Compilers & Compilation SCC 312 (b)

Lancaster University

Unit 1 : Introduction

Barry Porter
b.f.porter@lancaster.ac.uk

Compilers & Compilation SCC 312 (b)

Lancaster University

Unit 1 : Introduction

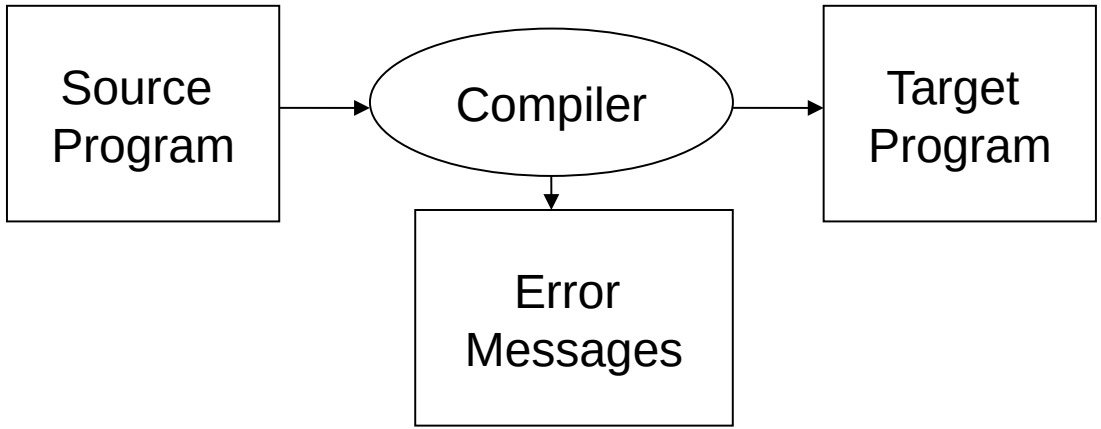
Barry Porter
b.f.porter@lancaster.ac.uk

Aims of this Unit

- To introduce the concept and function of a compiler
- To introduce the modular architecture of a compiler
- A very brief “History” of compilers and programming languages

A compiler

- Takes a *source program* in a *source language*
  - And translates it into an equivalent *target program* in a *target language*
- It also provides *error messages based on its parsing or code analysis*



The History of Programming Languages

Lancaster University

- the earliest computers were programmed in binary digits
- but soon assembler languages were introduced
  (see MIPS for an example of an “assembler” language)
- early high-level languages include autocodes, Fortran, Flowmatic, Algol 60, etc.
- later Lisp, Basic, Algol 68, Pascal, C, Ada, C++, Java, C# …
- As we go higher-level, we move further away from the machine the software will actually run on
- We therefore have an “abstract” machine

C / ASM / Machine Code

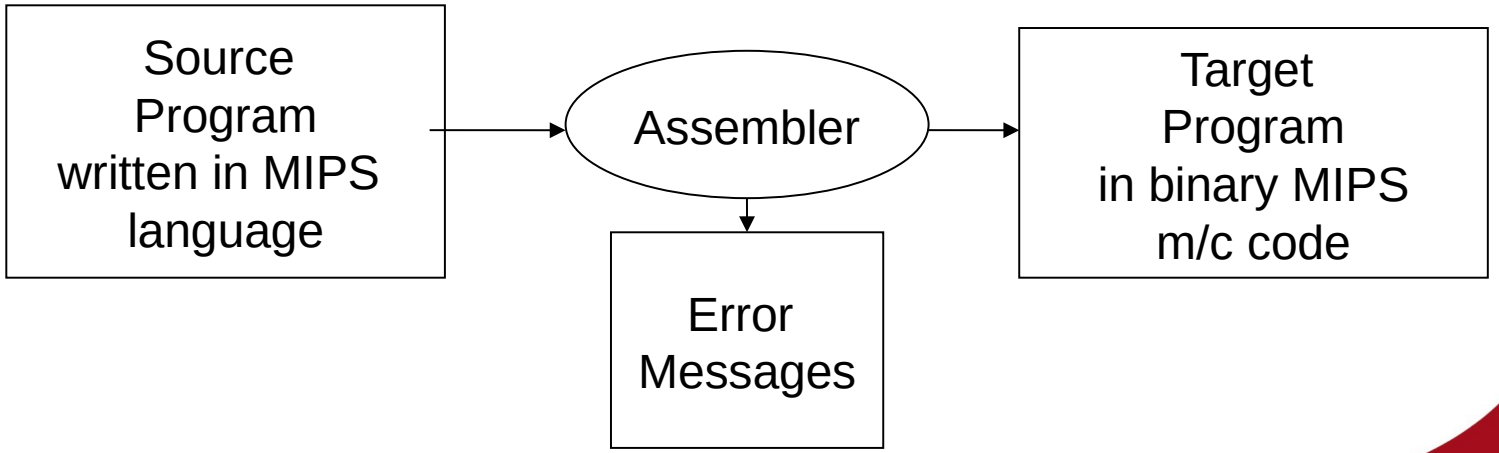
```c
#include <stdio.h>

int main(int argc, char* argv[])
    {
    printf("Hi there!\n");

    return 0;
    }
```

```
main:
.LFB0:
    .cfi_startproc
    endbr64
    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)
    leaq    .LC0(%rip), %rax
    movq    %rax, %rdi
    call    puts@PLT
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0"
    .section    .note.GNU-stack,"",@progbits
    .section    .note.gnu.property,"a"
    .align 8
    .long   1f - 0f
```

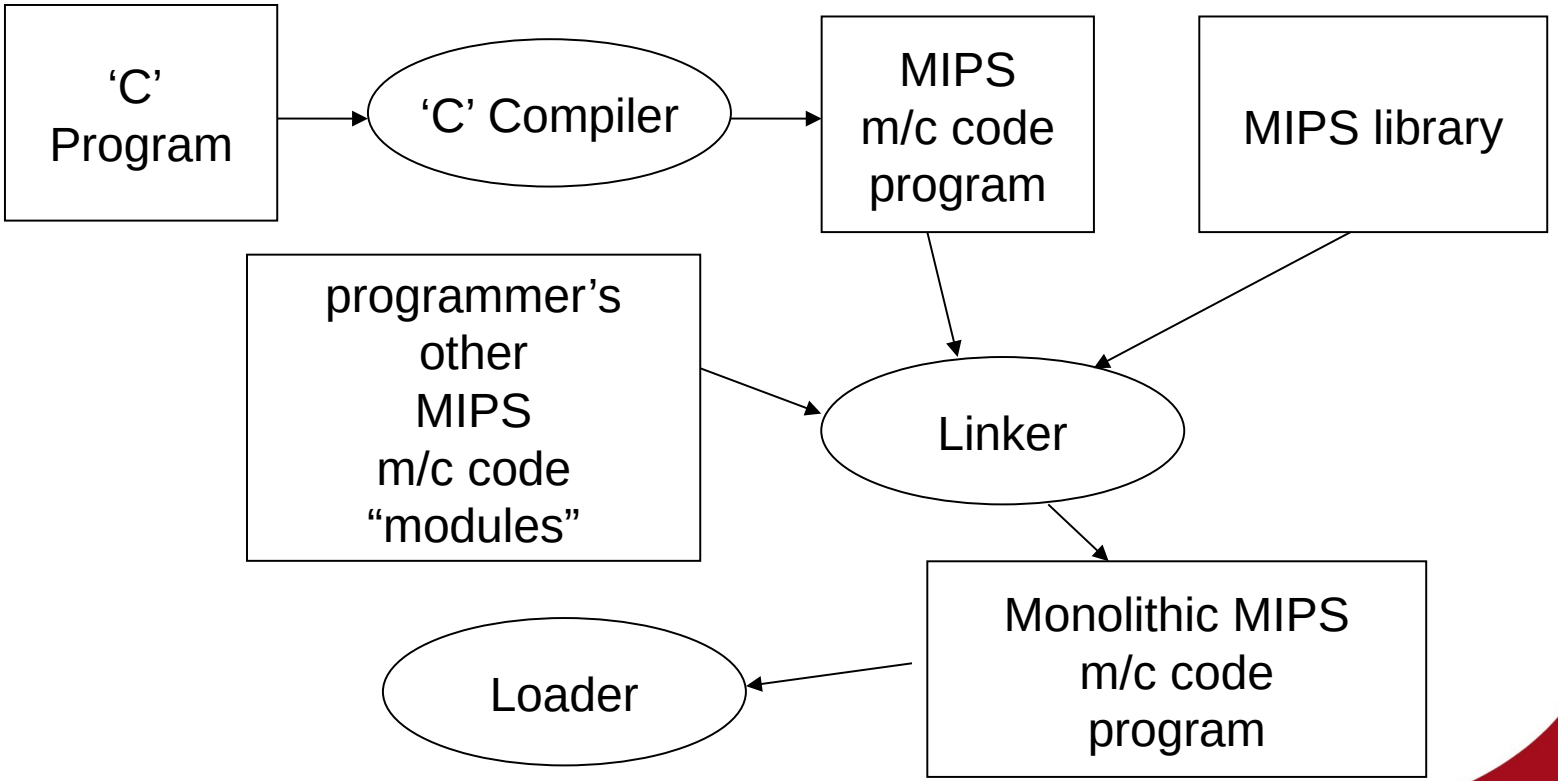
```
00011111100111
11010111100111
01110010100001
11110000100110
00010011100111
(etc.)
```

Assemblers

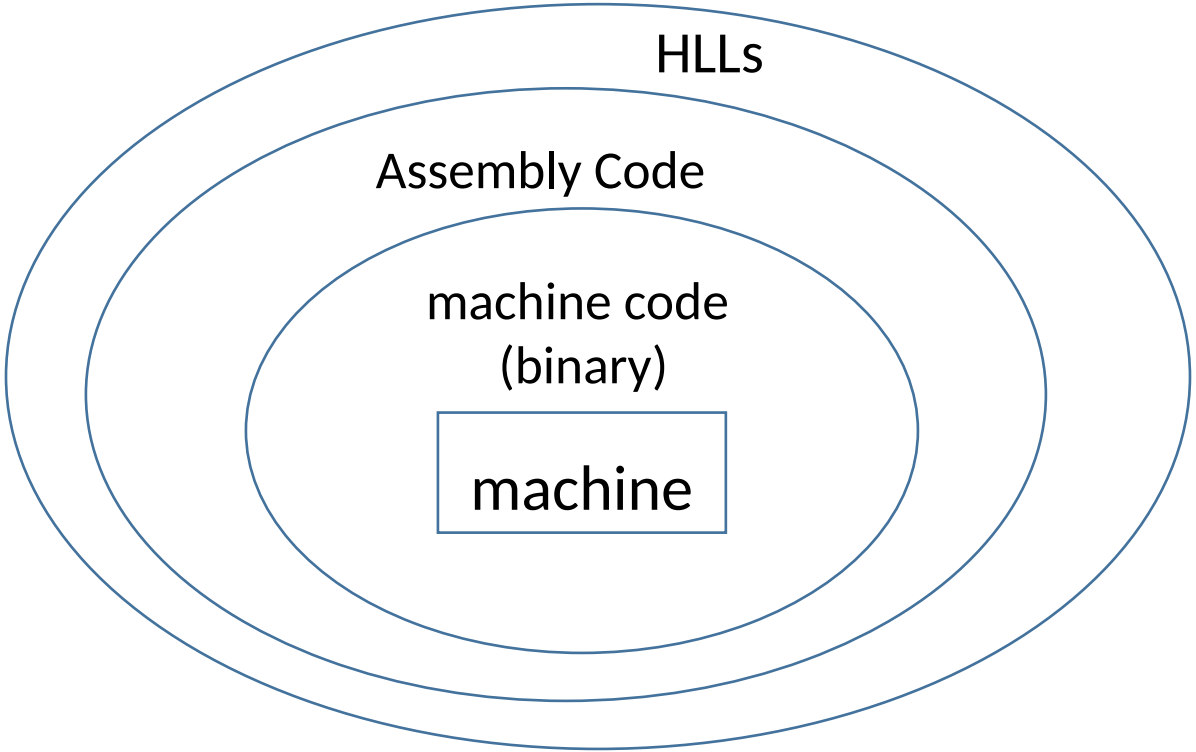
- Assemblers were much simpler programs than compilers were to become. They do share features in common as they both have to “translate” from the source to the target.



Typical Scenario



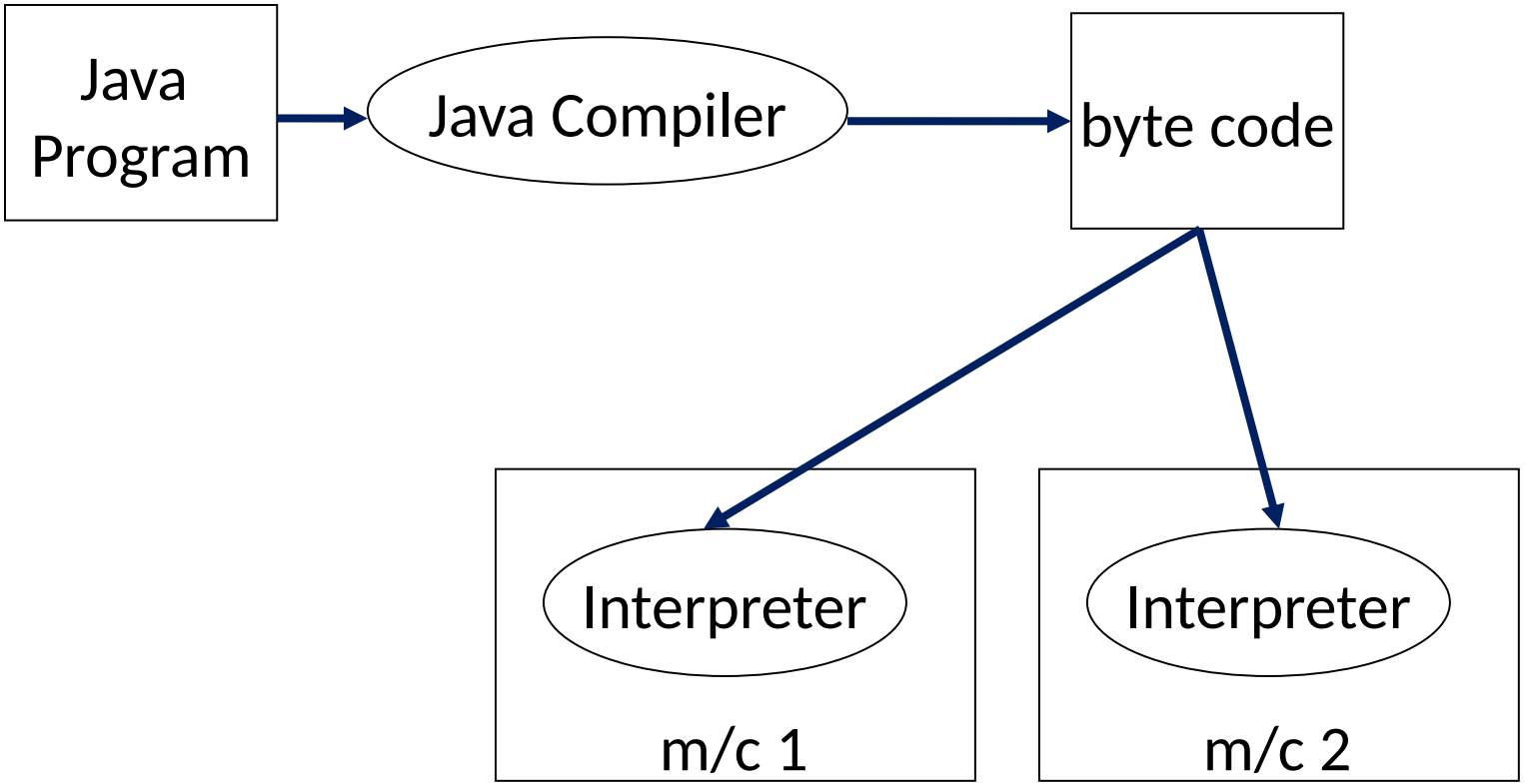
Distance from bits & bytes



Alternative Compiler Scenarios

- compilation to machine code
- interpretation
- output in assembler language
- output in intermediate code
- input from a pre-processor
- output used directly by a loader, or combined with other code by a link editor

Interpreter



We focus on ...

- compilation rather than interpretation (though similar techniques would be used in the latter)
- imperative high-level languages (Ada, Java), rather than functional or declarative languages (Lisp, Prolog)
  - syntactic analysis may be easier for the latter, but code generation is more complex

Other Applications of these Techniques

Lancaster University

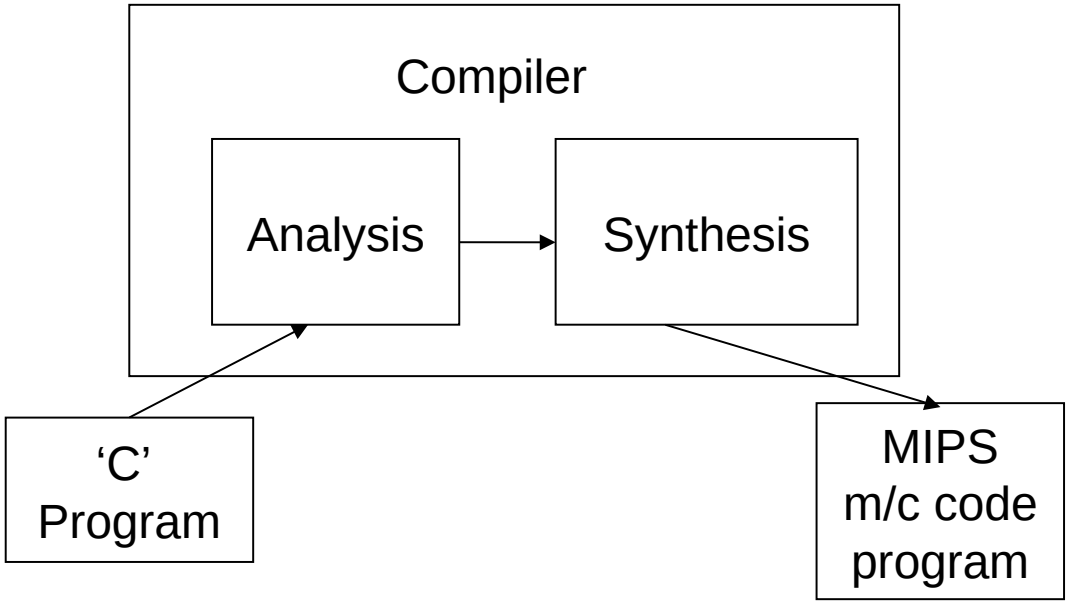
- nowadays there are formal languages for other things (Word macros, Postscript, HTML, etc.)
- any large program which processes complex input data may be considered as recognising a “language” (see, for example, XML)
- Natural Language Processing

The History of Compilers

- the first compilers were built without much theory
- now there is a lot of theory, particularly for syntax
- tools to help – UNIX brought lex and yacc, and derivatives
- also better programming languages in which to write compilers
  - also it is clearer what the issues are in designing programming languages

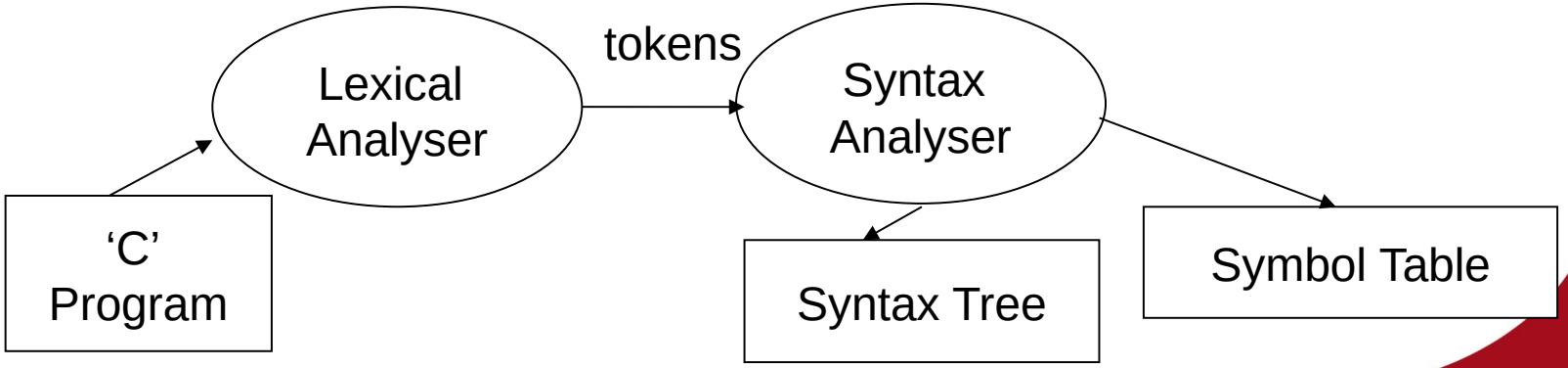
Compilers should be modular

- an analysis or front-end section
- a synthesis or back-end section



The Analysis Section

- the *lexical analyser* or scanner
- the *syntax analyser* or parser
- some *semantic analysis*
- we generate some form of intermediate representation and symbol table



The Synthesis Section

- a code-generation phase
- possibly one or more optimization phases
- all these phases may run in sequence, or in parallel

Structure of this Course

- We will follow the modular construction of a compiler.
- Lexical Analysis
- Syntax Analysis
- Code Generation

Learning Outcomes

- You should now be able to describe the function and architecture of a compiler at a high level.
- You should be able to talk about the modular components of a compiler, and what each module does.