None; binary = (fun () ->
          match x with
          | Float _ -> raise Todo
```

```
      | Int i -> [ [(i land 0xffffffff, 0)] ]
      | String _s ->
          (* stricter? what does 5l do with that? confusing I think *)
          error node "string not allowed with WORD; use DATA"
      | Address (Global (global, _offsetTODO)) ->
          let v = Hashtbl.find env.syms (T.symbol_of_global global) in
          (match v with
           | T.SText2 real_pc -> [ [(real_pc, 0)] ]
           | T.SData2 (offset, _kind) ->
               (match init_data with
               | None -> raise (Impossible "init_data should be set by now")
               | Some init_data -> [ [(init_data + offset, 0)] ]
               )
          )
      | Address (Param _ | Local _) -> raise Todo
    )}

| T.I (instr, cond) ->
  (match instr with
  | ArithF _ | CmpF _ -> raise Todo
  (* ---------------------------------------------------------------------- *)
  (* Arithmetics *)
  (* ---------------------------------------------------------------------- *)
  | Arith ((AND|ORR|EOR|ADD|SUB|BIC|ADC|SBC|RSB|RSC|MVN|MOV) as op, opt,
           from, middle, (R rt)) ->
      let r =
        if (op = MVN || op = MOV)
        then 0
        else
          match middle with
          | None -> rt
          | Some (R x) -> x
      in
      let from_part =
        match from with
        | Reg (R rf)      -> [(rf, 0)]
        | Shift (a, b, c) -> gshift a b c
        | Imm i ->
            (match immrot i with
            | Some (rot, v) -> [(1, 25); (rot, 8); (v, 0)]
            | None -> error node "TODO: LCON"
            )
      in
      { size = 4; pool = None; binary = (fun () ->
        [[gcond cond; gop_arith op] @ gsetbit opt @ [(r, 16); (rt, 12)]
          @ from_part]
      )}

  (* SLL I, [R], RT -> MOV (R << c), RT *)
  | Arith ((SLL|SRL|SRA) as op, opt, from, middle, (R rt)) ->
      let r =
        match middle with
        | None -> rt
        | Some (R x) -> x
      in
      let from_part =
        match from with
        | Imm i ->
            if i >= 0 && i <= 31
            then [(i, 7)]
```

```
          (* stricter: failwith, not silently truncate *)
          else error node (spf "shit value out of range %d" i)
      | Reg (R rf) -> [(rf, 8); (1, 4)]
      (* stricter: I added that *)
      | Shift _ -> error node "bitshift on shift operation not allowed"
    in
    { size = 4; pool = None; binary = (fun () ->
      [[gcond cond; gop_arith MOV] @ gsetbit opt @ [(rt, 12); gop_shift op]
        @ from_part @ [(r, 0)]]
    )}

| Arith (MUL, opt, from, middle, (R rt)) ->
    let rf =
      match from with
      | Reg (R rf) -> rf
      (* stricter: better error message *)
      | Shift _ | Imm _ ->
          error node "MUL can take only register operands"
    in
    let r =
      match middle with
      | None -> rt
      | Some (R x) -> x
    in
    (* ?? *)
    let (r, rf) = if rt = r then (rf, rt) else (r, rf) in

    { size = 4; pool = None; binary = (fun () ->
      [[gcond cond; (0x0, 21)] @ gsetbit opt
        @ [(rt, 16); (rf, 8); (0x9, 4); (r, 0) ]]
    )}



| Cmp (op, from, (R r)) ->
    let from_part =
      match from with
      | Reg (R rf) -> [(rf, 0)]
      | Shift (a, b, c) -> gshift a b c
      | Imm i ->
          (match immrot i with
          | Some (rot, v) -> [(1, 25); (rot, 8); (v, 0)]
          | None -> error node "TODO"
          )
    in
    { size = 4; pool = None; binary = (fun () ->
      [[gcond cond] @ gop_cmp op @ [(r, 16); (0, 12)] @ from_part]
    )}

| MOVE (Word, None, Imsr from, Imsr (Reg (R rt))) ->
    let from_part =
      match from with
      | Reg (R rf) -> [(rf, 0)]
      | Shift (a, b, c) -> gshift a b c
      | Imm i ->
          (match immrot i with
          | Some (rot, v) -> [(1, 25); (rot, 8); (v, 0)]
          | None -> error node "TODO"
          )
    in
```

```
    let r = if !Flags.kencc_compatible then rt else 0 in
    { size = 4; pool = None; binary = (fun () ->
      [[gcond cond; gop_arith MOV; (r, 16); (rt, 12)] @ from_part]
    )}

(* MOVBU R, RT -> ADD 0xff, R, RT *)
| MOVE (Byte U, None, Imsr (Reg (R r)), Imsr (Reg (R rt))) ->
    { size = 4; pool = None; binary = (fun () ->
      [[gcond cond; (1, 25); gop_arith AND; (r, 16); (rt, 12); (0xff, 0)]]
    )}

(* MOVB RF, RT  -> SLL 24, RF, RT; SRA 24, RT, RT -> MOV (RF << 24), RT;...
 * MOVH RF, RT  -> SLL 16, RF, RT; SRA 16, RT, RT -> ...
 * MOVHU RF, RT -> SLL 16, RF, RT; SRL 16, RT, RT ->
 *)
| MOVE ((Byte _|HalfWord _)as size, None, Imsr(Reg(R rf)),Imsr(Reg(R rt)))->
    let rop =
      match size with
      | Byte U | HalfWord U -> SRL
      | Byte S   | HalfWord S -> SRA
      | Word -> raise (Impossible "size matched in pattern")
    in
    let sh =
      match size with
      | Byte _ -> 24
      | HalfWord _ -> 16
      | Word -> raise (Impossible "size matched in pattern")
    in
    { size = 8; pool = None; binary = (fun () ->
      [
        [gcond cond; gop_arith MOV; (rt, 12); gop_shift SLL; (sh,7);(rf,0)];
        [gcond cond; gop_arith MOV; (rt, 12); gop_shift rop; (sh,7);(rt,0)];
      ]
    )}

| Arith ((DIV|MOD), _, _, _, _) -> error node "TODO: DIV/MOD"

(* ------------------------------------------------------------------------ *)
(* Control flow *)
(* ------------------------------------------------------------------------ *)
| B x ->
    if cond <> AL
    then raise (Impossible "B should always be with AL");
    { size = 4; pool = Some LPOOL; binary = (fun () ->
      match !x with
      | Absolute _ -> [ gbranch_static node AL false ]
      (* B (R) -> ADD 0, R, PC *)
      | IndirectJump (R r) ->
          let (R rt) = rPC in
          [ [gcond AL; (1, 25); gop_arith ADD; (r, 16); (rt, 12); (0, 0)] ]
      | _ -> raise (Impossible "5a or 5l should have resolved this branch")
    )}
| BL x ->
    (match !x with
    | Absolute _ ->
        { size = 4; pool = None; binary = (fun () ->
          [ gbranch_static node AL true ]
        )}
    (* BL (R) -> ADD $0, PC, LINK; ADD $0, R, PC *)
    | IndirectJump (R r) ->
```

```
       { size = 8; pool = None; binary = (fun () ->
         let (R r2) = rPC in
         let (R rt) = rLINK in
          [
            (* Remember that when PC is involved in input operand
             * there is an implicit +8 which is perfect for our case.
             *)
            [gcond cond;(1, 25); gop_arith ADD; (r2, 16); (rt, 12); (0, 0)];
            [gcond cond;(1, 25); gop_arith ADD; (r, 16);  (r2, 12); (0, 0)];
          ]
         )}
    | _ -> raise (Impossible "5a or 5l should have resolved this branch")
    )

| Bxx (cond2, x) ->
    if cond <> AL
    then raise (Impossible "Bxx should always be with AL");
    (match !x with
    | Absolute _ ->
        { size = 4; pool = None; binary = (fun () ->
          [ gbranch_static node cond2 false ]
        )}
    (* stricter: better error message at least? *)
    | IndirectJump _ -> error node "Bxx supports only static jumps"
    | _ -> raise (Impossible "5a or 5l should have resolved this branch")
    )

(* ------------------------------------------------------------------------ *)
(* Memory *)
(* ------------------------------------------------------------------------ *)

(* Address *)
| MOVE (Word, None, Ximm ximm, Imsr (Reg (R rt))) ->
    (match ximm with
    | Int _ | Float _ ->
       failwith "TODO: ?? because of refactor of imm_or_ximm"
    | String _ ->
        (* stricter? what does 5l do with that? confusing I think *)
        error node "string not allowed in MOVW; use DATA"
    | Address (Global (global, _offsetTODO)) ->
        let from_part_when_small_offset_to_R12 =
          try
            let v = Hashtbl.find env.syms (T.symbol_of_global global) in
            match v with
            | T.SData2 (offset, _kind) ->
                let final_offset = offset_to_R12 offset in
                (* super important condition! for bootstrapping
                 * setR12 in MOVW $setR12(SB), R12 and not
                 * transform it in ADD offset_set_R12, R12, R12.
                 *)
                if final_offset = 0
                then None
                else immrot final_offset
            | T.SText2 _ -> None
          (* layout_text has not been fully done yet so we may have
           * the address of a procedure we don't know yet
           *)
          with Not_found -> None
        in
        (match from_part_when_small_offset_to_R12 with
```

```
        | Some (rot, v) ->
          (* MOVW $x(SB), RT -> ADD $offset_to_r12, R12, RT  *)
          { size = 4; pool = None; binary = (fun () ->
            let (R r) = rSB in
            [[gcond cond; (1, 25); gop_arith ADD; (r, 16); (rt, 12);
              (rot, 8); (v, 0)]]
        )}
        | None ->
          (* MOVW $L(SB), RT -> LDR x(R15), RT *)
          { size = 4; pool=Some(PoolOperand(ximm)); binary=(fun () ->
            [ gload_from_pool node cond (R rt) ]
          )}
        )
    | Address (Local _ | Param _) -> raise Todo
    )

(* Load *)

| MOVE ((Word | Byte U) as size, opt, from, Imsr (Reg rt)) ->
    (match from with
    | Imsr (Imm _ | Reg _) ->
        if size = Word
        then raise (Impossible "pattern covered before")
        else error node "illegal combination?"
    | Imsr (Shift _) -> error node "TODO"
    | Ximm _ ->
        if size = Word
        then raise (Impossible "pattern covered before")
        else error node "illegal combination"
    | Indirect _ | Entity _ ->
        let (rbase, offset) =
          base_and_offset_of_indirect node env.syms env.autosize from in
        if immoffset offset
        then
          { size = 4; pool = None; binary = (fun () ->
            [ gmem cond LDR size opt (Left offset) rbase rt ]
          )}
        else
          error node "TODO: Large offset"
    )

(* Store *)

(* note that works for Byte Signed and Unsigned here *)
| MOVE ((Word | Byte _) as size, opt, Imsr (Reg rf), dest) ->
    (match dest with
    | Imsr (Reg _) -> raise (Impossible "pattern covered before")
    (* stricter: better error message *)
    | Imsr _ | Ximm _ ->
        error node "illegal to store in an (extended) immediate"
    | Indirect _ | Entity _ ->
        let (rbase, offset) =
          base_and_offset_of_indirect node env.syms env.autosize dest in
        if immoffset offset
        then
          { size = 4; pool = None; binary = (fun () ->
            [ gmem cond STR size opt (Left offset) rbase rf ]
          )}
        else
          error node "TODO: store with large offset"
```

```
    )

  (* Swap *)
  | SWAP _ -> error node "TODO: SWAP"

  (* Half words and signed bytes *)
  | MOVE ((HalfWord _ | Byte _), _opt, _from, _dest) ->
      error node "TODO: half"

  | MOVE (Word, _opt, _from, _dest) ->
     (* stricter: better error message *)
     error node "illegal combination: at least one operand must be a register"

  (* ------------------------------------------------------------------------ *)
  (* System *)
  (* ------------------------------------------------------------------------ *)
  | SWI i ->
      if i <> 0
      then error node (spf "SWI does not use its parameter under Plan 9/Linux");

      { size = 4; pool = None; binary = (fun () ->
        [ [gcond cond; (0xf, 24)] ]
      )}
  (* RFE -> MOVM.S.W.U 0(r13),[r15] *)
  | RFE ->
      { size = 4; pool = None; binary = (fun () ->
        [ [(0xe8fd8000, 0)] ]
      )}
  (* ------------------------------------------------------------------------ *)
  (* Other *)
  (* ------------------------------------------------------------------------ *)
  (*    | _ -> error node "illegal combination"*)
  )
```

⟨*function* Codegen5.size_of_instruction 53⟩≡ (79d)

```
let size_of_instruction (env : Codegen.env) (node : T5.node) : int (* a multiple of 4 *) * pool option =
  let action  = rules env None node in
  action.size, action.pool
```

8.5 Pseudo opcodes

8.5.1 ATEXT

8.5.2 AWORD

8.6 Operand subclasses and instoffset

8.6.1 D_CONST, C_xCON, and immrot()

8.6.2 D_OREG, C_xOREG, and immaddr()

8.6.3 D_OREG with globals, C_xEXT

8.6.4 D_OREG with automatics, C_xAUTO

8.6.5 D_ADDR, C_xxCON

8.7 Arithmetic and logic opcodes

8.7.1 Register-only operands

8.7.2 Small rotatable immediate constant operand

8.7.3 Bitshifted register

8.7.4 Bitshift opcodes

8.7.5 Byte and half word extractions

8.7.6 Multiplication opcodes

8.7.7 Large constant operand, REGTMP, and literal pools

8.8 Control flow opcodes

8.8.1 Direct jumps

8.8.2 Indirect jumps

8.9 Memory opcodes

8.9.1 Load

8.9.2 Store

8.9.3 Swaps

8.9.4 Symbol addresses

8.9.5 Half words and signed bytes

8.10 Software interrupt opcodes

8.11 Literal Pools

Chapter 9

Debugging Support

9.1 `asmb()` and the debugging tables

9.2 Executable symbol table

9.2.1 Symbol table format: `putsymb()`

9.2.2 Globals and procedures symbols: `asmsym()`

9.2.3 Stack variables symbols

9.2.4 Filename and line origin symbols

9.3 File and line information

9.3.1 Locations in objects: `AHISTORY`

9.3.2 Locations in `5l` memory

9.3.3 Locations in executables

Chapter 10

Profiling Support

10.1 `5l -p` and `_mainp`

10.2 `5l -p -1` and `__mcount`

10.3 `5l -p` and `_profin()/_profout()`

10.4 Disabling profiling attribute: `NOPROF`

Chapter 11

Advanced Topics

11.1 Dynamic linking

11.1.1 Export table: `5l -x`

11.1.2 Dynamic loading: `5l -u`

11.1.3 `SUNDEF`

11.1.4 Relocatable address: `C_ADDR`

11.2 Position independent code (PIC)

11.3 Optimizations

11.3.1 Opcode rewriting

11.3.2 Operand rewriting

11.3.3 Small data first

11.3.4 Compacting chains of `AB`, `brloop()`

11.3.5 Removing useless instructions, `follow()`

11.4 Overriding symbol attribute: `DUPOK`

11.5 Other executable formats

11.5.1 ELF (Linux)

⟨*signature* `Elf.header_size` 58a⟩≡ (88b)
```
val header_size: int
```

⟨*signature* `Elf.write_headers` 58b⟩≡ (88b)
```
(* return offset_disk_text and offset_disk_data for the caller to use seek_out *)
val write_headers:
  Exec_file.linker_config -> Exec_file.sections_size -> int (* entry_addr *) ->
  out_channel -> int * int
```

⟨*constant* `Elf.exec_header_32_size` 59a⟩≡ (88c)
```
let exec_header_32_size = 52
```

⟨*constant* `Elf.program_header_32_size` 59b⟩≡ (88c)
```
let program_header_32_size = 32
```

⟨*constant* `Elf.section_header_32_size` 59c⟩≡ (88c)
```
let section_header_32_size = 40
```

⟨*constant* `Elf.nb_program_headers` 59d⟩≡ (88c)
```
(* Text, Data, and Symbol table *)
let nb_program_headers = 2 (* TODO 3 *)
```

⟨*constant* `Elf.header_size` 59e⟩≡ (88c)
```
let header_size =
  Int_.rnd (exec_header_32_size + nb_program_headers * program_header_32_size)
  16
```

⟨*type* `Elf.ident_class` 59f⟩≡ (88c)
```
type ident_class =
  | CNone
  | C32
  | C64
  | CNum (* ?? *)
[@@warning "-37"]
```

⟨*type* `Elf.byte_order` 59g⟩≡ (88c)
```
type byte_order =
  | BNone
  | BLSB (* Least Significant Bit *)
  | BMSB (* Most Significant bit *)
  | BNum (* ?? *)
[@@warning "-37"]
```

⟨*type* `Elf.ident_version` 59h⟩≡ (88c)
```
type ident_version =
  | VNone
  | VCurrent
[@@warning "-37"]
```

⟨*type* `Elf.elf_type` 59i⟩≡ (88c)
```
type elf_type =
  | TNone
  | TRel
  | TExec
  | TDyn
  | TCore
[@@warning "-37"]
```

⟨*type* `Elf.machine` 59j⟩≡ (88c)
```
type machine =
  | MNone

  (* main one; the one we want to support in xix *)
  | MI386
  | MAmd64
  | MArm
  | MArm64
  | MMips
  | MRiscv
```

```
    (* | MRiscv64? *)

    (* other: *)
    | MM32
    | MSparc
    | MM68K
    | MM88K
    | MI486
    | MI860
    | MS370
    | MMipsr4K
    | MSparc64
    | MPower
    | MPower64
  [@@warning "-37"]
```

⟨*type* Elf.program_header_type 60a⟩≡ (88c)

```
  type program_header_type =
    | PH_None
    | PH_PT_Load
    | PH_Dynamic
    | PH_Interp
    | PH_Note
    | PH_Shlib
    | PH_Phdr
  [@@warning "-37"]
```

⟨*type* Elf.program_header_protection 60b⟩≡ (88c)

```
  type program_header_protection =
    | R
    | W
    | X
```

⟨*function* Elf.byte_of_class 60c⟩≡ (88c)

```
  let byte_of_class (class_ : ident_class) : int =
    match class_ with
    | CNone -> 0
    | C32 -> 1
    | C64 -> 2
    | CNum -> 3
```

⟨*function* Elf.byte_of_byte_order 60d⟩≡ (88c)

```
  let byte_of_byte_order (bo : byte_order) : int =
    match bo with
    | BNone -> 0
    | BLSB -> 1
    | BMSB -> 2
    | BNum -> 3
```

⟨*function* Elf.int_of_version 60e⟩≡ (88c)

```
  let int_of_version (v : ident_version) : int =
    match v with
    | VNone -> 0
    | VCurrent -> 1
```

⟨*function* Elf.int_of_elf_type 60f⟩≡ (88c)

```
  let int_of_elf_type (t : elf_type) : int =
    match t with
    | TNone -> 0
    | TRel -> 1
    | TExec -> 2
    | TDyn -> 3
    | TCore -> 4
```

⟨*function* Elf.int_of_machine 61a⟩≡ (88c)
```
let int_of_machine (m : machine) : int =
  match m with
  | MNone -> 0
  | MM32 -> 1
  | MSparc -> 2
  | MI386 -> 3
  | MM68K -> 4
  | MM88K -> 5
  | MI486 -> 6
  | MI860 -> 7
  | MMips -> 8
  | MS370 -> 9
  | MMipsr4K -> 10

  | MSparc64 -> 18
  | MPower -> 20
  | MPower64 -> 21

  | MArm -> 40
  | MAmd64 -> 62
  | MArm64 -> 183

  | MRiscv -> failwith "TODO: MRiscv"
```

⟨*function* Elf.int_of_program_header_type 61b⟩≡ (88c)
```
let int_of_program_header_type (ph : program_header_type) : int =
  match ph with
  | PH_None -> 0
  | PH_PT_Load -> 1
  | PH_Dynamic -> 2
  | PH_Interp -> 3
  | PH_Note -> 4
  | PH_Shlib -> 5
  | PH_Phdr -> 6
```

⟨*function* Elf.int_of_prot 61c⟩≡ (88c)
```
let int_of_prot (prot : program_header_protection) : int =
  match prot with
  | R -> 0x4
  | W -> 0x2
  | X -> 0x1
```

⟨*function* Elf.int_of_prots 61d⟩≡ (88c)
```
let int_of_prots xs =
  List.fold_left (fun acc e -> acc + int_of_prot e) 0 xs
```

⟨*function* Elf.write_ident 61e⟩≡ (88c)
```
(* first 16 bytes *)
let write_ident (bo : byte_order) (class_ : ident_class) (chan: out_channel) : unit =
  (* take care, using "\177ELF" like in C to OCaml does not work because
   * in C \177 is interpreted as an octal number but in OCaml it's an int
   * and 0o177 is different from 177. Simpler to use 0x7f.
   *)
  output_byte chan 0x7f;
  output_string chan "ELF";
  output_byte chan (byte_of_class class_);
  output_byte chan (byte_of_byte_order bo);
  output_byte chan (int_of_version VCurrent);
  output_byte chan 0; (* osabi = SYSV; 255 = boot/embedded/standalone? *)
```

```
    output_byte chan 0; (* abiversion = 3 *)
    output_string chan "\000\000\000\000\000\000\000";
    ()
```

⟨*function* `Elf.program_header_32` 62a⟩≡ (88c)

```
  let program_header_32 (endian: Endian.t) (ph: program_header_type)
    offset (vaddr, paddr) (filesz, memsz) prots align (chan : out_channel) =
    let (_, output_32) = Endian.output_functions_of_endian endian in
    output_32 chan (int_of_program_header_type ph);
    output_32 chan offset;
    output_32 chan vaddr;
    output_32 chan paddr;
    output_32 chan filesz;
    output_32 chan memsz;
    output_32 chan (int_of_prots prots);
    output_32 chan align
```

⟨*function* `Elf.write_headers` 62b⟩≡ (88c)

```
  (* entry point *)
  let write_headers (config : Exec_file.linker_config)
   (sizes : Exec_file.sections_size) (entry_addr : int) (chan : out_channel) : int * int =
    let arch = config.arch in

    (* ELF ident part (first 16 bytes) *)

    let endian = Arch.endian_of_arch arch in
    let bo : byte_order =
      match endian  with
      | Endian.Little -> BLSB
      | Endian.Big -> BMSB
    in
    let class_ : ident_class =
      match Arch.bits_of_arch arch with
      | Arch.Arch32 -> C32
      | Arch.Arch64 -> C64
    in
    write_ident bo class_ chan;

    (* Rest of ELF header (36 bytes => total 52 bytes) *)

    let mach : machine =
      match arch with
      | Arch.Arm -> MArm
      | Arch.Arm64 -> MArm64
      | Arch.Mips -> MMips
      | Arch.Riscv -> MRiscv
      | Arch.Riscv64 -> failwith "TODO: Riscv64"
      | Arch.X86 -> MI386 (* what about MI486? *)
      | Arch.Amd64 -> MAmd64
    in
    let output_16, output_32 = Endian.output_functions_of_endian endian in
    output_16 chan (int_of_elf_type TExec);
    output_16 chan (int_of_machine mach);
    output_32 chan (int_of_version VCurrent); (* again? *)
    output_32 chan entry_addr;
    output_32 chan exec_header_32_size; (* offset to first phdr *)
    output_32 chan 0; (* TODO: offset to first shdr HEADR+textsize+datsize+symsize *)
    (match arch with
    | Arch.Arm ->
          (* version5 EABI for Linux *)
```

```
      output_32 chan 0x5000200;
| _ -> output_32 chan 0
);
output_16 chan exec_header_32_size;
output_16 chan program_header_32_size;
output_16 chan nb_program_headers; (* # of Phdrs *)
output_16 chan section_header_32_size;
output_16 chan 0; (* # of Shdrs, TODO 3 *)
output_16 chan 0; (* Shdr table index, TODO 2 *)

Logs.debug (fun m -> m "after ELF header at pos %d" (pos_out chan));

(* Program headers *)

(* Text *)
let offset_disk_text = config.header_size in
program_header_32 endian PH_PT_Load
 offset_disk_text (config.init_text, config.init_text)
  (sizes.text_size, sizes.text_size) [R; X] config.init_round chan;

(* ELF Linux constrains that virtual
 * address modulo a page must match file offset modulo
 * a page, so simpler to start data at a page boundary
 *)
let offset_disk_data =
  Int_.rnd (config.header_size + sizes.text_size) config.init_round
in
let init_data =
  match config.init_data with
  | None ->
      raise (Impossible "init_data should be set by now after layout_text")
  | Some x -> x
in
program_header_32 endian PH_PT_Load
  offset_disk_data (init_data, init_data)
   (sizes.data_size, sizes.data_size + sizes.bss_size) [R; W; X]
   config.init_round chan;

Logs.debug (fun m -> m "after Program headers at pos %d" (pos_out chan));
offset_disk_text, offset_disk_data
```

11.5.2 OMach (mac OS)

11.5.3 PE (Windows)

11.6 Other instructions

11.6.1 Float operations

11.6.2 Division

11.6.3 Long multiplication

11.6.4 Multiple registers move

11.6.5 Status register

11.6.6 Half words and bytes moves since ARMv4

11.6.7 Shifted moves

11.7 Compiler-only pseudo opcodes

Chapter 12

Conclusion

Appendix A

Debugging

A.1 Dumpers

A.2 Verbose mode: `5l -v`

A.3 Objects loading debugging: `5l -W`

A.4 Machine code generation debugging: `5l -a`

Appendix B

Error Management

Appendix C

Profiling

Appendix D

Linker-related Programs

D.1 nm

⟨*type* Nm.caps 69a⟩≡ (92)
```
type caps = < Cap.open_in; Cap.stdout >
```

⟨*constant* Nm.usage 69b⟩≡ (92)
```
let usage =
  "usage: nm [-options] file ..."
```

⟨*function* Nm.visit_obj 69c⟩≡ (92)
```
(* TODO: visit also the uses, not just the defs *)
let visit_obj (caps : < Cap.stdout; .. >) (obj : 'instr Object_file.t) : unit =
  let (xs, _locs) = obj.prog in
  (* visit the defs *)
  xs |> List.iter (fun (line, _loc) ->
      match line with
      | Pseudo (TEXT (glob, _attrs, _int)) ->
          let ident = A.s_of_global glob in
          Console.print caps (spf " T %s" ident)
      | Pseudo (GLOBL (glob, _attrs, _int)) ->
          let ident = A.s_of_global glob in
          Console.print caps (spf " D %s" ident)
      | Pseudo (DATA _ | WORD _) -> ()
      | Virtual (RET | NOP) -> ()
      (* TODO: visit the uses *)
      | Instr _x -> ()
      | LabelDef _ -> raise (Impossible "objects should not have LabelDef")
  )
```

⟨*function* Nm.nm 69d⟩≡ (92)
```
let nm (caps : < caps; ..> ) (file : Fpath.t) : unit =
  match () with
  | _ when Object_file.is_obj_filename file ->
      let obj : 'instr Object_file.t =
        FS.with_open_in caps Object_file.load file
      in
      visit_obj caps obj
  | _ when Library_file.is_lib_filename file ->
      let objs : 'instr Library_file.t =
        FS.with_open_in caps Library_file.load file
      in
      objs |> List.iter (visit_obj caps)
  | _ ->
      failwith "TODO: handle exec format"
```

⟨*function* `Nm.main` 70a⟩≡ (92)

```
let main (caps : <caps; ..>) (argv : string array) : Exit.t =
  let infiles = ref [] in

  let backtrace = ref false in
  let level = ref (Some Logs.Warning) in

  let options = [
   (* TODO: support the many nm flags *)
  ] |> Arg.align
  in
  (try
    Arg.parse_argv argv options
      (fun f -> infiles := Fpath.v f::!infiles) usage;
  with
  | Arg.Bad msg -> UConsole.eprint msg; raise (Exit.ExitCode 2)
  | Arg.Help msg -> UConsole.print msg; raise (Exit.ExitCode 0)
  );
  Logs_.setup !level ();
  Logs.info (fun m -> m "nm ran from %s" (Sys.getcwd()));

  (match List.rev !infiles with
  | [] ->
      Arg.usage options usage;
      Exit.Code 1
  | xs ->
     try
        (* the main call *)
        xs |> List.iter (nm caps);
        Exit.OK
     with exn ->
     if !backtrace
     then raise exn
     else
      (match exn with
      | Failure s ->
          Logs.err (fun m -> m "%s" s);
          Exit.Code 1
      | _ -> raise exn
      )
  )
```

⟨*constant* `Nm._` 70b⟩≡ (92)

```
let _ =
  Cap.main (fun (caps : Cap.all_caps) ->
     Exit.exit caps (Exit.catch (fun () -> main caps (CapSys.argv caps)))
  )
```

D.2 ar

⟨*type* `Ar.caps` 70c⟩≡ (91)

```
type caps = < Cap.open_in; Cap.open_out >
```

⟨*constant* `Ar.usage` 70d⟩≡ (91)

```
let usage =
  "usage: oar [-options] objects"
```

⟨*function* Ar.archive 71a⟩≡ (91)

```
let archive (caps : < Cap.open_in; ..> ) (objfiles : Fpath.t list) (chan : Chan.o) : unit =
  (* sanity checks *)
  (* TODO? sanity check all of same arch? *)
  objfiles |> List.iter (fun file ->
    if not (Object_file.is_obj_filename file)
    then failwith (spf "The file extension of %s does not match an object file"
          !!file)
  );
  let libfile = Fpath.v (Chan.destination chan) in
  if not (Library_file.is_lib_filename libfile)
  then failwith (spf "The file extension of %s does not match a library file"
          !!libfile);

  let xs = objfiles |> List.map (FS.with_open_in caps Object_file.load) in
  Library_file.save xs chan
```

⟨*function* Ar.main 71b⟩≡ (91)

```
let main (caps : <caps; ..>) (argv : string array) : Exit.t =
  let infiles = ref [] in
  let outfile = ref (Fpath.v "lib.oa") in

  let level = ref (Some Logs.Warning) in
  (* for debugging *)
  let backtrace = ref false in

  let options = [
    "-o", Arg.String (fun s -> outfile := Fpath.v s),
    spf " <file> output file (default is %s)" !!(!outfile);

    (* pad: I added that *)
    "-v", Arg.Unit (fun () -> level := Some Logs.Info),
     " verbose mode";
    "-verbose", Arg.Unit (fun () -> level := Some Logs.Info),
    " verbose mode";
    "-debug", Arg.Unit (fun () -> level := Some Logs.Debug),
    " guess what";
    "-quiet", Arg.Unit (fun () -> level := None),
    " ";

    (* pad: I added that *)
    "-backtrace", Arg.Set backtrace,
    " dump the backtrace after an error";

  ] |> Arg.align
  in
  (try
    Arg.parse_argv argv options
      (fun f -> infiles := Fpath.v f::!infiles) usage;
  with
  | Arg.Bad msg -> UConsole.eprint msg; raise (Exit.ExitCode 2)
  | Arg.Help msg -> UConsole.print msg; raise (Exit.ExitCode 0)
  );
  Logs_.setup !level ();
  Logs.info (fun m -> m "ar ran from %s" (Sys.getcwd()));

  (match List.rev !infiles with
  | [] ->
      Arg.usage options usage;
      Exit.Code 1
```

```
  | xs ->
     try
        (* the main call *)
        !outfile |> FS.with_open_out caps (fun chan ->
          archive caps xs chan
        );
        Exit.OK
  with exn ->
    if !backtrace
    then raise exn
    else
      (match exn with
      | Failure s ->
          Logs.err (fun m -> m "%s" s);
          Exit.Code 1
      | _ -> raise exn
      )
  )
```

⟨*constant* `Ar._` 72⟩≡ (91)

```
let _ =
  Cap.main (fun (caps : Cap.all_caps) ->
     let argv = CapSys.argv caps in
     Exit.exit caps (Exit.catch (fun () -> main caps argv))
  )
```

Appendix E

Extra Code

E.1 Arch_linker.ml

⟨*type* Arch_linker.t 73a⟩≡ (73b)
```
type 'instr t = {
  branch_opd_of_instr: 'instr -> A.branch_operand option;
  visit_globals_instr: (A.global -> unit) -> 'instr -> unit;
}
```

⟨Arch_linker.ml 73b⟩≡
```
module A = Ast_asm

(* Arch-specific methods allowing to factorize code in the linker
 * (e.g., in Load.ml)
 * alt: use a functor, but can't with ocaml-light and records are fine!
 *)
```
⟨*type* Arch_linker.t 73a⟩

E.2 Check.mli

⟨Check.mli 73c⟩≡

⟨*signature* Check.check 36a⟩

E.3 Check.ml

⟨Check.ml 73d⟩≡
```
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common

open Types
module T = Types
```

⟨*function* Check.check 36b⟩

```
(* todo: could also check validity of object file:
 *  - no duplicate DATA for same global.
 *    for instance with int foo = 1; int foo = 2;  5c does not say anything,
 *    but we should warn at least in linker because generated assembly
 *    contains DATA concerning same global!
 * less:
```

```
 *  - registers are in range,
 *  - integers are in range,
 *    ex with immediate in shifting operation:
 *    if i >= 0 && i <= 31
 *    then ...
 *    else failwith "shift value out of range"
 *  - that cond is AL for B and Bxx,
 *  - that use Local or Param only when inside a Text (and for Local
 *    that if fits the size specified),
 *    | None -> error loc "use of parameter outside of procedure"
 *    | None -> error loc "use of local outside of procedure"
 *  - ...
 *
 *  See use of error() in codegen5.ml, or notes about 5l in ocaml in
 *  Linker.nw for more invariants to check.
 *)
```

E.4 CLI.mli

⟨CLI.mli 74a⟩≡

```
⟨type CLI.caps 27b⟩

⟨signature CLI.main 27c⟩

⟨signature CLI.link 30a⟩
```

E.5 CLI.ml

⟨CLI.ml 74b⟩≡

```
(* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)
open Common
open Fpath_.Operators

module T = Types

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* An OCaml port of 5l/vl, the Plan 9 ARM/MIPS linkers.
 *
 * Main limitations compared to 5l/vl/...:
 * - no -E digit
 *   (What was it anyway?)
 * - no optimisation about small data, strings in text section
 *   (really gain?)
 * - no extensions not yet understood (import/export, dynamic linking)
 *   (not sure it was used by any Plan 9 programs)
 * - address of parameter or local is not supported
 *   (Why would you want that? Does 5c generate that?)
 *   update: actually I support it now no?
 *
 * Main limitations compared to 5l:
 * - no half-word specialized instructions and immhalf()
 *   (rare instructions anyway?)
 * Main limitations compared to vl:
 *  - no sched/nosched support (no scheduling)
```

```
 *    (but better not to be too smart? in fact vl was only linker doing that)
 *
 * Better than 5l/vl/...:
 * - greater code reuse across all linkers thanks to:
 *    * use of marshalling for objects and libraries
 *    * factorized analysis such as Resolve.build_graph, Datagen.gen,
 *      Load.load, Layout.layout_data
 *
 * todo?:
 *  - -v is quite useful to debug "redefinition" linking errors
 *    (see pb I had when linking bcm/ kernel)
 *  - when get undefined symbol, print function you are currently in!
 *    very useful to diagnose issue to give context and where to look for
 *  - arith LCON less: NCON
 *  - half word and byte load/store basic version
 *  - endianess and datagen
 *  - advanced instructions: floats, MULL, coprocessor, psr, etc
 *  - library ranlib/symdef indexing
 *  - profiling -p
 *  - symbol table
 *  - program counter line table
 *  - nice error reporting for signature conflict, conflicting objects
 *
 * later:
 *  - look at the 5l Go sources in the Golang source, maybe ideas to steal?
 *)

(*****************************************************************************)
(* Types, constants, and globals *)
(*****************************************************************************)
⟨type CLI.caps 27b⟩

⟨constant CLI.init_text 29a⟩
⟨constant CLI.init_round 29b⟩
⟨constant CLI.init_data 29c⟩

⟨constant CLI.init_entry 29d⟩

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
⟨function CLI.config_of_header_type 29e⟩

(*****************************************************************************)
(* Main algorithm *)
(*****************************************************************************)
⟨function CLI.link5 31a⟩

(* similar to link5 *)
let linkv (caps : < Cap.open_in; ..> ) (config : Exec_file.linker_config) (files : Fpath.t list) (chan : Chan.o
  let arch : Ast_asmv.instr Arch_linker.t = {
    Arch_linker.branch_opd_of_instr = Ast_asmv.branch_opd_of_instr;
    Arch_linker.visit_globals_instr = Ast_asmv.visit_globals_instr;
  }
  in
  let (code, data, symbols) = Load.load caps files arch in
  T.lookup (config.entry_point, T.Public) None symbols |> ignore;
  let graph = Resolve.build_graph arch.branch_opd_of_instr symbols code in
  let graph = Rewritev.rewrite graph in
  let symbols2, (data_size, bss_size) =
```

```
    Layout.layout_data symbols data in
  Layout.xdefine symbols2 symbols ("setR30" , T.Public) (T.SData2 (0, T.Data));
  Check.check symbols;
  let symbols2, graph, text_size =
    Layoutv.layout_text symbols2 config.init_text graph in
  let sizes : Exec_file.sections_size =
    Exec_file.{ text_size; data_size; bss_size }
  in
  let init_data =
    match config.init_data with
    | None -> Int_.rnd (text_size + config.init_text) config.init_round
    | Some x -> x
  in
  let config = { config with Exec_file.init_data = Some init_data } in
  Logs.info (fun m -> m "final config is %s"
        (Exec_file.show_linker_config config));
  let instrs = Codegenv.gen symbols2 config graph in
  let datas  = Datagen.gen symbols2 init_data sizes data in
  Execgen.gen config sizes instrs datas symbols2 chan
```

⟨*function* CLI.link 30b⟩

```
(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```
⟨*function* CLI.main 27d⟩

E.6 Codegen5.mli

⟨*type* Codegen5.pool (Codegen5.mli) 76a⟩≡ (76b)

⟨Codegen5.mli 76b⟩≡

⟨*type* Codegen5.pool (Codegen5.mli) 76a⟩

⟨*signature* Codegen5.size_of_instruction 45a⟩

⟨*signature* Codegen5.gen 46c⟩

```
(* internals *)
val rules: Codegen.env -> Types.addr option -> Types5.node -> Codegen.action
```

E.7 Codegen5.ml

⟨*type* Codegen5.pool 76c⟩≡ (79d)

⟨*function* Codegen5.int_of_bits 76d⟩≡ (79d)

```
let int_of_bits (n : node) (x : Bits.int32) : int =
  try
    Bits.int_of_bits32 x
  with Failure s -> error n s
```

⟨*function* Codegen5.offset_to_R12 76e⟩≡ (79d)

```
let offset_to_R12 x =
  (* less: x - BIG at some point if want some optimisation *)
  x
```

⟨*function* Codegen5.base_and_offset_of_indirect 77a⟩≡ (79d)

```
let base_and_offset_of_indirect node symbols2 autosize x =
  match x with
  | Indirect (r, off) -> r, off
  | Entity (Param (_s, off)) ->
      (* remember that the +4 below is because we access the frame of the
       * caller which for sure is not a leaf. Note that autosize
       * here had possibly a +4 done if the current function
       * was a leaf, but still we need another +4 because what matters
       * now is the adjustment in the frame of the caller!
       *)
      rSP, autosize + 4 + off
  | Entity (Local (_s, off)) ->
      rSP, autosize + off
  | Entity (Global (global, off)) ->
      let v = Hashtbl.find symbols2 (T.symbol_of_global global) in
      (match v with
        | T.SData2 (offset, _kind) ->
          rSB, offset_to_R12 (offset + off)
      (* stricter: allowed in 5l but I think with wrong codegen *)
      | T.SText2 _ ->
          error node (spf "use of procedure %s in indirect with offset"
                      (A.s_of_global global))
      )
  | Imsr _ | Ximm _ -> raise (Impossible "should be called only for indirects")
```

⟨*function* Codegen5.immrot 77b⟩≡ (79d)

```
let immrot x =
  if x >= 0 && x <= 0xff
  then Some (0, x)
  else raise Todo
```

⟨*function* Codegen5.immoffset 77c⟩≡ (79d)

```
let immoffset x =
  (x >= 0 && x <= 0xfff) || (x < 0 && x >= -0xfff)
```

⟨*function* Codegen5.gcond 77d⟩≡ (79d)

```
(* gxxx below means gen_binary_code of xxx *)

let gcond cond =
  match cond with
  | EQ           -> (0x0, 28)
  | NE           -> (0x1, 28)
  | GE (U) -> (0x2, 28)
  | LT (U) -> (0x3, 28)
  | MI           -> (0x4, 28)
  | PL           -> (0x5, 28)
  | VS           -> (0x6, 28)
  | VC           -> (0x7, 28)
  | GT (U) -> (0x8, 28)
  | LE (U) -> (0x9, 28)
  | GE (S)   -> (0xa, 28)
  | LT (S)   -> (0xb, 28)
  | GT (S)   -> (0xc, 28)
  | LE (S)   -> (0xd, 28)
  | AL           -> (0xe, 28)
  | NV           -> (0xf, 28)
```

⟨*function* Codegen5.gop_arith 78a⟩≡ (79d)

```
let gop_arith op =
  match op with
  | AND -> (0x0, 21)
  | EOR -> (0x1, 21)
  | SUB -> (0x2, 21)
  | RSB -> (0x3, 21)
  | ADD -> (0x4, 21)
  | ADC -> (0x5, 21)
  | SBC -> (0x6, 21)
  | RSC -> (0x7, 21)
  (* TST 0x8, TEQ 0x9, CMP 0xa, CMN 0xb via gop_cmp below *)
  | ORR -> (0xc, 21)
  | MOV -> (0xd, 21) (* no reading syntax in 5a, but can be generated by 5l *)
  | BIC -> (0xe, 21)
  | MVN -> (0xf, 21)

  | MUL | DIV | MOD -> raise (Impossible "should match those cases separately")
  | SLL | SRL | SRA -> raise (Impossible "should match those cases separately")
```

⟨*function* Codegen5.gsetbit 78b⟩≡ (79d)

```
let gsetbit opt =
  match opt with
  | None -> []
  | Some Set_condition -> [(1, 20)]
```

⟨*function* Codegen5.gop_shift 78c⟩≡ (79d)

```
let gop_shift op =
  match op with
  | SLL -> (0, 5)
  | SRL -> (1, 5)
  | SRA -> (2, 5)
  | _ -> raise (Impossible "should match those cases separately")
```

⟨*function* Codegen5.gop_cmp 78d⟩≡ (79d)

```
let gop_cmp op =
  match op with
  (* Set_condition set by default for comparison opcodes *)
  | TST -> [(0x8, 21); (1, 20)]
  | TEQ -> [(0x9, 21); (1, 20)]
  | CMP -> [(0xa, 21); (1, 20)]
  | CMN -> [(0xb, 21); (1, 20)]
```

⟨*function* Codegen5.gop_bitshift_register 78e⟩≡ (79d)

```
let gop_bitshift_register op =
  match op with
  | Sh_logic_left   -> (0x0, 5)
  | Sh_logic_right  -> (0x1, 5)
  | Sh_arith_right  -> (0x2, 5)
  | Sh_rotate_right -> (0x3, 5)
```

⟨*function* Codegen5.gop_rcon 78f⟩≡ (79d)

```
let gop_rcon x =
  match x with
  | Left (R r) -> [(r,8); (1, 4)]
  | Right i    -> [(i, 7); (0, 4)]
```

⟨*function* Codegen5.gshift 78g⟩≡ (79d)

```
let gshift (R rf) op2 rcon =
  gop_rcon rcon @ [gop_bitshift_register op2; (rf, 0)]
```

⟨*function* Codegen5.gbranch_static 79a⟩≡ (79d)

```
let gbranch_static (nsrc : T5.node) cond is_bl =
  match nsrc.branch with
  | None -> raise (Impossible "resolving should have set the branch field")
  | Some ndst ->
      let dst_pc = ndst.real_pc in
      let v = (dst_pc - nsrc.real_pc) - 8 in
      if v mod 4 <> 0
      then raise (Impossible "layout text wrong, not word aligned node");
      let v = (v asr 2) land 0xffffff in
      (* less: stricter: warn if too big, but should never happens *)
      [gcond cond; (0x5, 25);
       (if is_bl then (0x1, 24) else (0x0, 24));
       (v, 0)
       ]
```

⟨*function* Codegen5.gmem 79b⟩≡ (79d)

```
let gmem cond op move_size opt offset_or_rm (R rbase) (R rt) =
  [gcond cond; (0x1, 26) ] @
  (match opt with
  | None ->                  [(1, 24)] (* pre offset *)
  | Some PostOffsetWrite ->  [(0, 24)]
  | Some WriteAddressBase -> [(1, 24); (1, 21)]
  ) @
  [(match move_size with
   | Word -> (0, 22)
   | Byte _ -> (1, 22)
   | HalfWord _ -> raise (Impossible "should use different pattern rule")
   );
   (match op with
    | LDR -> (1, 20)
    | STR -> (0, 20)
   );
   (rbase, 16); (rt, 12);
  ] @
  (match offset_or_rm with
  | Either.Left offset ->
      if offset >= 0
      then [(1, 23); (offset, 0)]
      else [(0, 23); (-offset, 0)]
  | Either.Right (R r) -> [(1, 25); (r, 0)]
  )
```

⟨*function* Codegen5.gload_from_pool 79c⟩≡ (79d)

```
let gload_from_pool (nsrc : T5.node) cond rt =
  match nsrc.branch with
  | None -> raise (Impossible "literal pool should be attached to node")
  | Some ndst ->
      (* less: could assert the dst node is a WORD *)
      let dst_pc = ndst.real_pc in
      let v = (dst_pc - nsrc.real_pc) - 8 in
      if v mod 4 <> 0
      then raise (Impossible "layout text wrong, not word aligned node");
      (* LDR v(R15), RT (usually R11) *)
      gmem cond LDR Word None (Left v) rPC rt
```

⟨Codegen5.ml 79d⟩≡

```
(* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)
open Common
open Either
```

```
open Ast_asm
open Ast_asm5

module T = Types
module T5 = Types5
open Types
open Types5
open Codegen

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* ARM code generation.
 *
 * ocaml: No need for optab/oplook/ocmp/cmp as in 5l. Just use pattern matching!
 *)

(*****************************************************************************)
(* Types and constants *)
(*****************************************************************************)
```

⟨*type* Codegen5.pool 76c⟩

⟨*type* Codegen5.action 46a⟩

⟨*type* Codegen5.mem_opcode 46b⟩

```
(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
```

⟨*function* Codegen5.error 47a⟩

⟨*function* Codegen5.int_of_bits 76d⟩

⟨*function* Codegen5.offset_to_R12 76e⟩

⟨*function* Codegen5.base_and_offset_of_indirect 77a⟩

```
(*****************************************************************************)
(* Operand classes *)
(*****************************************************************************)
```

⟨*function* Codegen5.immrot 77b⟩

⟨*function* Codegen5.immoffset 77c⟩

```
(*****************************************************************************)
(* Code generation helpers *)
(*****************************************************************************)
```

⟨*function* Codegen5.gcond 77d⟩

⟨*function* Codegen5.gop_arith 78a⟩

⟨*function* Codegen5.gsetbit 78b⟩

⟨*function* Codegen5.gop_shift 78c⟩

⟨*function* Codegen5.gop_cmp 78d⟩

⟨*function* Codegen5.gop_bitshift_register 78e⟩

⟨*function* Codegen5.gop_rcon 78f⟩

```
(*****************************************************************************)
(* More complex code generation helpers *)
(*****************************************************************************)
```

⟨*function* Codegen5.gshift 78g⟩

⟨*function* Codegen5.gbranch_static 79a⟩

⟨*function* Codegen5.gmem 79b⟩

⟨*function* Codegen5.gload_from_pool 79c⟩

```
(*****************************************************************************)
(* The rules! *)
(*****************************************************************************)
```
⟨*function* Codegen5.rules 47b⟩

```
(*****************************************************************************)
(* Entry points *)
(*****************************************************************************)
(* TODO: could reuse this code and only things changing are the rules to pass?*)
```
⟨*function* Codegen5.size_of_instruction 53⟩

```
(* TODO: could reuse this code and only things changing are the rules to pass?*)
```
⟨*function* Codegen5.gen 46d⟩

E.8 Datagen.mli

⟨Datagen.mli 81a⟩≡

⟨*signature* Datagen.gen 35b⟩

E.9 Datagen.ml

⟨Datagen.ml 81b⟩≡
```
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common

module A = Ast_asm
module T = Types

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```
⟨*function* Datagen.gen 35c⟩

E.10 Execgen.mli

⟨Execgen.mli 82a⟩≡

⟨*signature* Execgen.gen 33⟩

E.11 Execgen.ml

⟨Execgen.ml 82b⟩≡
```
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common

module T = Types

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```

⟨*function* Execgen.gen 34a⟩

E.12 Layout5.mli

⟨Layout5.mli 82c⟩≡

⟨*signature* Layout5.layout_data 43a⟩

⟨*signature* Layout5.layout_text 44a⟩

E.13 Layout5.ml

⟨Layout5.ml 82d⟩≡
```
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common

module T = Types
module T5 = Types5
module A = Ast_asm
```

⟨*function* Layout5.xdefine 45b⟩

```
(*****************************************************************************)
(* Entry points *)
(*****************************************************************************)
```
⟨*function* Layout5.layout_data 43b⟩

```
(* TODO: seems reusable if pass Codegen5.size_of_instruction? move
 * to Latout.ml with layout_date?
 *)
```
⟨*function* Layout5.layout_text 44b⟩

E.14 Load.mli

⟨Load.mli 83a⟩≡

⟨*signature* Load.load 31b⟩

E.15 Load.ml

⟨*function* Load.process_global 83b⟩≡ (83c)
```
(* "Names", modifies global, modifies h *)
let process_global (global : A.global) (h : T.symbol_table) (idfile : int) : unit =
  (match global.priv with
  | Some _ -> global.priv <- Some idfile
  | None -> ()
  );
  (* populate symbol table with SXref if new entity *)
  T.lookup_global global h |> ignore
```

⟨Load.ml 83c⟩≡
```
(* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)
open Common
open Fpath_.Operators
module A = Ast_asm
module T = Types

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

⟨function Load.process_global 83b⟩

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)

⟨function Load.load 31c⟩
```

E.16 Main.ml

⟨Main.ml 83d⟩≡
```
(* Copyright 2025 Yoann Padioleau, see copyright.txt *)
(* open Xix_linker *)

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
⟨toplevel Main._1 27a⟩
```

E.17 Resolve.mli

⟨Resolve.mli 83e⟩≡

⟨*signature* Resolve.build_graph 40a⟩

E.18 Resolve.ml

⟨Resolve.ml 84a⟩≡
```
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common

open Types
module A = Ast_asm
module T = Types

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```
⟨*function* Resolve.build_graph 40b⟩

E.19 Rewrite5.mli

⟨Rewrite5.mli 84b⟩≡

⟨*signature* Rewrite5.rewrite 41a⟩

E.20 Rewrite5.ml

⟨*function* Rewrite5.find_first_no_nop_node 84c⟩≡ (84d)

⟨Rewrite5.ml 84d⟩≡
```
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common

module A = Ast_asm
module A5 = Ast_asm5
open Types
module T = Types
module T5 = Types5

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
```
⟨*function* Rewrite5.find_first_no_nop_node 84c⟩
```
(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```
⟨*function* Rewrite5.rewrite 41b⟩

E.21 Types.ml

⟨*function* Types.lookup 84e⟩≡ (85e)
```
(* create new entry with SXRef if not found *)
let lookup (k : symbol) (sigopt : signature option) (h : symbol_table) : value =
  let v =
    try
      Hashtbl.find h k
    with Not_found ->
      let v = { section = SXref; sig_ = sigopt } in
```

```
        Hashtbl.add h k v;
        v
    in
    (match sigopt, v.sig_ with
    | None, None -> ()
    | Some i1, Some i2 ->
        (* todo: report also offending object files *)
        if i1 <> i2
        then failwith (spf "incompatible type signatures %d and %d" i1 i2)
    (* less: could report error when one define sig and not other *)
    | _ -> ()
    );
    v
```

⟨*function* Types.lookup_global 85a⟩≡ (85e)

```
  let lookup_global (x : A.global) (h : symbol_table) : value =
    let symbol = symbol_of_global x in
    lookup symbol x.signature h
```

⟨*function* Types.s_of_loc 85b⟩≡ (85e)

```
  (* less: would need Hist mapping for this file to convert to original source *)
  let s_of_loc (file, line) =
    spf "%s:%d" !!file line
```

⟨*function* Types.iter 85c⟩≡ (85e)

```
  let rec iter f n =
    f n;
    n.next |> Option.iter (fun n -> iter f n)
```

⟨*function* Types.iter_with_env 85d⟩≡ (85e)

```
  let rec iter_with_env f env n =
    let env = f env n in
    n.next |> Option.iter (fun n -> iter_with_env f env n)
```

⟨Types.ml 85e⟩≡

```
  (* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)
  open Common
  open Fpath_.Operators

  module A = Ast_asm

  (*****************************************************************************)
  (* Types *)
  (*****************************************************************************)

  (* ---------------------------------------- *)
  (* basic types *)
  (* ---------------------------------------- *)

  ⟨type Types.loc 23a⟩
  [@@deriving show]

  ⟨type Types.byte 23b⟩
  [@@deriving show]
  ⟨type Types.word 23c⟩
  [@@deriving show]

  ⟨type Types.addr 23d⟩
  [@@deriving show]
  ⟨type Types.offset 23e⟩
```

```
[@@deriving show]

⟨type Types.symbol 23f⟩
⟨type Types.scope 23g⟩
[@@deriving show]

(* ---------------------------------------- *)
(* The virtual pc world *)
(* ---------------------------------------- *)

⟨type Types.virt_pc 24a⟩
[@@deriving show]

⟨type Types.section 24b⟩
[@@deriving show]

⟨type Types.signature 24d⟩
[@@deriving show]

⟨type Types.value 24c⟩
[@@deriving show]

⟨type Types.symbol_table 24e⟩
(*[@@deriving show]*)

(* ---------------------------------------- *)
(* The real pc world *)
(* ---------------------------------------- *)

⟨type Types.real_pc 24f⟩
[@@deriving show]

⟨type Types.section2 24g⟩
⟨type Types.data_kind 24h⟩
[@@deriving show]

⟨type Types.value2 24i⟩
[@@deriving show]

⟨type Types.symbol_table2 24j⟩
(*[@@deriving show]*)

(* ---------------------------------------- *)
(* Code vs Data *)
(* ---------------------------------------- *)


(* Split Asm instructions in code vs data.
 *
 * For 'code' below we want to do some naming. We could copy many of
 * ast_asm.ml and replace 'global' with the fully resolved 'symbol'.
 * But it would be a big copy paste. Instead, we opted for a mutable field
 * in ast_asm.ml set by the linker (see Ast_asm.entity.priv).
 *)
⟨type Types.code 24k⟩
⟨type Types.code_bis 25a⟩
[@@deriving show {with_path = false}]

⟨type Types.data 25b⟩
[@@deriving show]
```

```
  (* graph via pointers, like in original 5l *)
  ⟨type Types.node 25c⟩
  [@@deriving show]

  ⟨type Types.code_graph 25d⟩
  [@@deriving show]

  (*****************************************************************************)
  (* Helpers *)
  (*****************************************************************************)

  ⟨function Types.lookup 84e⟩

  ⟨function Types.s_of_symbol 23h⟩

  ⟨function Types.symbol_of_global 23i⟩

  ⟨function Types.lookup_global 85a⟩


  ⟨function Types.s_of_loc 85b⟩

  ⟨function Types.iter 85c⟩

  ⟨function Types.iter_with_env 85d⟩
```

E.22 Types5.ml

```
⟨function Types5.show_instr 87a⟩≡                                    (87b)
  (* for ocaml-light, to work without deriving *)
  let show_instr _ = "NO DERIVING"
  [@@warning "-32"]
```

```
⟨Types5.ml 87b⟩≡
  (* Copyright 2016, 2025 Yoann Padioleau, see copyright.txt *)
  open Common

  (*****************************************************************************)
  (* Types *)
  (*****************************************************************************)

  ⟨function Types5.show_instr 87a⟩

  ⟨type Types5.instr 25e⟩

  ⟨type Types5.node 25f⟩

  ⟨type Types5.code_graph 25g⟩
```

E.23 executables/A_out.mli

```
⟨executables/A_out.mli 87c⟩≡

  ⟨signature A_out.header_size 25h⟩
```

⟨*signature* A_out.write_header 34b⟩

E.24 executables/A_out.ml

⟨executables/A_out.ml 88a⟩≡
```
(* Copyright 2016 Yoann Padioleau, see copyright.txt *)
open Common

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```
⟨*constant* A_out.header_size 25i⟩
```
(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```
⟨*function* A_out.write_header 35a⟩

E.25 executables/Elf.mli

⟨executables/Elf.mli 88b⟩≡

⟨*signature* Elf.header_size 58a⟩

⟨*signature* Elf.write_headers 58b⟩

E.26 executables/Elf.ml

⟨executables/Elf.ml 88c⟩≡
```
(* Copyright 2025 Yoann Padioleau, see copyright.txt *)
open Common

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* ELF executable format types and IO helpers.
 *
 * spec: ???
*)

(*****************************************************************************)
(* Types and constants *)
(*****************************************************************************)
```
⟨*constant* Elf.exec_header_32_size 59a⟩
⟨*constant* Elf.program_header_32_size 59b⟩
⟨*constant* Elf.section_header_32_size 59c⟩

⟨*constant* Elf.nb_program_headers 59d⟩

⟨*constant* Elf.header_size 59e⟩

```
(* ---------------------------------------------------------------------- *)
```

```
(* Exec header *)
(* ------------------------------------------------------------------------ *)
```

⟨*type* `Elf.ident_class` 59f⟩

```
(* alt: ident_data *)
```
⟨*type* `Elf.byte_order` 59g⟩

⟨*type* `Elf.ident_version` 59h⟩

⟨*type* `Elf.elf_type` 59i⟩

⟨*type* `Elf.machine` 59j⟩

```
(* ------------------------------------------------------------------------ *)
(* Program header *)
(* ------------------------------------------------------------------------ *)
```

⟨*type* `Elf.program_header_type` 60a⟩

⟨*type* `Elf.program_header_protection` 60b⟩

```
(* ------------------------------------------------------------------------ *)
(* Section header *)
(* ------------------------------------------------------------------------ *)

(****************************************************************************)
(* Conversions *)
(****************************************************************************)
```
⟨*function* `Elf.byte_of_class` 60c⟩

⟨*function* `Elf.byte_of_byte_order` 60d⟩

⟨*function* `Elf.int_of_version` 60e⟩

⟨*function* `Elf.int_of_elf_type` 60f⟩

⟨*function* `Elf.int_of_machine` 61a⟩

⟨*function* `Elf.int_of_program_header_type` 61b⟩

⟨*function* `Elf.int_of_prot` 61c⟩

⟨*function* `Elf.int_of_prots` 61d⟩

```
(****************************************************************************)
(* IO *)
(****************************************************************************)
```
⟨*function* `Elf.write_ident` 61e⟩

⟨*function* `Elf.program_header_32` 62a⟩

```
(****************************************************************************)
(* Entry point *)
(****************************************************************************)
```
⟨*function* `Elf.write_headers` 62b⟩

E.27 executables/Exec_file.ml

⟨*function* Exec_file.show_linker_config 90a⟩≡ (90b)
```
(* for ocaml-light to work without deriving *)
let show_linker_config _ = "NO DERIVING"
[@@warning "-32"]
```

⟨executables/Exec_file.ml 90b⟩≡

```
⟨type Exec_file.addr 26c⟩
[@@deriving show]

⟨type Exec_file.header_type 26b⟩
[@@deriving show]

⟨type Exec_file.sections_size 26d⟩
[@@deriving show]

⟨function Exec_file.show_linker_config 90a⟩

⟨type Exec_file.linker_config 26a⟩
[@@deriving show]
```

E.28 libraries/Library_file.mli

⟨libraries/Library_file.mli 90c⟩≡
```
⟨type Library_file.t 38a⟩

⟨signature Library_file.load 38b⟩

⟨signature Library_file.save 38c⟩

⟨signature Library_file.is_lib_filename 38d⟩
```

E.29 libraries/Library_file.ml

⟨libraries/Library_file.ml 90d⟩≡
```
(* Copyright 2025 Yoann Padioleau, see copyright.txt *)
open Common
open Regexp_.Operators
open Fpath_.Operators

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Simple API to load/save archive library files (e.g., libc.a) *)

(*****************************************************************************)
(* Types and constants *)
(*****************************************************************************)

(* An archive (.a) is really essentially just a list of objects, which in Plan 9
 * are just a list of serialized assembly ASTs
 *
 * TODO: do SYMDEF/ranlib indexing so can avoid objects that are not
 * needed by the linked program like in 5l/vl/...
```

```
 *)
```
⟨*type* `Library_file.t` 38a⟩

```
(*****************************************************************************)
(* API *)
(*****************************************************************************)
```

⟨*function* `Library_file.save` 38e⟩

⟨*function* `Library_file.load` 38f⟩

⟨*function* `Library_file.is_lib_filename` 38g⟩

E.30 tools/ar.ml

⟨`tools/ar.ml` 91⟩≡

```
(* Copyright 2025 Yoann Padioleau, see copyright.txt *)
open Common
open Fpath_.Operators

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* An OCaml port of ar, the Plan 9 (object) archiver.
 *
 * Main limitations compared to ar:
 *  - no complex CLI flags; just 'oar objfiles [-o libfile]'
 *    no ar vu, ar rcs, ... just archive!
 *
 * todo:
 *  - index the archive a la SYMDEF/ranlib to reduce size of
 *    binaries in the linker for unneeded object files
 *
 * later:
 *)

(*****************************************************************************)
(* Types, constants, and globals *)
(*****************************************************************************)
```
⟨*type* `Ar.caps` 70c⟩

⟨*constant* `Ar.usage` 70d⟩

```
(*****************************************************************************)
(* Main algorithm *)
(*****************************************************************************)
```
⟨*function* `Ar.archive` 71a⟩

```
(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```
⟨*function* `Ar.main` 71b⟩

```
(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```
⟨*constant* `Ar._` 72⟩

E.31 tools/nm.ml

⟨tools/nm.ml 92⟩≡

```
(* Copyright 2025 Yoann Padioleau, see copyright.txt *)
open Common
open Ast_asm
module A = Ast_asm

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* An OCaml port of nm, the Plan 9 object/executable/library symbol inspector.
 * 'nm' probably stands for "names" or "name map".
 *
 * Main limitations compared to nm:
 *  -
 *
 * todo:
 *  - handle executable too
 *
 * later:
 *)

(*****************************************************************************)
(* Types, constants, and globals *)
(*****************************************************************************)
⟨type Nm.caps 69a⟩

⟨constant Nm.usage 69b⟩

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

⟨function Nm.visit_obj 69c⟩

(*****************************************************************************)
(* Main algorithm *)
(*****************************************************************************)

⟨function Nm.nm 69d⟩

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)

⟨function Nm.main 70a⟩

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
⟨constant Nm._ 70b⟩
```

Glossary

LDR = LoaD Register
STR = STore Register
ARM = Acorn RISC Machine
RISC = Reduced Instruction Set Computer
CISC = Complex Instruction Set Computer
PC = Program Counter
SB = Static Base register
SP = Stack Pointer
FP = Frame Pointer

Indexes

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

Bibliography

[BO10] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2010. cited page(s) 10

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 10

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 10

[Lev99] John R. Levine. *Linkers and Loaders*. Morgan-Kauffman, 1999. Available at http://www.iecc.com/linker/. cited page(s) 10, 14

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 10

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 11, 14

[Pad15a] Yoann Padioleau. *Principia Softwarica: The ARM Assembler 5a*. 2015. cited page(s) 10, 13, 15, 17, 19

[Pad15b] Yoann Padioleau. *Principia Softwarica: The ARM Emulator 5i*. 2015. cited page(s) 10

[Pad16] Yoann Padioleau. *Principia Softwarica: The Plan 9 Debuggers and Tracers*. 2016. cited page(s) 20

[Sea01] David Seal. *ARM, Architecture Reference Manual*. Addison-Wesley, 2001. cited page(s) 8

[Tay08] Ian Lance Taylor. A new elf linker. In *Proceedings of the GCC Developers' Summit*, 2008. Available at http://ols.fedoraproject.org/GCC/Reprints-2008/taylor-reprint.pdf. cited page(s) 8