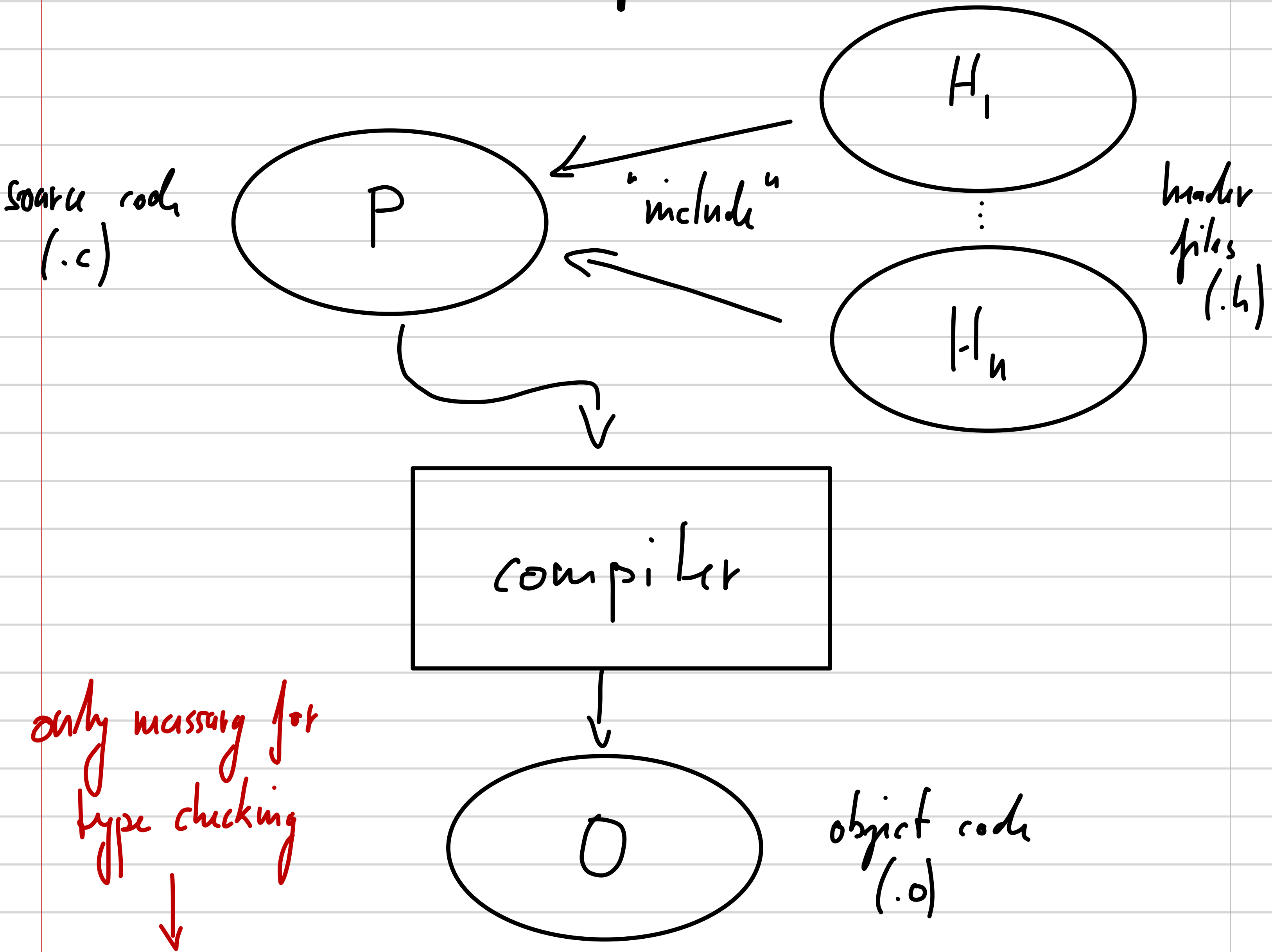
Compiler



only necessary for type checking ↓

header file: syntactically equivalent to source code except that procedure bodies are missing
~> declarations only

compiler first parses all included header files
~> constructs symbol table, no code generation!

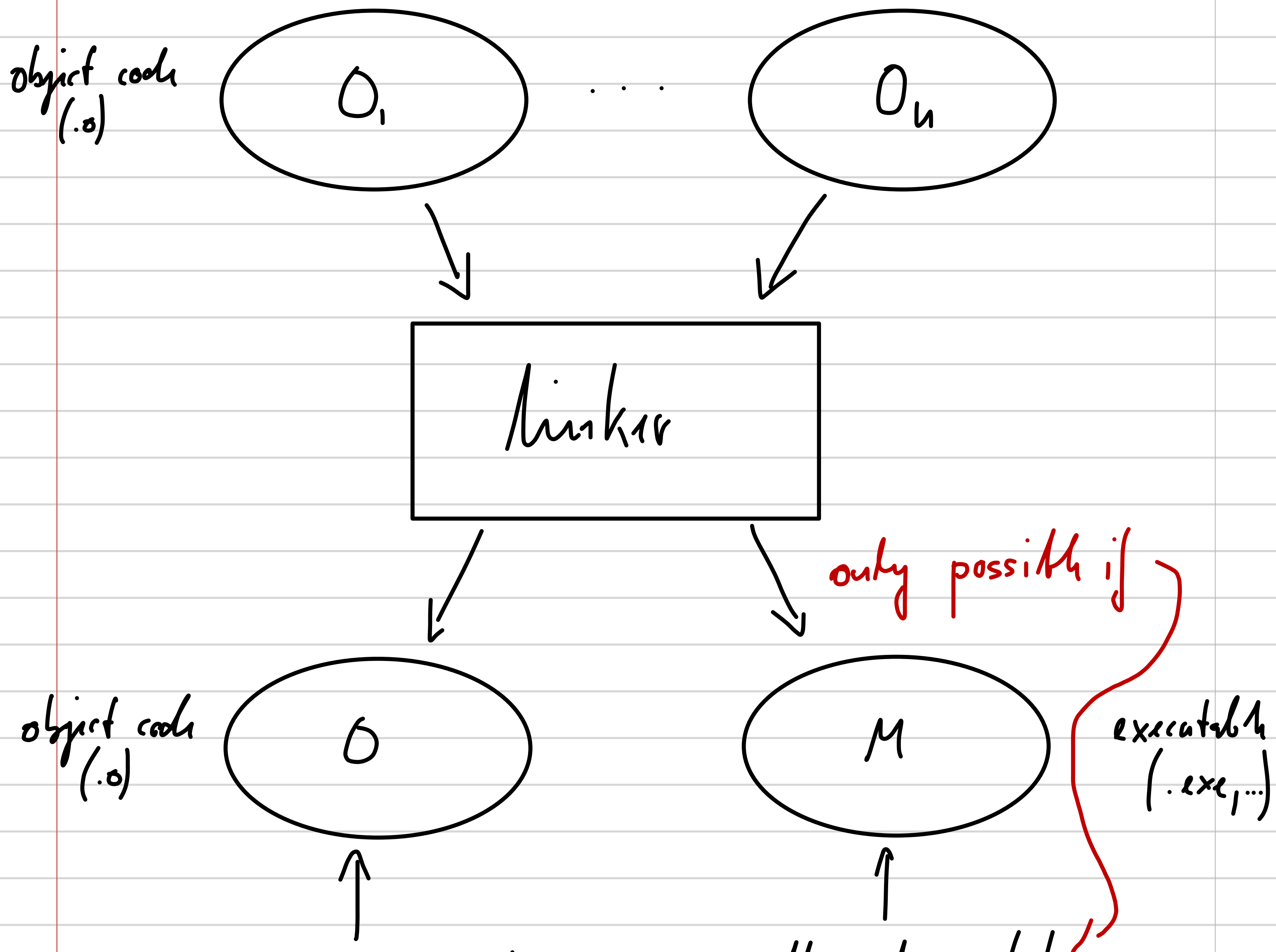
source code: uses global variables and procedures declared in imported header files

object code: contains relocatable code generated for source code, relocatable data (global vars, strings) for source code (not header files), and "symbols" (definitions, uses)

names of global variables, strings, and procedures

compiler then parses source code and generates object code

Linker



$O_1, \ldots, O_n$ merged with all used symbols that are defined linked to relocatable addresses

still need fixup

only possible if

all used symbols linked and all relocatable addresses fixed up

Java: .class files correspond to .o files but are only loaded and linked dynamically at runtime on demand } no executables!

[DLX Tools Guide, Knight, Ashenden] → DLX

Object File Format

```
objectFile = relocatableSegment { relocatableSegment }
 { symdef } { linkup } { fixup } .
```

later

```
relocatableSegment = ( ".codeseg" | ".dataseg" ) segmentName
 segmentSize { memoryData } .
segmentName = "$" identifier .
segmentSize = integer .
memoryData = word .
```

distinction simplifies relocation!

word: or any other data representation

```
.codeseg $module1_code 80 ...
.dataseg $module1_variables 20
.dataseg $module1_strings 160 ...
.codeseg $module2_code 120 ...
.dataseg $module2_variables 8
.dataseg $module2_strings 100 ...
...
```

← no memory data means all zeros

size is 80 even if there is more memory data (less means fill rest with zeros)

→ code segments will be located before data segments!

```
symdef = ".symdef" symbolName relocatableAddress .
symbolName = "#" identifier .
relocatableAddress = segmentName ( "+" | "-" ) offset .
offset = integer .
```

offset relative to: beginning ; end

```
.symdef #x $module1_variables-4
.symdef #y $module1_variables-8
```

} global variables

```
...
.symdef #module1_string1 $module1_strings-32
```

} string constants

```
...
.symdef #test $module1_code+0
.symdef #main $module1_code+42
```

} procedures

```
...
```

linkup and Fixup

```
objectFile = relocatableSegment { relocatableSegment }
  { symdef } { linkup } { fixup } .

linkup = ".linkup" relocatableAddress symbolicReference .
symbolicReference = symbolName .

.linkup $module1_code+20 #x
.linkup $module1_code+24 #y
.linkup $module1_code+110 #module1_string1
.linkup $module2_code+40 #test
...
```

access of global variables

access of string constants

procedure calls

```
.symdef #x $module1_variables-4
.symdef #y $module1_variables-8
.symdef #module1_string1 $module1_strings-32
.symdef #test $module1_code+0
```

linking

```
fixup = ".fixup" relocatableAddress relocatableAddress

.fixup $module1_code+20 $module1_variables-4
.fixup $module1_code+24 $module1_variables-8
.fixup $module1_code+110 $module1_strings-32
.fixup $module2_code+40 $module1_code+0
...
```

relocation type omitted for simplicity

Executable

```
executableFile = absoluteSegment start .

absoluteSegment =
 ".code" { memoryData } ".data" { memoryData } .
```

distinction not necessary!

```
start = ".start" offset .
```

$$address(\$i+o) = o + \sum_{1 \le j < i} size(\$j)$$

$$offset(\$i-o) = -o - \sum_{i < j \le \#\text{of data segments}} size(\$j)$$

```
.code
  memoryData<$module1_code, 80>
  memoryData<$module2_code, 120>
  ...
.data
  memoryData<$module1_variables, 20>
  memoryData<$module1_strings, 160>
  memoryData<$module2_variables, 8>
  memoryData<$module2_strings, 100>
  ...
.start 42
                    .symdef #main $module1_code+42

    .fixup $module1_code+20 $module1_variables-4
      ---> set parameter c @ address 20 in code segment to -272
           (-272 = -4-160-8-100)
    .fixup $module1_code+24 $module1_variables-8
      ---> set parameter c @ address 24 in code segment to -276
           (-276 = -8-160-8-100)
    .fixup $module1_code+110 $module1_strings-32
      ---> set parameter c @ address 110 in code segment to -140
           (-140 = -32-8-100)
    .fixup $module2_code+40 $module1_code+0
      ---> set parameter c @ address 120 in code segment to 0
           (120 = 40+80)
```

Loading and go

1. load code segment into memory starting @ address 0
2. load data segment into memory right after code segment
3. set PC to start offset — runtime PC!
4. set GP to size of code segment plus size of data segment
5. set SP to size of memory
6. set HP to GP (heap "bump" pointer) ← use HP: reg[26] in malloc implementation
7. push console parameters onto stack
8. set LINK to 0 — termination condition
9. start emulator
10. terminate on RET to 0

Bootstrapping

⇝ implement system procedures in object file:

library code

(standard approach when linker is available)

⇝ or implement as part of emulator:

virtual machine

(requires special instructions)

⇝ or have compiler generate code

(problem: code duplication)