Formal Languages and Compilers - Exercises
Lecture 2
Compilers and Interpreters

20/03/2012

Outline

O'CaML quick recap

- Run the interpreter with ocaml
- Save the file in `myfile.ml`
- the interpreter can run it with #use myfile.ml;;
- Compile a single module with `ocamlc -c myfile.ml`, creating `myfile.cmo`
- You can use the compiled file in the interpeter using #load myfile.cmo;; and **open** Myfile;;

O'CaML quick recap

- Run the interpreter with ocaml
- Save the file in `myfile.ml`
- the interpreter can run it with #use myfile.ml;;
- Compile a single module with `ocamlc -c myfile.ml`, creating `myfile.cmo`
- You can use the compiled file in the interpeter using #load myfile.cmo;; and **open** Myfile;;

O'CaML quick recap

- Run the interpreter with ocaml
- Save the file in `myfile.ml`
- the interpreter can run it with #use myfile.ml;;
- Compile a single module with `ocamlc -c myfile.ml`, creating `myfile.cmo`
- You can use the compiled file in the interpeter using #load myfile.cmo;; and **open** Myfile;;

Value binding and pattern matching

Variables

```
let (x, y) = ("hi", (1,2));;
let (a, (b,c)) = (z, (3,4));;
```

Lists

```
let h::t = [4;5;6];;
let h::t = [4]::[5;6];;
```

Declarations

```
let x = 1 and y = 2 in x*y;;
let a = 3 and b = 4 in c=a+b;;
let a = 3 and b=4 in c=a+b in c+2;;
```

Value binding and pattern matching

Variables

```
let (x, y) = ("hi", (1,2));;
let (a, (b,c)) = (z, (3,4));;
```

Lists

```
let h::t = [4;5;6];;
let h::t = [4]::[5;6];;
```

Declarations

```
let x = 1 and y = 2 in x*y;;
let a = 3 and b = 4 in c=a+b;;
let a = 3 and b=4 in c=a+b in c+2;;
```

Value binding and pattern matching

Variables

```
let (x, y) = ("hi", (1,2));;
let (a, (b,c)) = (z, (3,4));;
```

Lists

```
let h::t = [4;5;6];;
let h::t = [4]::[5;6];;
```

Declarations

```
let x = 1 and y = 2 in x*y;;
let a = 3 and b = 4 in c=a+b;;
let a = 3 and b=4 in c=a+b in c+2;;
```

Outline

Functions

Declaration of functions

```
fun x -> (x*2, x*4, x*8);;
let f x = x*2;;
let y = (f 2) in y*2;;
let f x = if x>0 then x else 0;;
```

Functions - 2

Recursive functions

```
let rec f1 = function
    0 -> 0
  | n -> n + f1(n-1);;

let rec f2 n = match n with
    0 -> 0
  | n -> n + f2 n-1;;

let rec f3 n m = match n with
    0 -> m
  | n -> f3 (n-1) m+n
```

Useful functions

`String` module

- String.length;;
- String.contains;;

`List` module

- List.rev;;
- List.hd;;
- List.tl;;
- List.append;; (same as list1@list2)

Examples

- List.hd [1;2;3];;
- List.hd (List.tl [4;5;6]);;
- [1;2;3]@[4;5];;

Useful functions

`String` module

- String.length;;
- String.contains;;

`List` module

- List.rev;;
- List.hd;;
- List.tl;;
- List.append;; (same as list1@list2)

Examples

- List.hd [1;2;3];;
- List.hd (List.tl [4;5;6]);;
- [1;2;3]@[4;5];;

Useful functions

`String` module

- String.length;;
- String.contains;;

`List` module

- List.rev;;
- List.hd;;
- List.tl;;
- List.append;; (same as list1@list2)

Examples

- List.hd [1;2;3];;
- List.hd (List.tl [4;5;6]);;
- [1;2;3]@[4;5];;

Exercise time!

- Given a list of strings `l`, define a function `filter` that builds a new list that contains elements from `l` such that every element of the list contains the character `y` and has length greater than 3.
- The order of elements should be preserved.
- For example, if

```
l = ["tool";
     "hammer";
     "Formal languages and compilers lecture";
     "ocaml"]
y = 't'
```

the result is

```
["tool";
"Formal languages and compilers lecture"]
```

Exercise time!

- Given a list of strings `l`, define a function `filter` that builds a new list that contains elements from `l` such that every element of the list contains the character `y` and has length greater than 3.
- The order of elements should be preserved.
- For example, if

```
l = ["tool";
     "hammer";
     "Formal languages and compilers lecture";
     "ocaml"]
y = 't'
```

the result is

```
["tool";
"Formal languages and compilers lecture"]
```

Exercise time!

- Given a list of strings `l`, define a function `filter` that builds a new list that contains elements from `l` such that every element of the list contains the character `y` and has length greater than 3.
- The order of elements should be preserved.
- For example, if

```
l = ["tool";
     "hammer";
     "Formal languages and compilers lecture";
     "ocaml"]
y = 't'
```

the result is

```
["tool";
"Formal languages and compilers lecture"]
```

Outline

Compilers and Interpreters

Interpreter

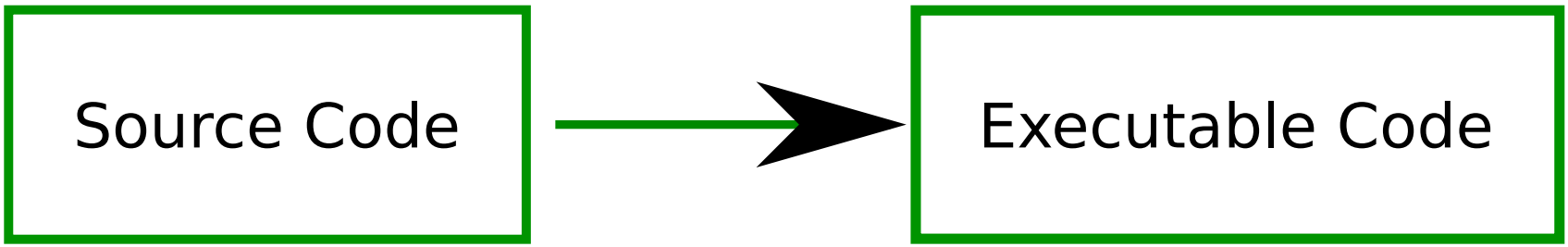
- run with `ocaml`
- exit with #quit;;

Compilers

- `ocamlc` compiles in bytecode
- `ocamlc -c <fileName>.ml` produces `<fileName>.cmo`

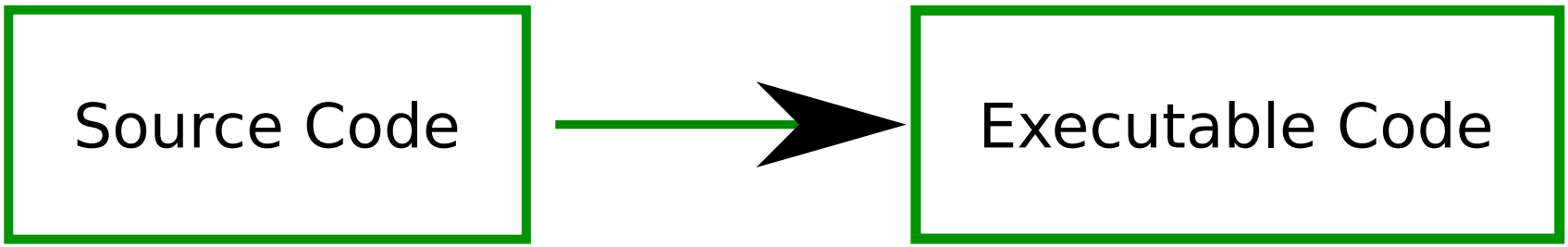
What's the difference?

Compiler (High level)



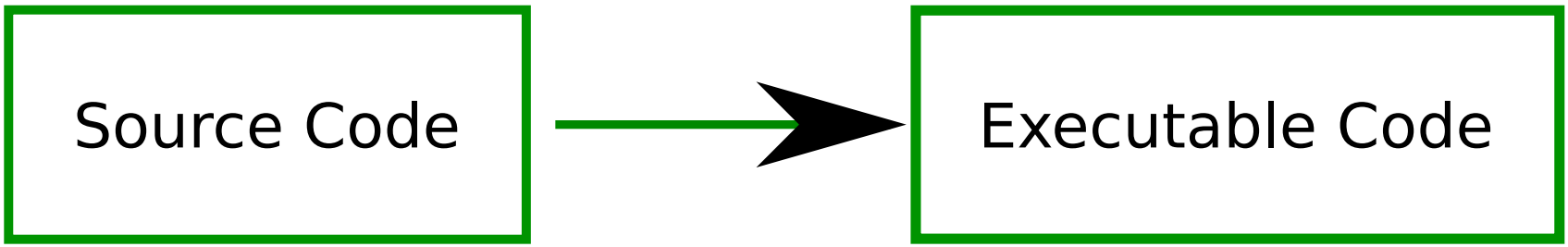
- From a high-level language to machine language
- Single translation
- If an error is found, the source code is not converted

Compiler (High level)



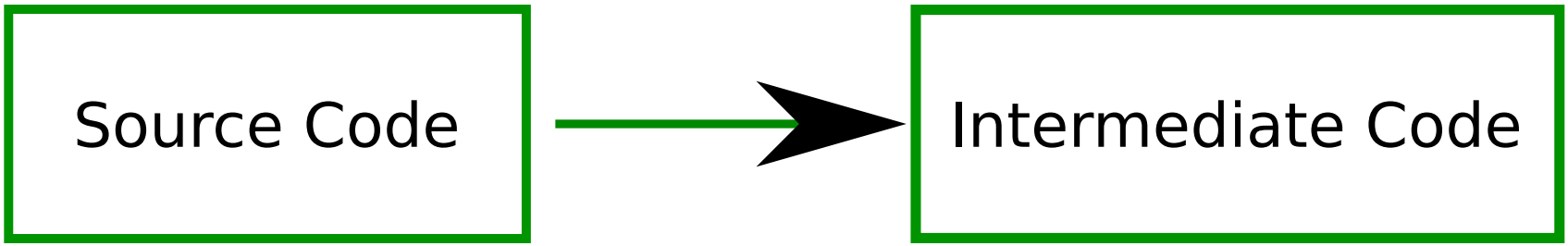
- From a high-level language to machine language
- Single translation
- If an error is found, the source code is not converted

Compiler (High level)



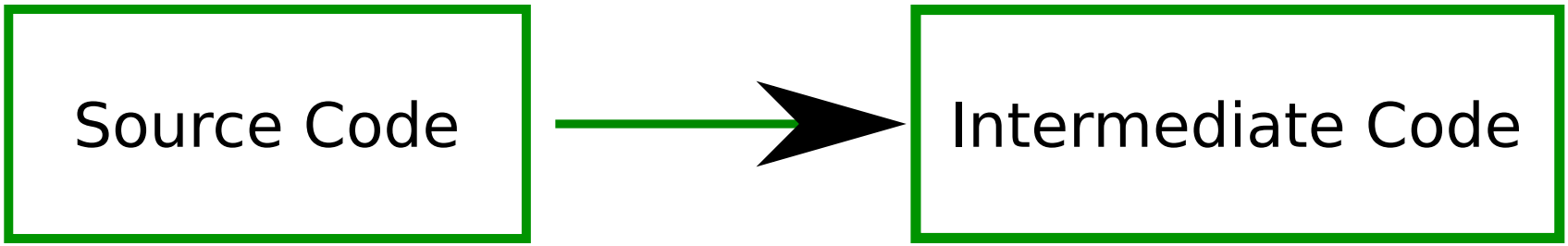
- From a high-level language to machine language
- Single translation
- If an error is found, the source code is not converted

Interpreter (High level)



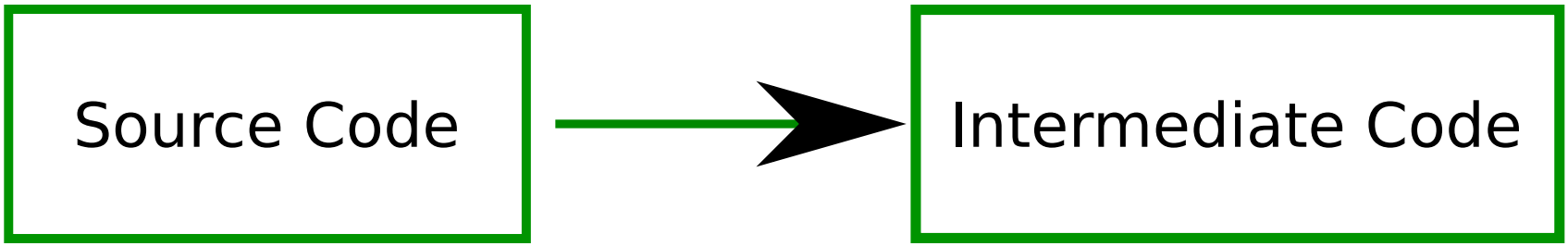
- From a high-level language to some intermediate language
- Statement by statement
- If an error is found, the interpreter stops and shows an error

Interpreter (High level)



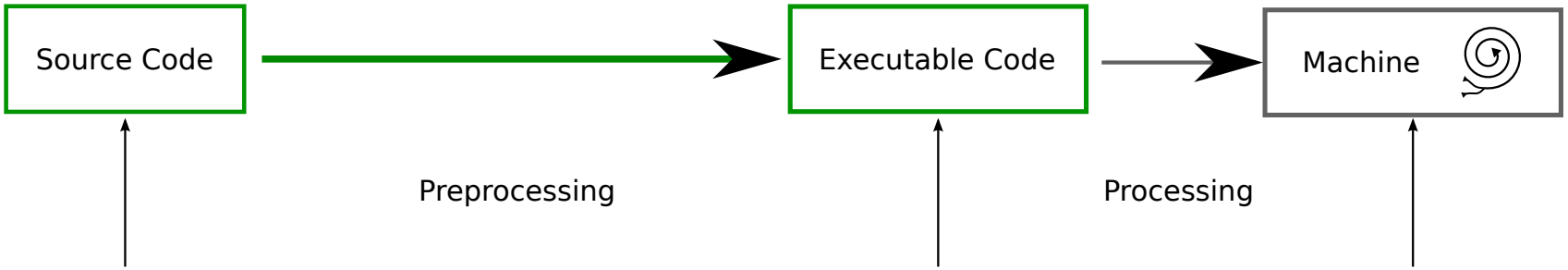
- From a high-level language to some intermediate language
- Statement by statement
- If an error is found, the interpreter stops and shows an error

Interpreter (High level)



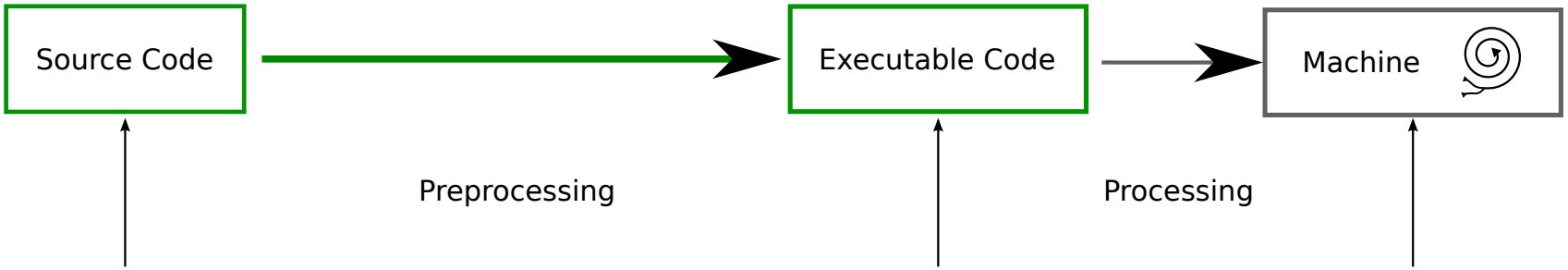
- From a high-level language to some intermediate language
- Statement by statement
- If an error is found, the interpreter stops and shows an error

Compiler (Execution flow)



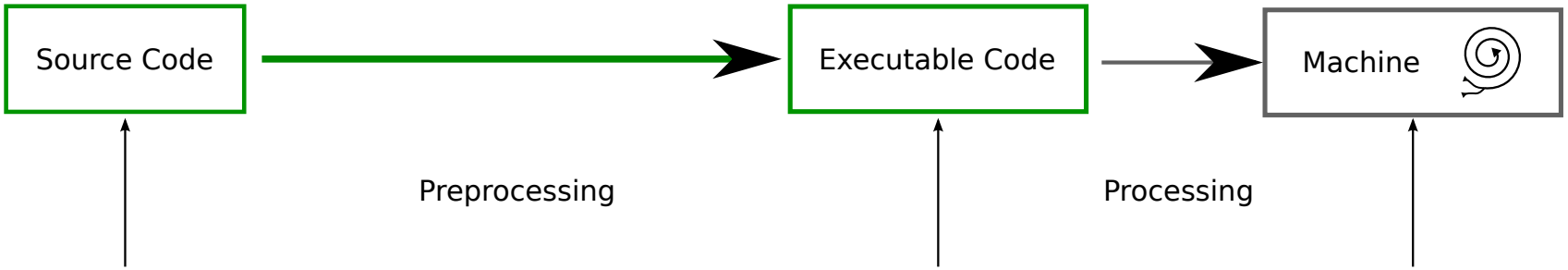
- Spends a lot of time analyzing and processing the program
- The resulting executable is machine-specific instructions
- The hardware executes the resulting code
- Program execution is fast(er)

Compiler (Execution flow)



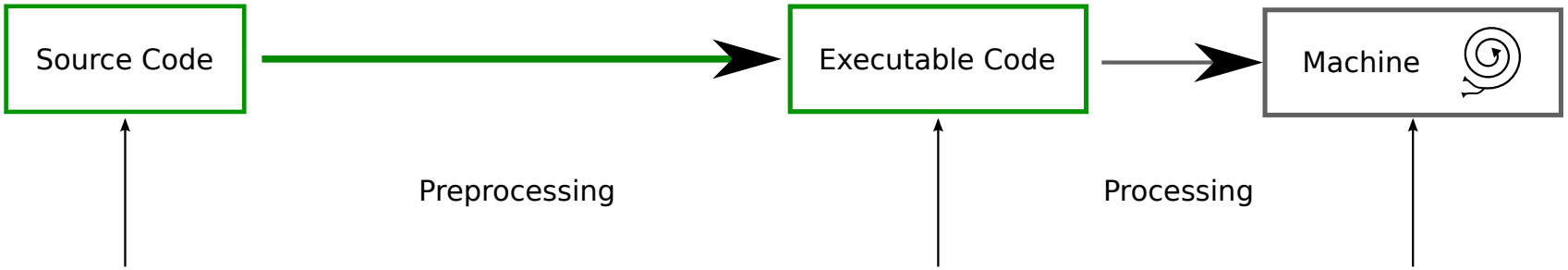
- Spends a lot of time analyzing and processing the program
- The resulting executable is machine-specific instructions
- The hardware executes the resulting code
- Program execution is fast(er)

Compiler (Execution flow)



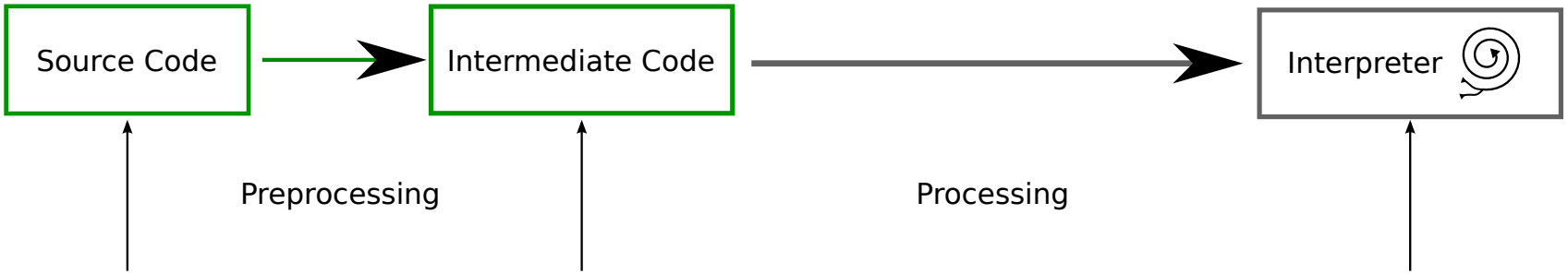
- Spends a lot of time analyzing and processing the program
- The resulting executable is machine-specific instructions
- The hardware executes the resulting code
- Program execution is fast(er)

Compiler (Execution flow)



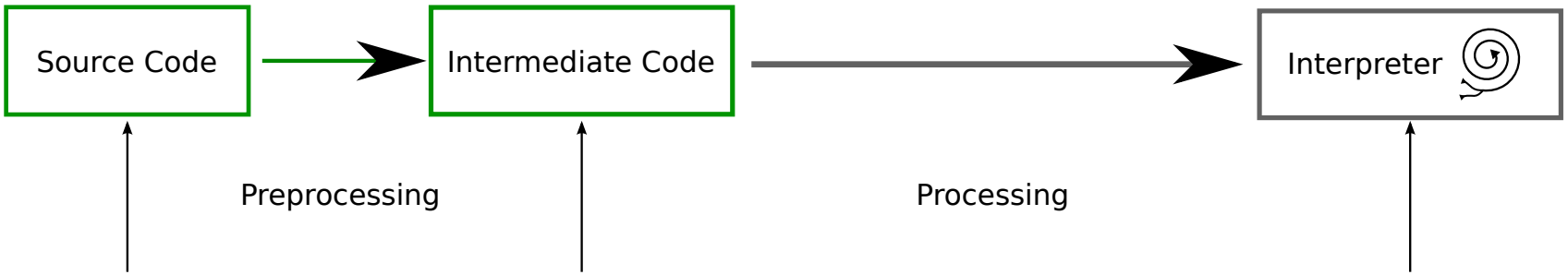
- Spends a lot of time analyzing and processing the program
- The resulting executable is machine-specific instructions
- The hardware executes the resulting code
- Program execution is fast(er)

Interpreter (Execution flow)



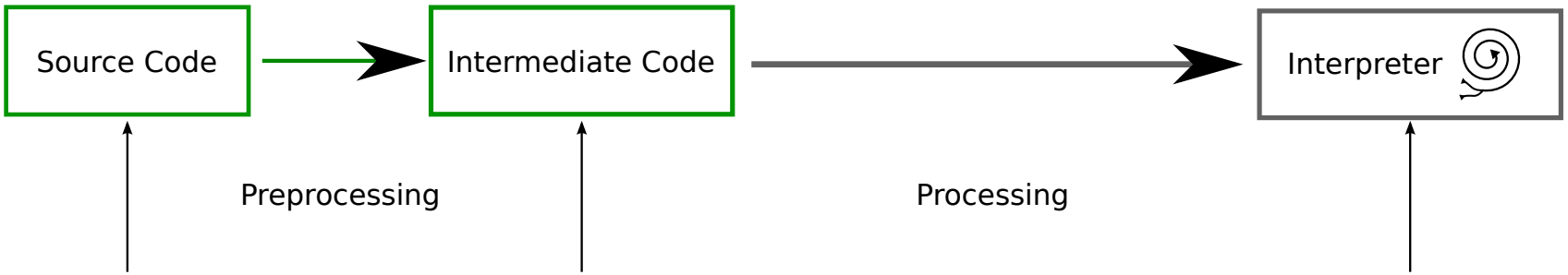
- Relatively little time is spent analyzing and processing the program
- The resulting code is some sort of intermediate code
- The intermediate code is interpreted by another program
- Program execution is slow(er)

Interpreter (Execution flow)



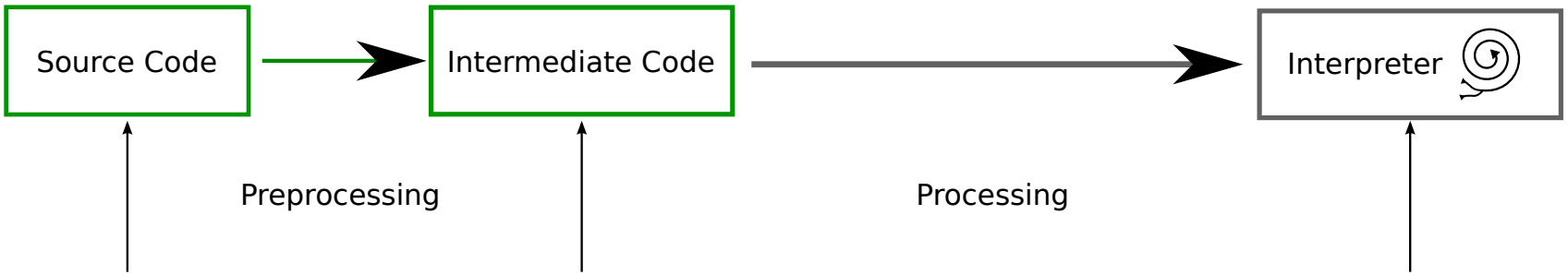
- Relatively little time is spent analyzing and processing the program
- The resulting code is some sort of intermediate code
- The intermediate code is interpreted by another program
- Program execution is slow(er)

Interpreter (Execution flow)



- Relatively little time is spent analyzing and processing the program
- The resulting code is some sort of intermediate code
- The intermediate code is interpreted by another program
- Program execution is slow(er)

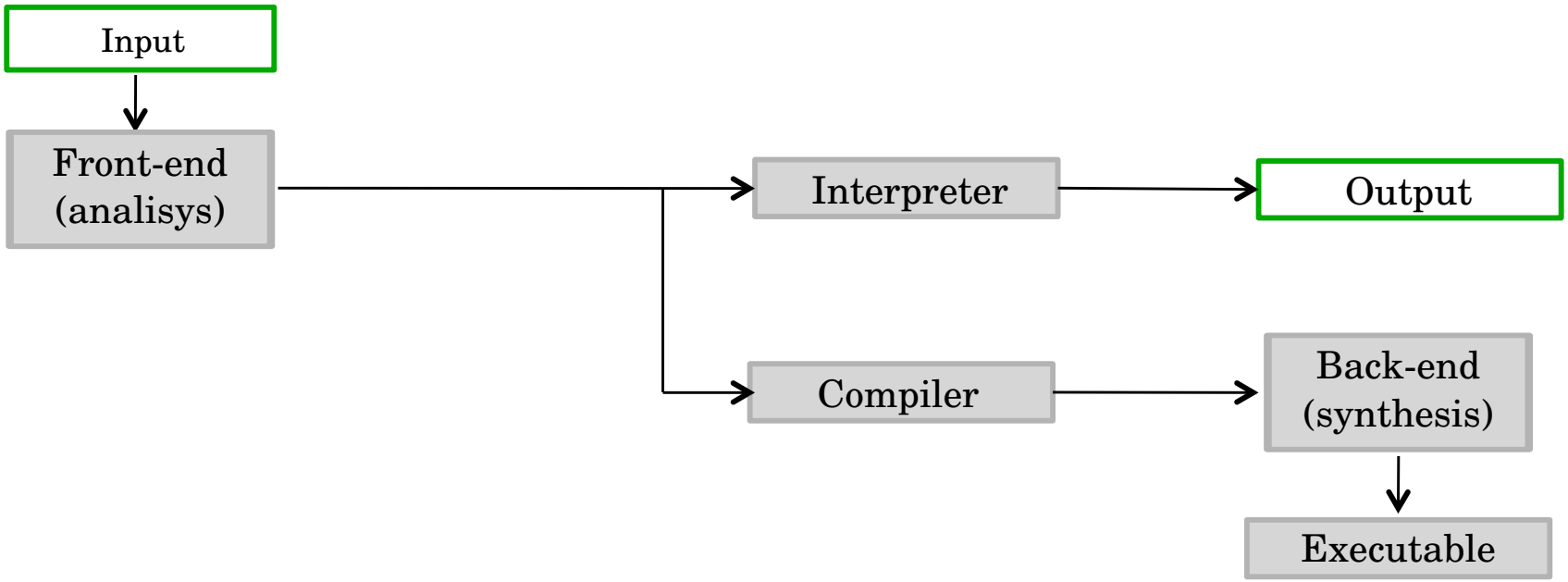
Interpreter (Execution flow)



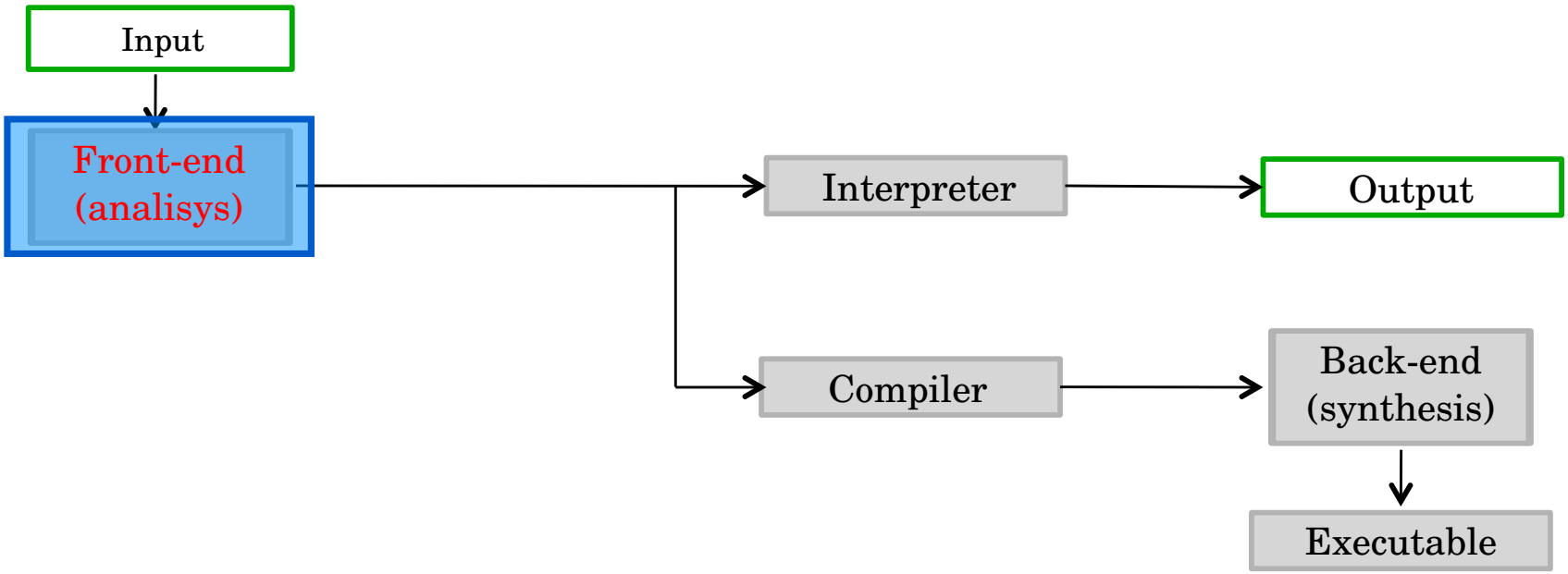
- Relatively little time is spent analyzing and processing the program
- The resulting code is some sort of intermediate code
- The intermediate code is interpreted by another program
- Program execution is slow(er)

Outline

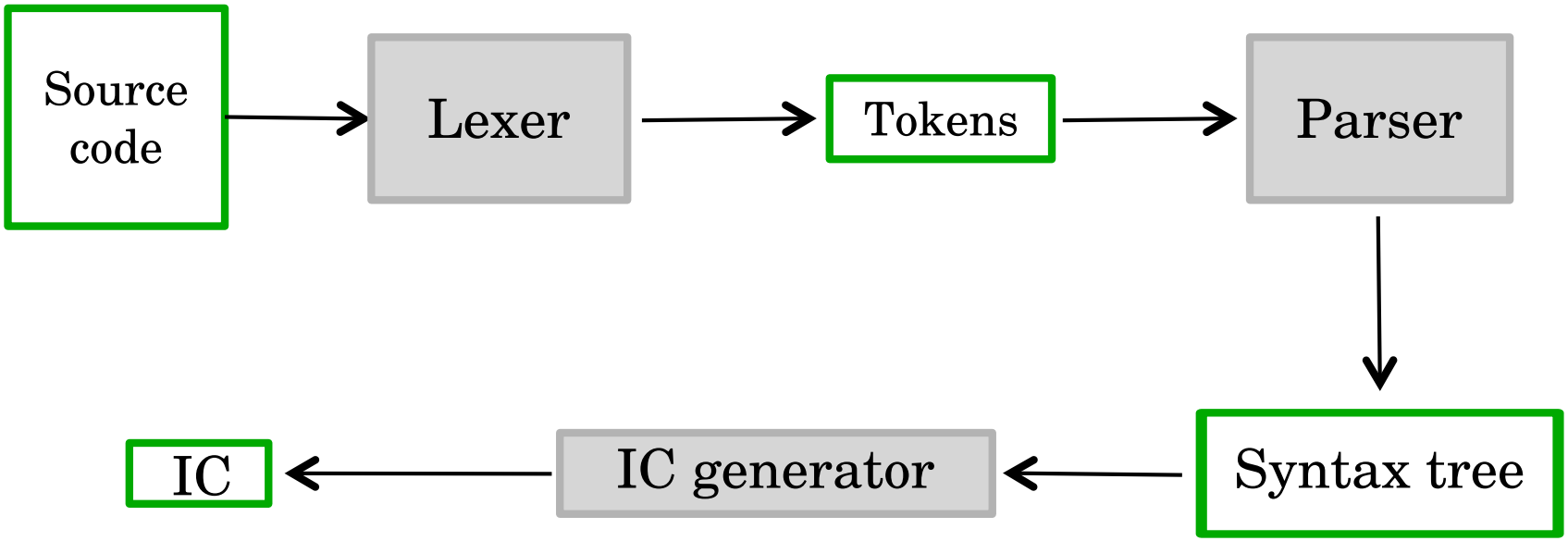
A little more detailed view...



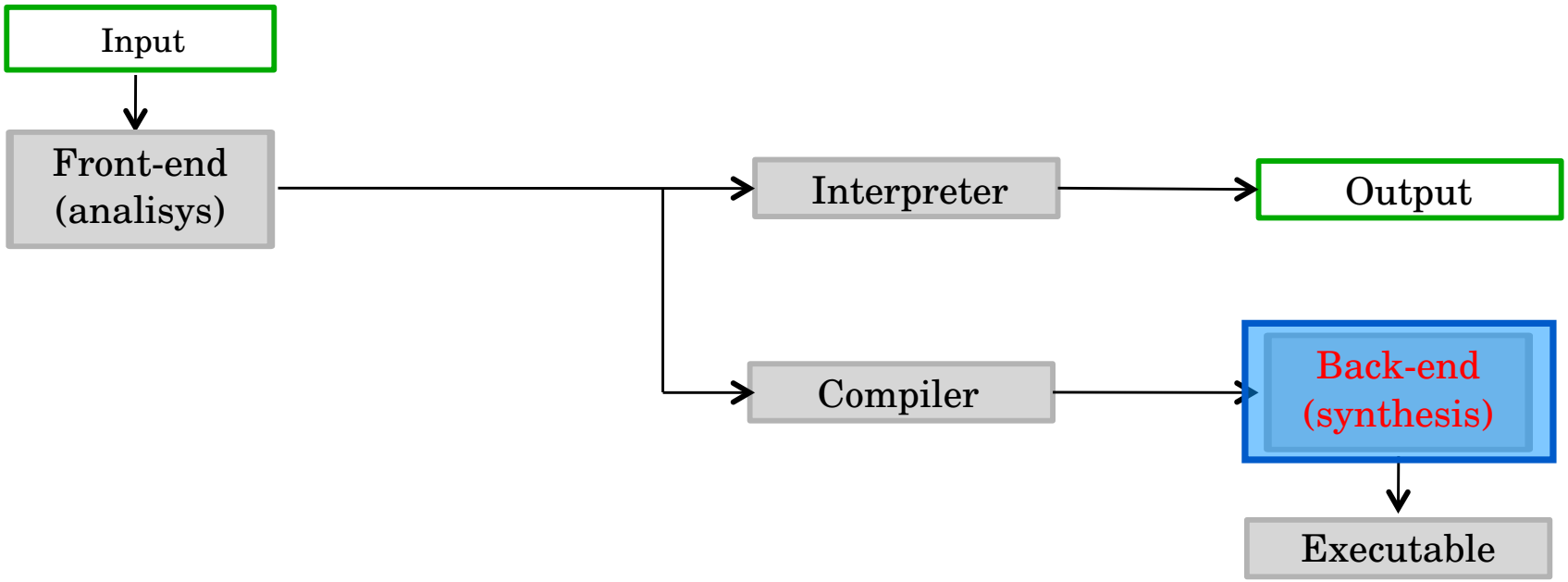
A little more detailed view...



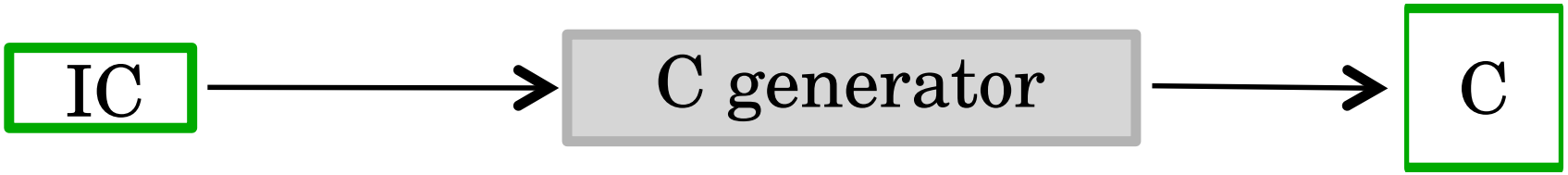
Front-end details



A little more detailed view...



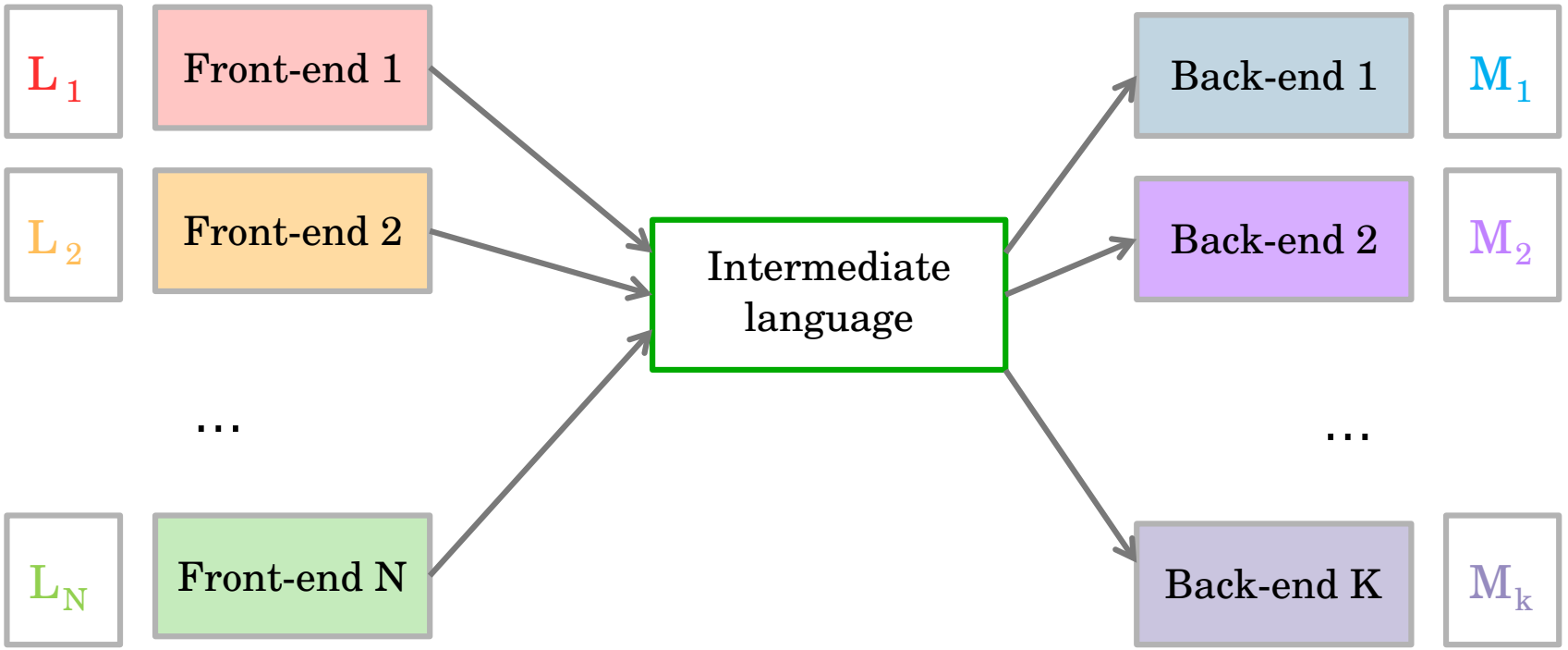
Back-end details



Is responsible for emitting the final version of the source program

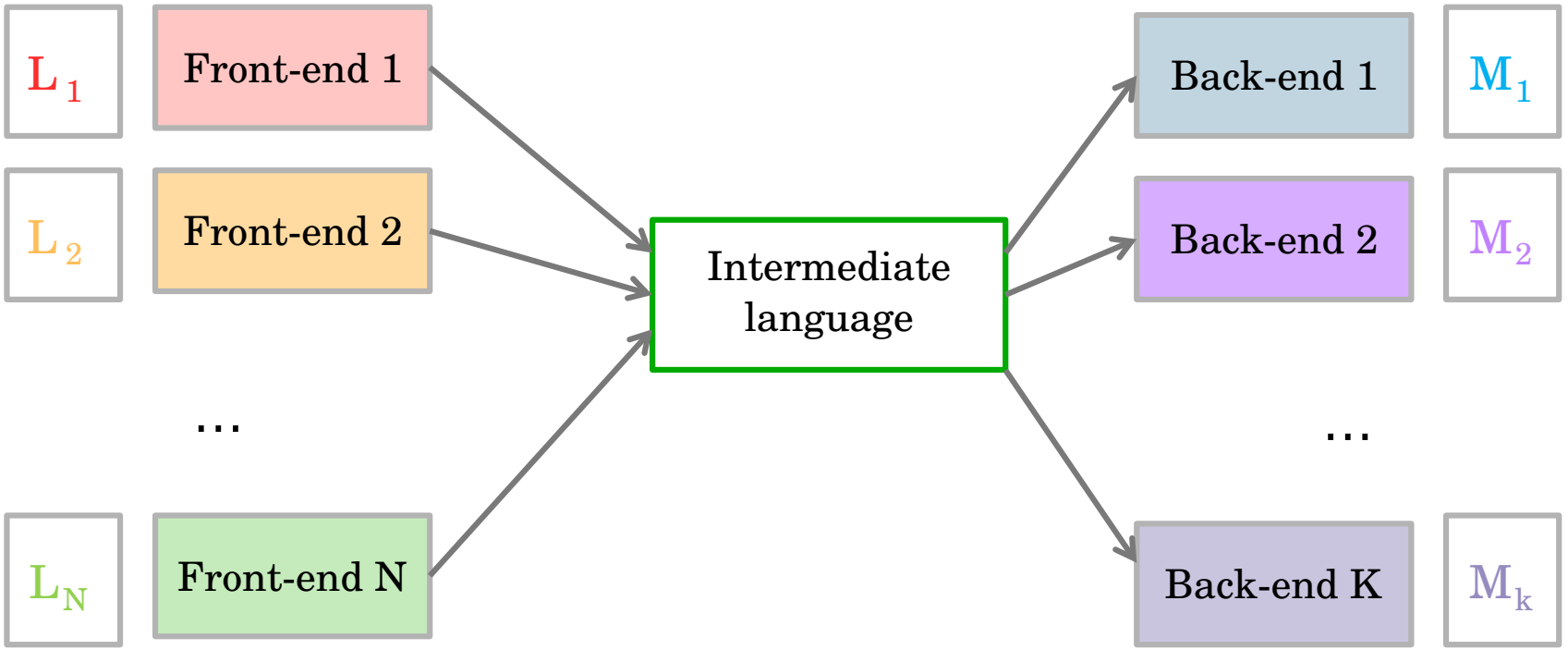
1. Instruction selection
2. Register allocation
3. Memory management
4. Instruction scheduling

Multiple front-ends and back-ends



- Reuse the same front-end for different machines
- Reuse the same back-end for different source languages

Multiple front-ends and back-ends



- Reuse the same front-end for different machines
- Reuse the same back-end for different source languages