Compiler Construction
Lent Term 2021

```
int main( int argc, char *argv[] )
{
  printf("hello world\n");
  return 0;
}
```

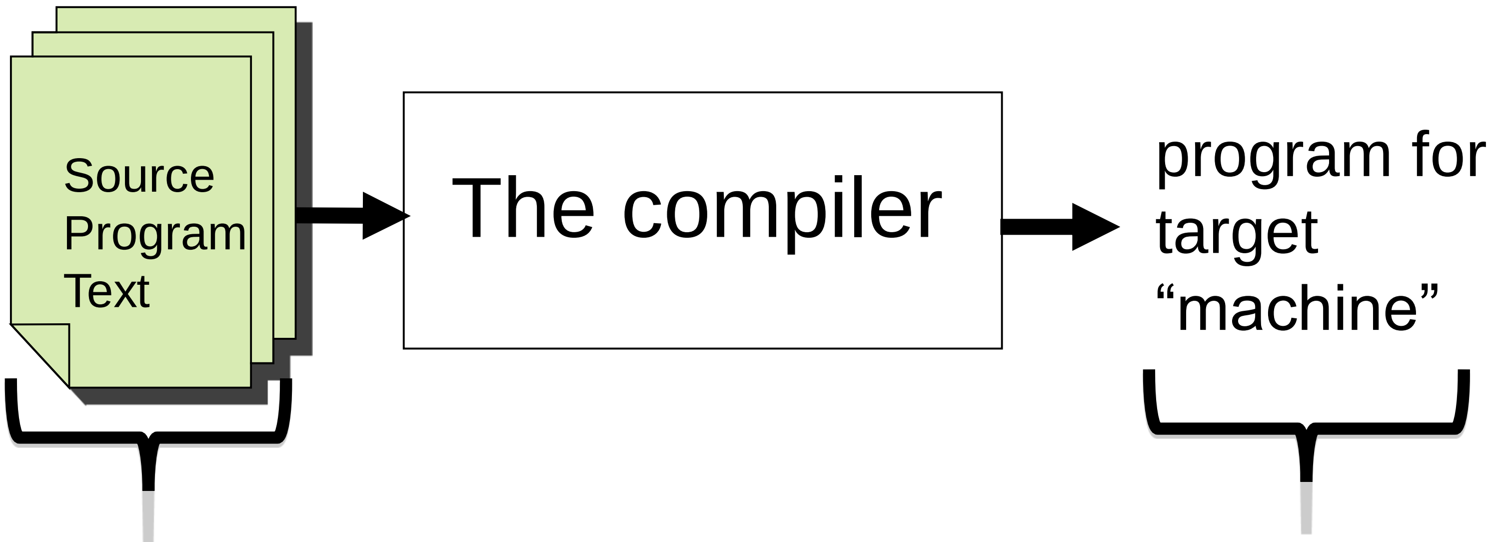
```
.LC0:
        .string     "hello world"
        .text
        .globl      main
        .type       main, @function
main:
.LFB0:
        .cfi_startproc
        pushq       %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq        %rsp, %rbp
        .cfi_def_cfa_register 6
        subq        $16, %rsp
        movl        %edi, -4(%rbp)
        movq        %rsi, -16(%rbp)
        movl        $.LC0, %edi
        call  puts
        movl        $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

Timothy G. Griffin
tgg22@cam.ac.uk
Computer Laboratory
University of Cambridge

Why Study Compilers?

- **Although many of the basic ideas were developed over 60 years ago, compiler construction is still an evolving and active area of research and development.**
- **Compilers are intimately related to programming language design and evolution.**
- **Compilers are a Computer Science success story illustrating the hallmarks of our field --- higher-level abstractions implemented with lower-level abstractions.**
- **Every Computer Scientist should have a basic understanding of how compilers work.**

Compilation is a special kind of translation



Just text – no way to run program!

We have a “machine” to run this!

A good compiler should …

- **<u>be correct in the sense that meaning is preserved</u>**
- **produce usable error messages**
- **generate efficient code**
- **itself be efficient**
- **be well-structured and maintainable**

This course! (be correct in the sense that meaning is preserved; produce usable error messages)

OptComp, Part II (generate efficient code)

Pick any 2?

Just 1?

Mind The Gap

High Level Language

- “Machine” independent
- Complex syntax
- Complex type system
- Variables
- Nested scope
- Procedures, functions
- Objects
- Modules
- …

Typical Target Language

- “Machine” specific
- Simple syntax
- Simple types
- memory, registers, words
- Single flat scope

Help!!! Where do we begin???

The Gap, illustrated

```
public class Fibonacci {
    public static long fib(int m) {
        if (m == 0) return 1;
        else if (m == 1) return 1;
            else return
                    fib(m - 1) + fib(m - 2);
    }
    public static void
        main(String[] args) {
        int m =
            Integer.parseInt(args[0]);
        System.out.println(
            fib(m) + "\n");
    }
}
```

javac Fibonacci.java
javap –c Fibonacci.class

```
public class Fibonacci {
 public Fibonacci();
  Code:
    0: aload_0
    1: invokespecial #1
    4: return
 public static long fib(int);
  Code:
    0: iload_0
    1: ifne          6
    4: lconst_1
    5: lreturn
    6: iload_0
    7: iconst_1
    8: if_icmpne     13
   11: lconst_1
   12: lreturn
   13: iload_0
   14: iconst_1
   15: isub
   16: invokestatic  #2
   19: iload_0
   20: iconst_2
   21: isub
   22: invokestatic  #2
   25: ladd
   26: lreturn
 public static void
   main(java.lang.String[]);
  Code:
    0: aload_0
    1: iconst_0
    2: aaload
    3: invokestatic  #3
    6: istore_1
    7: getstatic     #4
   10: new           #5
   13: dup
   14: invokespecial #6
   17: iload_1
   18: invokestatic  #2
   21: invokevirtual #7
   24: ldc           #8
   26: invokevirtual #9
   29: invokevirtual #10
   32: invokevirtual #11
   35: return
}
```

JVM bytecodes

The Gap, illustrated

fib.ml

```
(* fib : int -> int *)
let rec fib m =
    if m = 0
    then 1
    else if m = 1
         then 1
         else fib(m - 1) + fib (m - 2)
```

ocamlc –dinstr fib.ml

```
branch L2
L1:     acc 0
push
const 0
eqint
branchifnot L4
const 1
return 1
L4:     acc 0
push
const 1
eqint
branchifnot L3
const 1
return 1
L3:     acc 0
offsetint -2
push
offsetclosure 0
apply 1
push
acc 1
offsetint -1
push
offsetclosure 0
apply 1
addint
return 1
L2:     closurerec 1, 0
acc 0
makeblock 1, 0
pop 1
setglobal Fib!
```

OCaml VM bytecodes

The Gap, illustrated

fib.c

```
#include<stdio.h>

int Fibonacci(int);
int main()
{
   int n;
   scanf("%d",&n);
   printf("%d\n", Fibonacci(n));
   return 0;
}

int Fibonacci(int n)
{
   if ( n == 0 ) return 0;
   else if ( n == 1 ) return 1;
   else return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

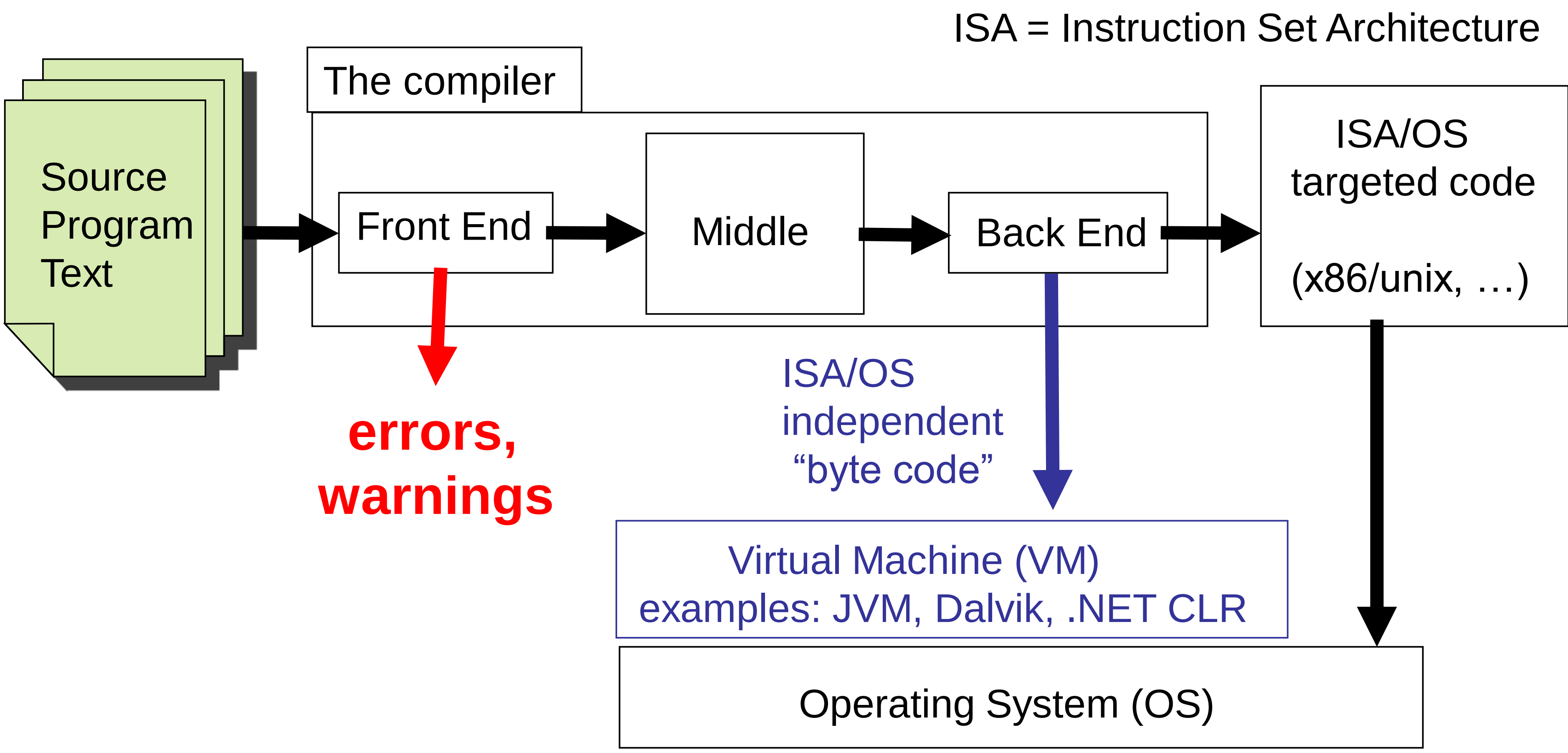
gcc –S fib.c

The Gap, illustrated

```
.section            __TEXT,__text,regular,pure_instructions
.globl              _main
.align              4, 0x90
_main:                          ## @main
.cfi_startproc
## BB#0:
pushq               %rbp
Ltmp2:
.cfi_def_cfa_offset 16
Ltmp3:
.cfi_offset %rbp, -16
movq                %rsp, %rbp
Ltmp4:
.cfi_def_cfa_register %rbp
subq                $16, %rsp
leaq                L_.str(%rip), %rdi
leaq                -8(%rbp), %rsi
movl                $0, -4(%rbp)
movb                $0, %al
callq               _scanf
movl                -8(%rbp), %edi
movl                %eax, -12(%rbp)        ## 4-byte Spill
callq               _Fibonacci
leaq                L_.str1(%rip), %rdi
movl                %eax, %esi
movb                $0, %al
callq               _printf
movl                $0, %esi
movl                %eax, -16(%rbp)        ## 4-byte Spill
movl                %esi, %eax
addq                $16, %rsp
popq                %rbp
ret
.cfi_endproc

.globl              _Fibonacci
.align              4, 0x90
_Fibonacci:                     ## @Fibonacci
.cfi_startproc
## BB#0:
pushq               %rbp
Ltmp7:
.cfi_def_cfa_offset 16
Ltmp8:
.cfi_offset %rbp, -16
movq                %rsp, %rbp
Ltmp9:
.cfi_def_cfa_register %rbp
subq                $16, %rsp
movl                %edi, -8(%rbp)
cmpl                $0, -8(%rbp)
jne                 LBB1_2
## BB#1:
movl                $0, -4(%rbp)
jmp                 LBB1_5
LBB1_2:
cmpl                $1, -8(%rbp)
jne                 LBB1_4
## BB#3:
movl                $1, -4(%rbp)
jmp                 LBB1_5
LBB1_4:
movl                -8(%rbp), %eax
subl                $1, %eax
movl                %eax, %edi
callq               _Fibonacci
movl                -8(%rbp), %edi
subl                $2, %edi
movl                %eax, -12(%rbp)        ## 4-byte Spill
callq               _Fibonacci
movl                -12(%rbp), %edi        ## 4-byte Reload
addl                %eax, %edi
movl                %edi, -4(%rbp)
LBB1_5:
movl                -4(%rbp), %eax
addq                $16, %rsp
popq                %rbp
ret
.cfi_endproc

.section            __TEXT,__cstring,cstring_literals
L_.str:                         ## @.str
.asciz              "%d"

L_.str1:                        ## @.str1
.asciz              "%d\n"


.subsections_via_symbols
```

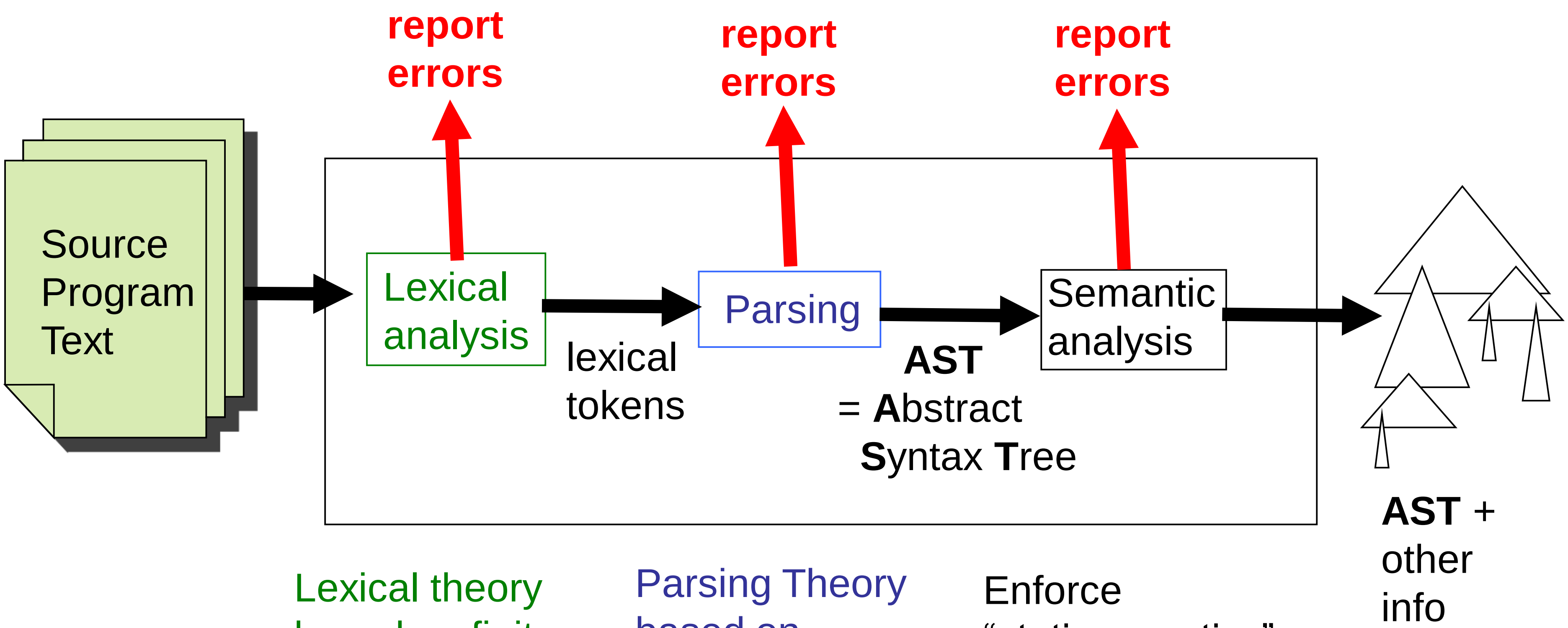
x86/Mac OS

Conceptual view of a typical compiler



Key to bridging The Gap : divide and conquer.
The Big Leap is broken into small steps.
Each step broken into yet smaller steps …

The shape of a typical “front end”



Lexical theory based on finite automaton and regular expressions

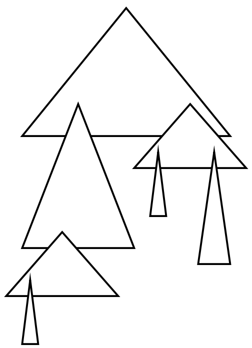
Parsing Theory based on push-down automaton and context-free grammars

Enforce “static sematics” of language: type checking, def/use rules, and so on (**SPL**!)

The AST output from the front-end should represent a legal program in the source language. (“Legal” of course does not mean “bug-free”!)

SPL = Semantics of Programming Languages, Part 1B

The middle



AST +
other
info

--High-level to
low-level

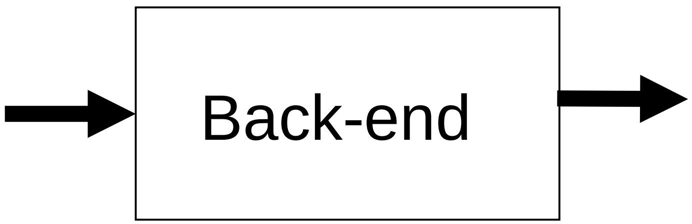
--Optimisations

Low-level
retargetable
representation

Trade-off: with more optimisations the generated code is (normally) **faster**, but the compiler is **slower**

The back-end

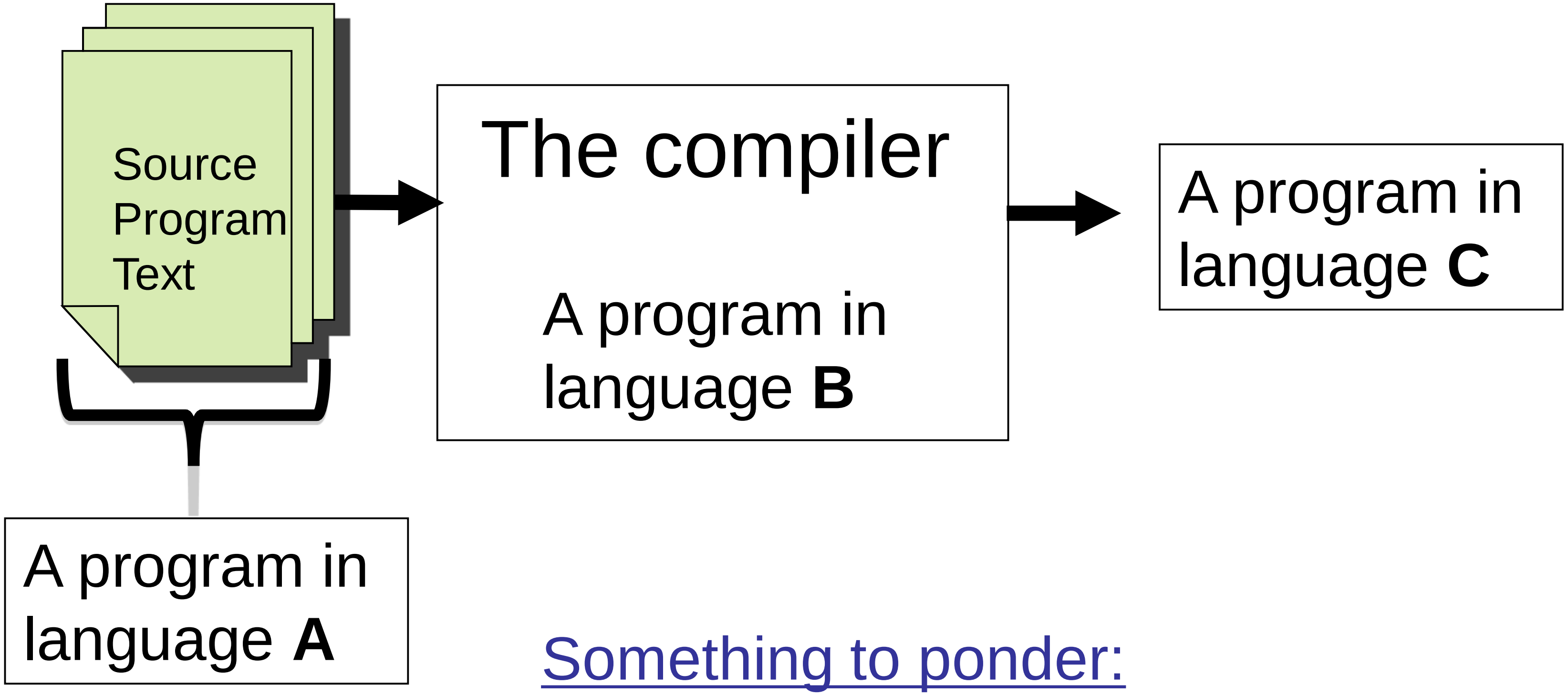
Low-level **retargetable** representation



- JVM bytecodes
- x86/Linux
- x86/MacOS
- x86/FreeBSD
- x86/Windows
- ARM/Android
- ….
- ….

- Requires intimate knowledge of instruction set and details of target machine
- When generating assembler, need to understand details of OS interface
- Target-dependent optimisations happen here!

Compilers must be compiled



Something to ponder:
A compiler is just a program.
But how did it get compiled?
The OCaml compiler is written in OCaml.

How was the compiler compiled?

The Shape of this Course

- Part I (Lectures 2 – 6) :Lexical analysis and parsing
- Part II (Lectures 7 – 16) : Development of the SLANG (Simple LANGuage) compiler. SLANG is based on L3 from 1B Semantics. A compiler for SLANG, written in Ocaml, will soon be posted on the course web page.