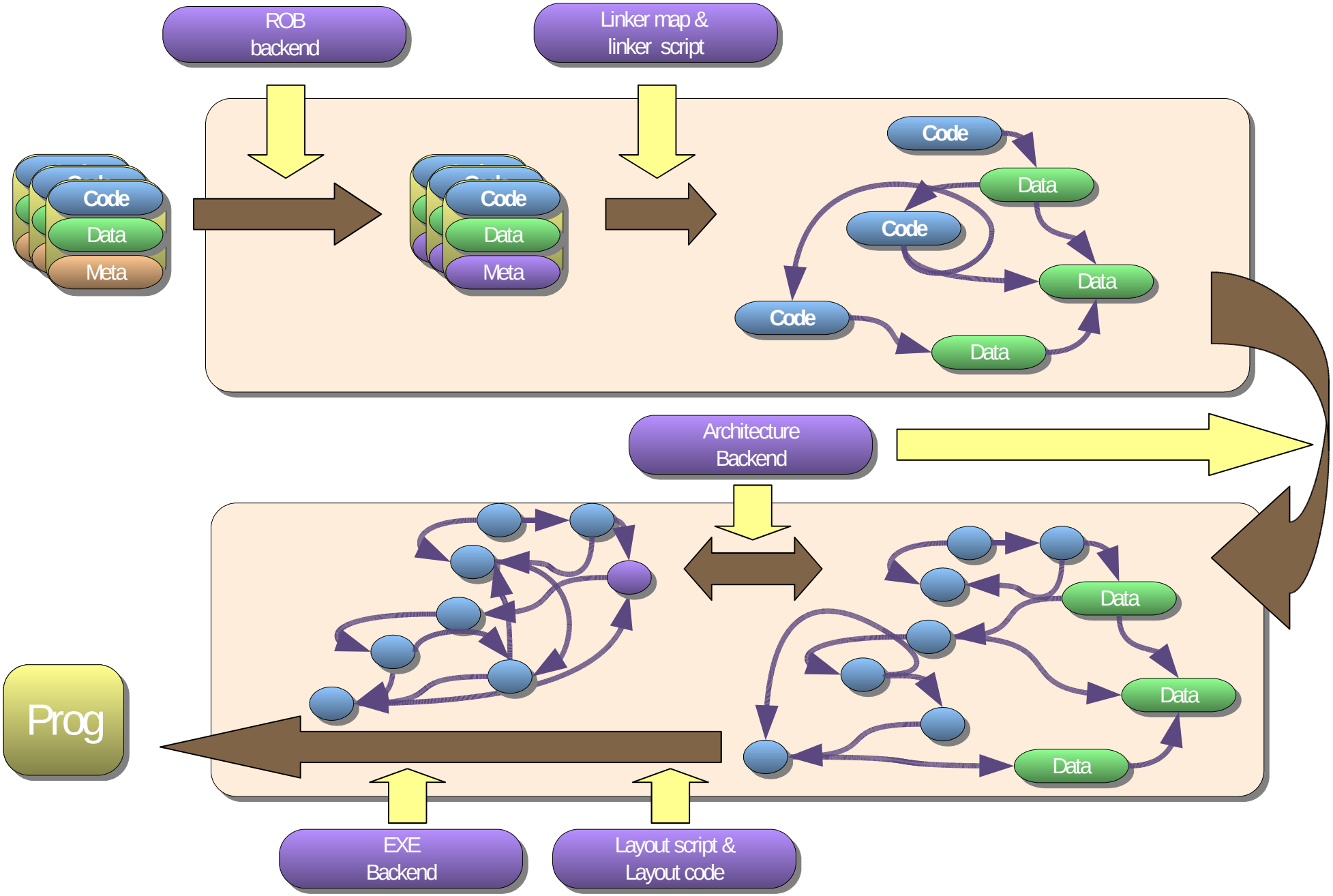
Creating a new Diablo architecture backend


ROB
backend
Linker map &
linker script
Code
Data
Meta
Code
Data
Meta
Code
Data
Code
Data
Code
Data
Architecture
Backend
Data
Data
Data
Prog
EXE
Backend
Layout script &
Layout code

ROB Backend

- interpret architecture-specific information in the relocatable object files
  - relocation types
  - architecture-specific section flags
  - ...
- convert all this to the architecture-independent Diablo format

EXE Backend

- Converts internal Diablo representation back to architecture-specific executable format
- In practice: the same as ROB backend

Map file parser

- Diablo uses a *linker map* file to relink the program in the exact same way as the native linker.
- Different linkers == different map file formats
- Currently, parsers for
  - GNU **`ld`**
  - Alpha **`ld`**
  - ARM ADS/RVCT **`armlink`**

Linker & Layout Scripts

- Linker needs to know
  - where to add symbols (*e.g.* **`_etext`**)
  - how to place sections
- Scriptable

```
.eh_frame_hdr : { *(.eh_frame_hdr) }

. = ALIGN (0x1000) - ((0x1000 - .) & (0x1000 - 1));

PROVIDE (__preinit_array_start = .);
.preinit_array     : { *(.preinit_array) }
PROVIDE (__preinit_array_end = .);
PROVIDE (__init_array_start = .);
.init_array     : { *(.init_array) }
PROVIDE (__init_array_end = .);
```

Linker & Layout Scripts

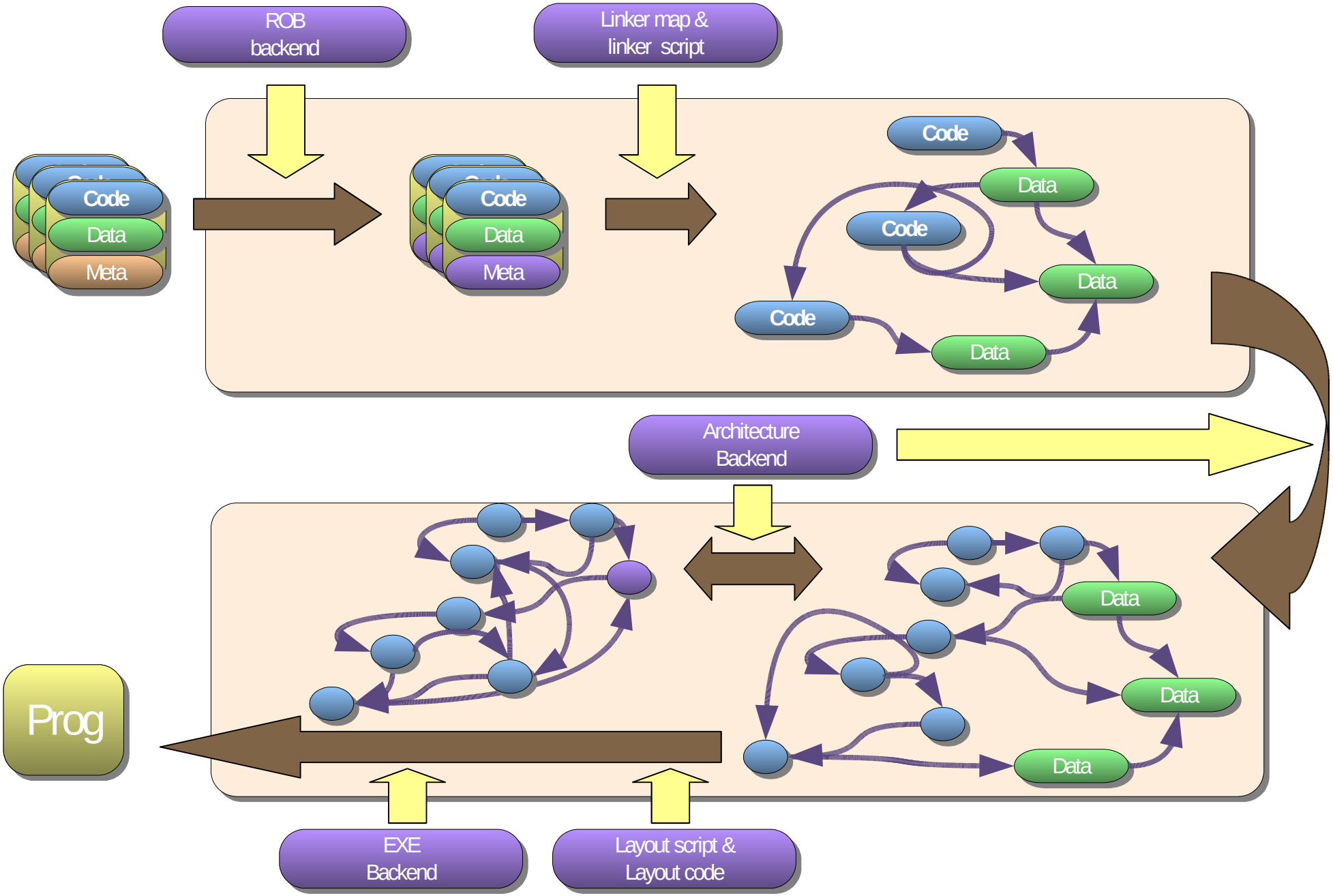
- Diablo separates the script's two tasks
  - *layout script* determines section placement in rewritten binary
    = basically a stripped-down original linker script
  - *linker script* describes symbol placement and other linker actions but with extra semantic information

Linker Script Example

```
PROVIDE (__preinit_array_start = .);
.preinit_array     : { *(.preinit_array) }
PROVIDE (__preinit_array_end = .);
PROVIDE (__init_array_start = .);
.init_array     : { *(.init_array) }
PROVIDE (__init_array_end = .);
```

```
InitVector {
   action  { VECTORIZE(".init_array") }
   trigger { SUBSECTION_EXISTS("*",".init_array") }
}

InitStart {
 action  { ADD_SYMBOL("__init_array_start") }
 trigger { UNDEFINED_SYMBOL("__init_array_start") &&
           SUBSECTION_EXISTS("Linker",
                             "VECTOR___.init_array") }
 symbol  { START_OF_SUBSECTION("Linker",
                              "VECTOR___.init_array") }
}
```


ROB
backend
Linker map &
linker script
Code
Data
Meta
Code
Data
Meta
Code
Data
Code
Data
Code
Data
Architecture
Backend
Data
Data
Data
Prog
EXE
Backend
Layout script &
Layout code

Architecture Backend

- defines the *architecture description structure*
  - instruction size
  - register description (integer, float, predicate, branch, special regs)
  - calling conventions
  - callbacks (UsedRegisters, DefinedRegisters, IsLoad, IsControlTransfer, ...)

Architecture Backend

- `t_arch_ins` structure
  - derives from generic `t_ins`
  - all arch specific instruction information: opcode, source & target registers, immediate values, etc
  - all instruction callbacks in the backend accept *this* type, not `t_ins`

Architecture Backend

- disassemble to & assemble from `t_arch_ins`
- build flowgraph
  - mostly arch independent
  - mark basic block boundaries
  - create flow graph edges
  - handle special constructs (switch statements, pc-relative constructions, ...)

Architecture Backend

- Break down flowgraph
  - optimal graph layout
  - reconstruct pc-relative constructions
- Arch independent analysis callbacks
  - instruction emulation for constant propagation
  - ...
- Arch specific analyses & optimizations

Naming conventions

- ROB backends:
  `object/object_backends/$FF/diablo$FF_$ARCH.c`
- map parsers:
  `object/map_backends/$PLATFORM/diablo$PLATFORM*.c`
- linker & layout scripts:
  `object/data/$FF-$ARCH-$PLATFORM.ld`
- architecture backends:
  `arch/$ARCH/$ARCH_*.c`
  - structures: `t_arch_xxx`
  - procedures: `ArchXxxYyy`
  - getters & setters: `ARCH_INS_XXX`