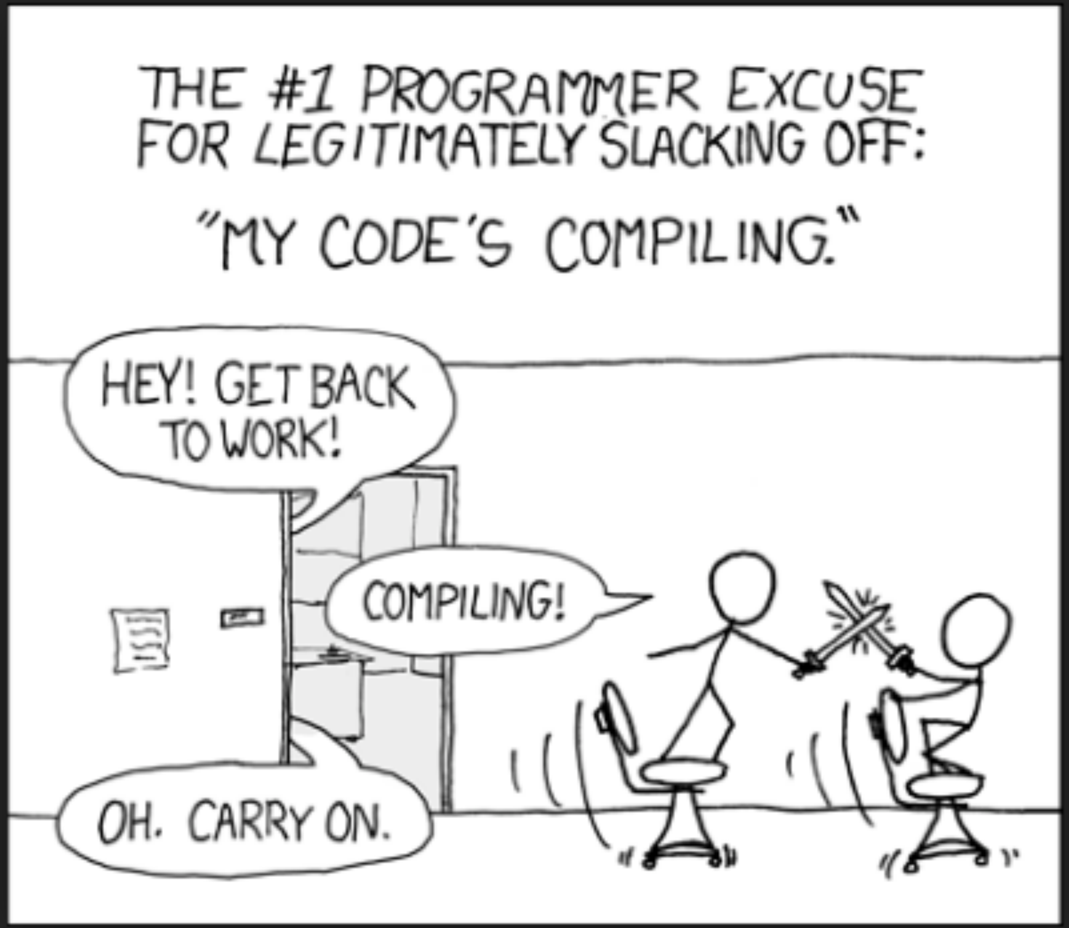
Compilers

```
module = Module(
    code="ELEE1147",
    name="Programming for Engineers",
    credits=15,
    module_leader="Seb Blair BEng(H) PGCAP MIET MIHEEM FHEA"
)
```

Download as a PDF

What we will cover

1. We will understand how ‘high’ and ‘low’ level programming languages are compiled to machine code so that it controls the hardware.
2. We will compare a number of programming languages and how they compile to machine code.

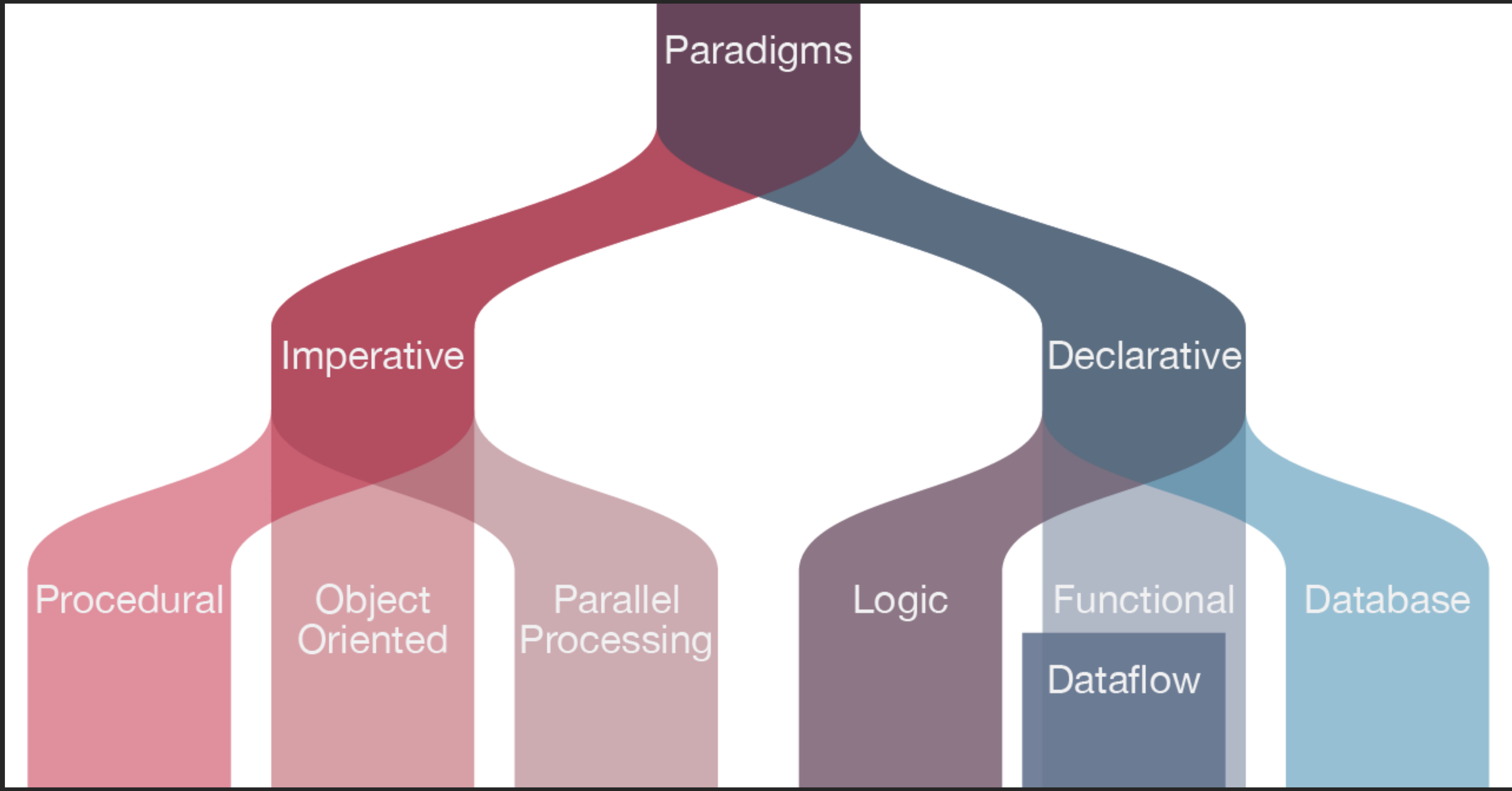


What is programming?

- A programming language is a computer language programmers use to develop software programs, scripts, or other sets of instructions for computers to execute...
- Automate process, Create digital records, Communication ,Simulation/emulation, …
- But ultimately programming provides instructions on how hardware is controlled, remember at the end of the day it is all zeroes and ones that represent electrical signals.

Programming Paradigms

- ~1605 programming lanaguages
- 94 Types
- 65 Paradigms



Human Language and Programming Languages

Are they all in English?

Linotte

It has been a developer for using French keywords:

```
BonjourLeMonde:
  début
    affiche "Bonjour le monde!"

------------------------

HelloWorld:
  beginning
    poster "Hello world!
```

Has a web engine for HTML and PHP and JSP.

SAKO

System Automatycznego Kodowania Operacji (Automatic Operation Encoding System) programming language, which uses polish as for its keywords:

```
LINIA
  TEKST:
  HELLO WORLD
KONIEC

---------------------------------

LINE
  TEXT:
  HELLO WORLD
END
```

Really only used in the late 1950s and early 1960s for the XYZ computers.

Rapira

Rapira is another awesome example of non-english programming languages. It uses Russian keywords:

```
ПРОЦ СТАРТ()

    ВЫВОД: 'Привет, мир!'

КОН ПРОЦ

-------------------

proc start()
     output: 'Hello, world!!!';
end proc
```

EPL

易语言 (Easy Programming Language, as known as EPL):

```
公开 类 启动类
{
  公开 静态 启动()
  {
    控制台.输出("你好，世界！");
  }
}

-------------------

public class startup class
{
  public static start()
  {
    console.output("Hello, World!");
  }
}
```

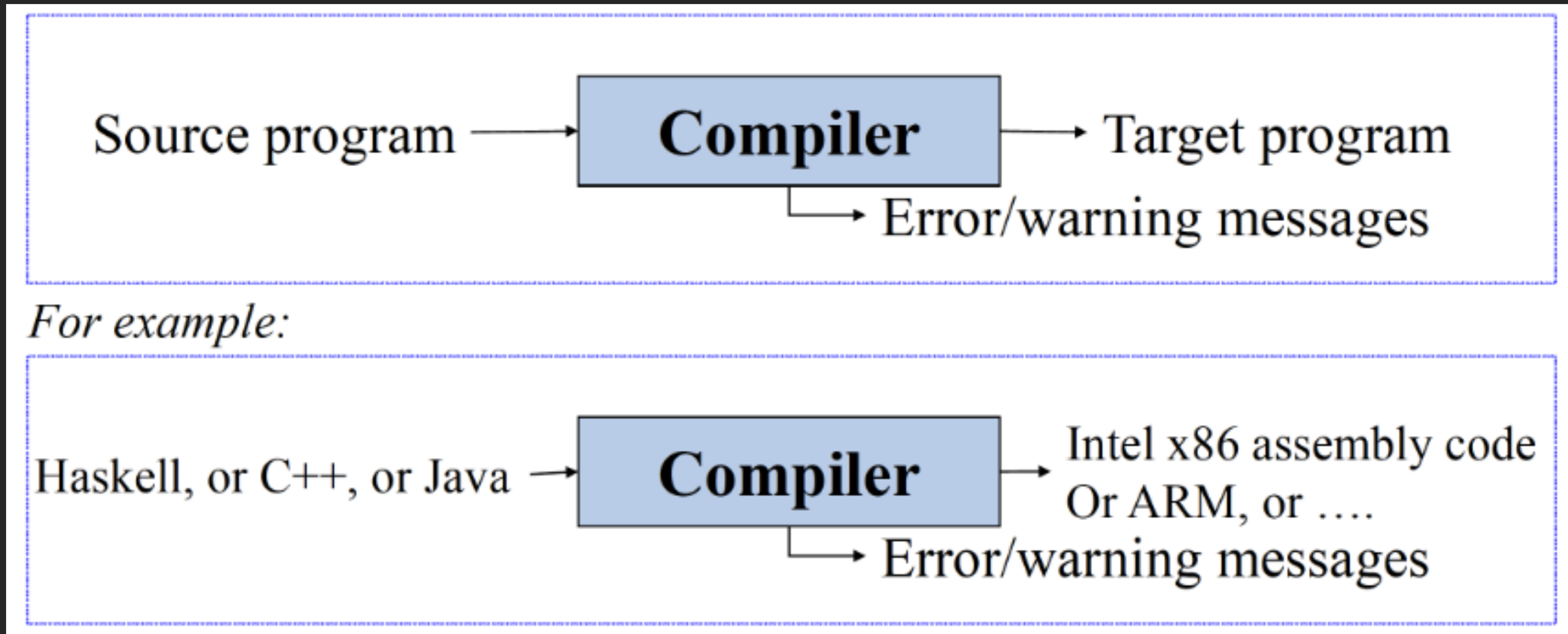
Compiler

A compiler is a program that `processes` source code written in a programming language.

- **Program Processing:** A compiler serves as a crucial tool in handling programs written in various programming languages.
- **Program Generation:** It functions as a program generator, capable of producing executable programs in a specified language.
- **Language Translation:** The compiler translates programs written in one language into equivalent programs in another language.

- **Increased Productivity:** Allows for faster and more efficient development by focusing on the logic and design rather than intricate details.
- **Enhanced Readability:** Code becomes more readable and understandable, facilitating collaboration and maintenance.
- **Code Portability:** Encourages code portability by minimizing dependencies on specific hardware or architecture

- Translates from one language into another
- Output a low level program which behaves as specified by
  the input, higher level program.
- Mediate between higher level human concepts, and the
  word by word data manipulation which the machine performs.



GCC

GCC compiler example

```
$ gcc -S -O test.c
```

Input file `test.c`

```c
int A;
int B;
test_fun()
{
  A = b + 123;
}
```

Output file `test.out`

```
.comm _A,4
.comm _B,4
_test_fun:
pushl %ebp
movl %esp,%ebp
movl _B,%eax
addl $123,%_A
movl %ebp,%esp
popl %ebp
ret
```

The flag `S` tells the compiler to produce assembly code, `O` turns optimisation on

Assembly code

- **Assembly code** is a `low-level` programming language that serves as an `interface` between `high-level` programming languages and the computer's `hardware`.
- **Human-Readable Machine Code:** Assembly code is a human-readable representation of machine code, making it more understandable than binary machine code.
- **Close to Hardware:** Unlike high-level languages, assembly code provides a direct correspondence to the architecture and operations of the underlying hardware.

Symbolic Representation

- Uses mnemonics and symbols to represent machine instructions, making it more comprehensible than raw machine code.

Binary	Opcode	Mnemonic	Description
1000 0111	87	`ADD A`	Add the contents of the register A to that of the accumulator
0011 1010	3A	`LDA`	Load data stored in the given memory address
0111 1001	79	`MOV A C`	Move data from register A to C
1100 0011	C3	`JMP`	Jump to instruction in specified memory address
1100 0001	C1	`POP` B	Pop from stack and copy to memory register B + C

Example

- The .data section declares a null-terminated string "Hello, Assembly!".
- `msg` - name of the varibable
- `db` - Define Byte
  - `msg db 'Hello', 0xA ; stores 6 bytes: H, e, l, l, o, newline`
    - `010001000`, `01100101`, `01101100`, `01101100`, `01101111`,
- `0xA` means new line
  - `00001010`

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(){

  //Declare and initialise the string
  char msg [] = "Hello, Assembly!";

  // Write the message to stdout
  write(1, msg, strlen(msg));

  // Exit the program with a return code of 0
  exit(0);
}
```

```
section .data
  msg db 'Hello, Assembly!', 0xA  ; Message with newline (0xA = '\n')

section .text
  global _start

_start:
  ; Write the message to stdout
  mov eax, 4          ; syscall: write
  mov ebx, 1          ; file descriptor: stdout
  mov ecx, msg        ; pointer to the message
  mov edx, 17           ; length of message (16 chars + newline)
  int 0x80            ; invoke the kernel

; Exit the program
  mov eax, 1          ; syscall: exit
  xor ebx, ebx        ; exit code 0
  int 0x80            ; invoke the kernel
```

Example:

- The .text section contains the program logic.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(){

  //Declare and initialise the string
  char msg [] = "Hello, Assembly!";

  // Write the message to stdout
  write(1, msg, strlen(msg));

  // Exit the program with a return code of 0
  exit(0);
}
```

```
section .data
  msg db 'Hello, Assembly!', 0xA  ; Message with newline (0xA = '\n')

section .text
  global _start

_start:
  ; Write the message to stdout
  mov eax, 4        ; syscall: write
  mov ebx, 1        ; file descriptor: stdout
  mov ecx, msg      ; pointer to the message
  mov edx, 17         ; length of message (16 chars + newline)
  int 0x80          ; invoke the kernel

; Exit the program
  mov eax, 1        ; syscall: exit
  xor ebx, ebx      ; exit code 0
  int 0x80          ; invoke the kernel
```

Example:

- The _start label marks the entry point of the program.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(){

  //Declare and initialise the string
  char msg [] = "Hello, Assembly!";

  // Write the message to stdout
  write(1, msg, strlen(msg));

  // Exit the program with a return code of 0
  exit(0);
}
```

```asm
section .data
  msg db 'Hello, Assembly!', 0xA  ; Message with newline (0xA = '\n')

section .text
  global _start

_start:
  ; Write the message to stdout
  mov eax, 4          ; syscall: write
  mov ebx, 1          ; file descriptor: stdout
  mov ecx, msg        ; pointer to the message
  mov edx, 17           ; length of message (16 chars + newline)
  int 0x80            ; invoke the kernel

; Exit the program
  mov eax, 1          ; syscall: exit
  xor ebx, ebx        ; exit code 0
  int 0x80            ; invoke the kernel
```

Example:

Register	Meaning	Value
`eax`	syscall number	4 = write
`ebx`	file descriptor	1 = stdout
`ecx`	pointer to buffer	address of msg
`edx`	number of bytes to write	17
`mov`	Copy data from one place to another	-
`int`	Software interrupt (calls the kernel) Control/Interrupt	-

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(){

  //Declare and initialise the string
  char msg [] = "Hello, Assembly!";

  // Write the message to stdout
  write(1, msg, strlen(msg));

  // Exit the program with a return code of 0
  exit(0);
}
```

```
section .data
  msg db 'Hello, Assembly!', 0xA  ; Message with newline (0xA = '\n')

section .text
  global _start

_start:
  ; Write the message to stdout
  mov eax, 4        ; syscall: write
  mov ebx, 1        ; file descriptor: stdout
  mov ecx, msg      ; pointer to the message
  mov edx, 17         ; length of message (16 chars + newline)
  int 0x80          ; invoke the kernel

; Exit the program
  mov eax, 1        ; syscall: exit
  xor ebx, ebx      ; exit code 0
  int 0x80          ; invoke the kernel
```

Example:

Register	Role	Value
eax	syscall number	1 = exit
ebx	exit code	0 = success
int	0x80 syscall trigger	invokes the kernel
xor	Bitwise XOR (commonly used to zero a register) Logical/ bitwise	-

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(){

  //Declare and initialise the string
  char msg [] = "Hello, Assembly!";

  // Write the message to stdout
  write(1, msg, strlen(msg));

  // Exit the program with a return code of 0
  exit(0);
}
```

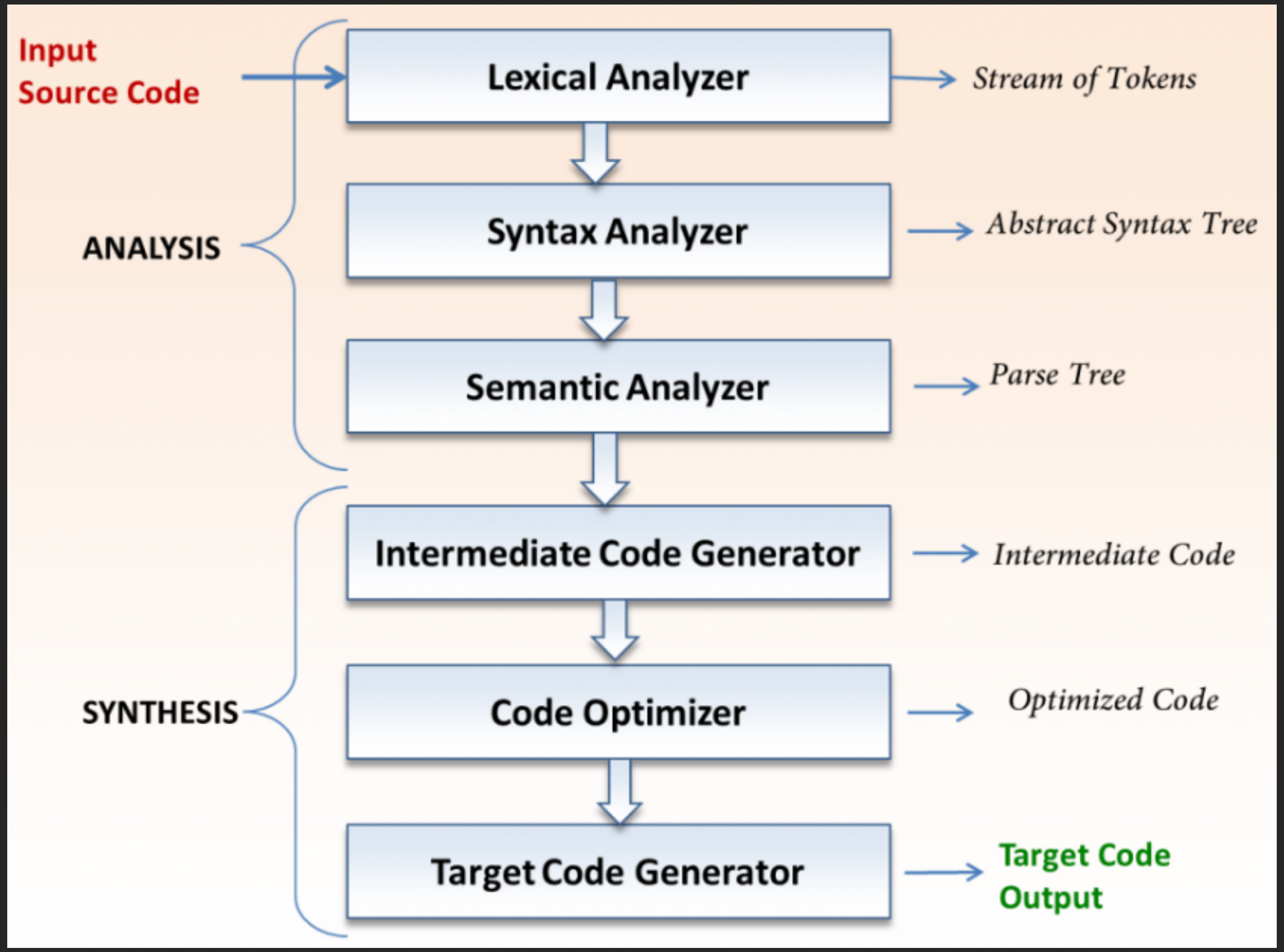
```asm
section .data
  msg db 'Hello, Assembly!', 0xA  ; Message with newline (0xA = '\n')

section .text
  global _start

_start:
  ; Write the message to stdout
  mov eax, 4          ; syscall: write
  mov ebx, 1          ; file descriptor: stdout
  mov ecx, msg        ; pointer to the message
  mov edx, 17           ; length of message (16 chars + newline)
  int 0x80            ; invoke the kernel

; Exit the program
  mov eax, 1          ; syscall: exit
  xor ebx, ebx        ; exit code 0
  int 0x80            ; invoke the kernel
```

Compiling Stages



Lexical Analysis

The compiler begins converting the series of characters into tokens

Token name	Example token values
identifier	`n`, `q`
keyword	`int`, `float`, `if`, `else`, `return`, `while`
separator	`{ }`, `( )`, `[ ]`, `;`
operator	`+`, `-`, `*`, `/`, `=`, `<`, `>`, `:`, `?`
literal	`True`, `false`, `6.02e23`, `"string"`
comment	`// this is a comment`

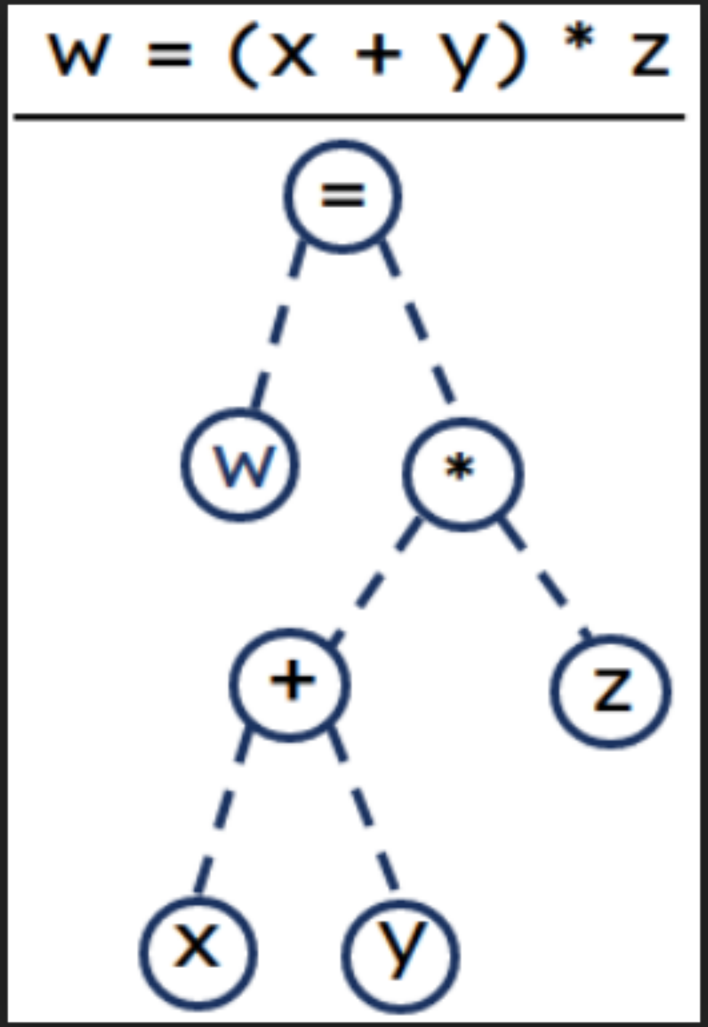
High Level Code

```
int n = 11;
float q = 1.618f;
if (n < 12)
{
  return q;
}
else
{
  return n;
}
```

Syntax Analysis

Syntax analysis is based on the rules based on the specific programming language by constructing the parse tree with the help of tokens.

- Interior node: record with an operator filed and two files for children
- Leaf: records with 2/more fields; one for token and other information about the token
- Ensure that the components of the program fit together meaningfully
- Gathers type information and checks for type compatibility
- Checks operands are permitted by the source language



Semantic Analyser

The Semantic Analyser checks for type mismatches, incompatible operands, improper function calls, undeclared variables, and more — including **implicit type conversions**.

```c
int n = 11;
float q = 1.618 * n;
```

- Type Promotion: `int → float`
- Operation: `float * float = float`
- Safe, no data loss

```c
float f = 3.9;
int x = f * 2;
```

- Promotes `2` to `2.0f`
- Computes `3.9 * 2.0 = 7.8`
- Then **casts** **`7.8 → 7`** **(truncation)**
- Type Conversion: `float → int`
- Potential data loss

Intermediate Code Generation

Removes unnecessary code lines.

Arranges the sequence of statements to speed up the execution of the program without wasting resources.

Original

```
a = int_to_float(10)
b = c * a
d = e + b
f = d
```

Stage	IR Code

Intermediate Representation	t1 = 10
	t2 = int_to_float t1
	t3 = c * t2
	t4 = e + t3
	f = t4

Optimised IR	t2 = 10.0f
	t3 = c * t2
	f = e + t3

Fully Optimised	f = e + (c * 10.0f)

Code Generation

Now we are going to see how we go from C to Assembly to machine code…

```c
int square(int num) {
    return num * num;
}
```

```
square:
  pushq   %rbp
  movq    %rsp, %rbp
  movl    %edi, -4(%rbp)
  movl    -4(%rbp), %eax
  imull   %eax, %eax
  popq    %rbp
  ret
```

	Memory Addresses
	0x0007556ff0e0
	0x0007556ff0df
	0x0007556ff0de
saved %rbp	**0x0007556ff0dd**
	...
	0x0007556ff0da
↑ %rbp	**0x0007556ff0d9**

Code Generation

Instruction sets up the stack frame, and how it maps to our memory layout.

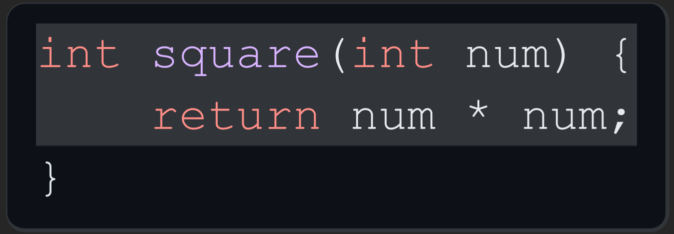
```c
int square(int num) {
    return num * num;
}
```

```
square:
  pushq   %rbp
  movq    %rsp, %rbp
  movl    %edi, -4(%rbp)
  movl    -4(%rbp), %eax
  imull   %eax, %eax
  popq    %rbp
  ret
```

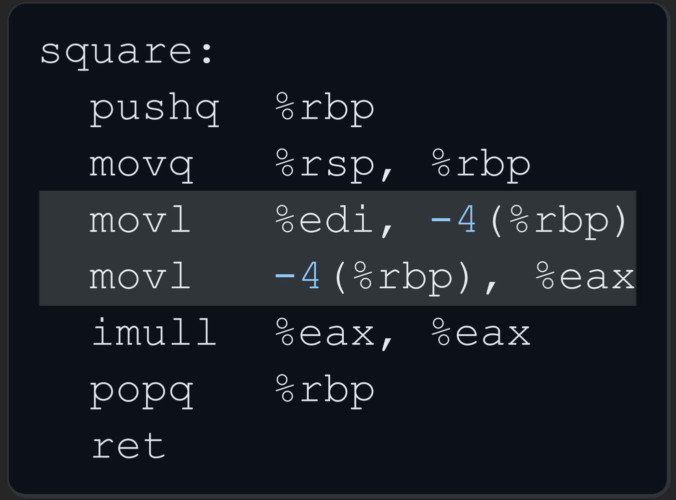
	Memory Addresses
	0x0007556ff0e0
	0x0007556ff0df
	0x0007556ff0de
saved %rbp	**0x0007556ff0dd**
	...
	0x0007556ff0da
↑ **new %rbp**	**0x0007556ff0d9**

Code Generation

Now we are going to see how we go from C to Assembly to machine code…



```c
int square(int num) {
    return num * num;
}
```



```
square:
  pushq   %rbp
  movq    %rsp, %rbp
  movl    %edi, -4(%rbp)
  movl    -4(%rbp), %eax
  imull   %eax, %eax
  popq    %rbp
  ret
```

	Memory Addresses
	0x0007556ff0e0
	0x0007556ff0df
	0x0007556ff0de
saved %rbp	**0x0007556ff0dd**
	...
	0x0007556ff0da
↓ **num**	**0x0007556ff0d5**

Code Generation

Now we look at the final operation: `imull %eax, %eax`, which computes `num * num`. This instruction directly maps to the C expression and uses the loaded value from the stack.

```c
int square(int num) {
    return num * num;
}
```

```
square:
  pushq   %rbp
  movq    %rsp, %rbp
  movl    %edi, -4(%rbp)
  movl    -4(%rbp), %eax
  imull   %eax, %eax
  popq    %rbp
  ret
```

	Memory Addresses
	0x0007556ff0e0
	0x0007556ff0df
	0x0007556ff0de
saved %rbp	**0x0007556ff0dd**
	...
	0x0007556ff0da
num	**0x0007556ff0d5**
eax = num*num	**(register)**

Code Generation

Finally, we examine the `popq %rbp` instruction — the function epilogue that restores the previous stack frame, preparing for return to the caller.

```c
int square(int num) {
    return num * num;
}
```

```
square:
  pushq   %rbp
  movq    %rsp, %rbp
  movl    %edi, -4(%rbp)
  movl    -4(%rbp), %eax
  imull   %eax, %eax
  popq    %rbp
  ret
```

	Memory Addresses
↑ **restored %rbp**	**0x0007556ff0dd**
	0x0007556ff0dc
	0x0007556ff0db
	0x0007556ff0da
num	0x0007556ff0d5

Code Generation: Assembly and Hex Representation

```c
int square(int num) {
    return num * num;
}
```

∴

```
square:
  pushq  %rbp
  movq   %rsp, %rbp
  movl   %edi, -4(%rbp)
  movl   -4(%rbp), %eax
  imull  %eax, %eax
  popq   %rbp
  ret
```

```
HEX
55
48 89 e5
89 7d fc
8b 45 fc
0f af c0
5d
c3
```

Symbol Management Table

A symbol table contains a record for each identifier with fields for the attributes of the identifier.

Operation	Function
`allocate`	to allocate a new empty symbol table
`free`	to remove all entries and free storage of symbol table
`lookup`	to search for a name and return a pointer to its entry
`insert`	to insert a name in a symbol table and return a pointer to its entry
`set_attribute`	to associate an atrribute to a given entry
`get_attribute`	to get an attribute associated with a given entry

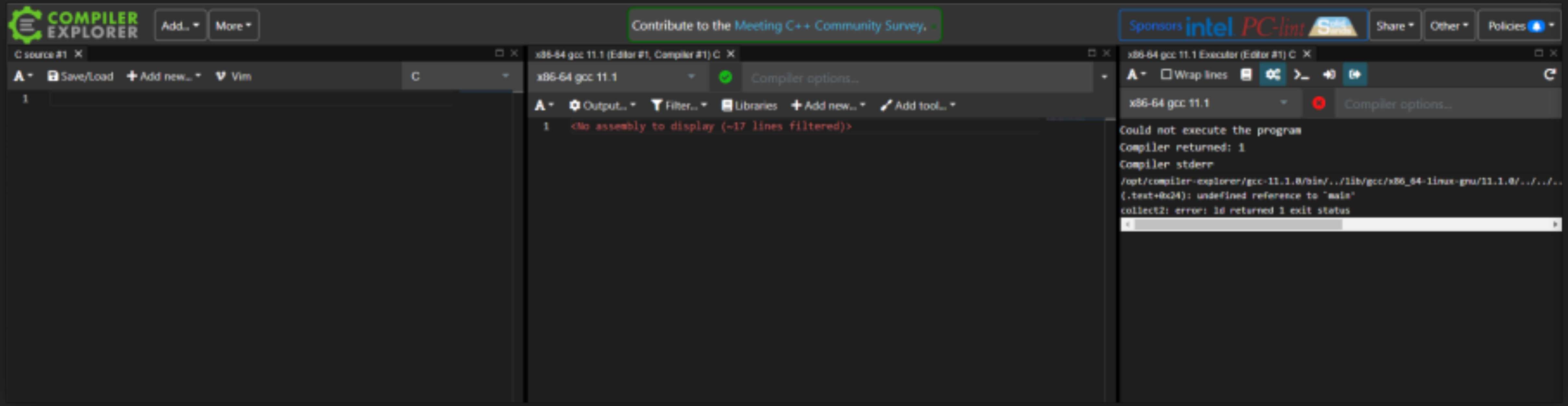
Error Handling Routine

During compilation process error(s) may occur in all the below-given phases:

- Lexical analyser: Wrongly spelled tokens
- Syntax analyser: Missing parenthesis
- Semantic analyser: Mismatched data types, missing arguments
- Intermediate code generator: Mismatched operands for an operator
- Code Optimizer: When the statement is not reachable
- Code Generator: Unreachable statements
- Symbol tables: Error of multiple declared identifiers

Labs

You are going experience programming in several languages <C , Python and Ada> to do similar operations, and see how the code compiles and the subsequent outputs!



```
<No assembly to display (~17 lines filtered)>
```

```
Could not execute the program
Compiler returned: 1
Compiler stderr
/opt/compiler-explorer/gcc-11.1.0/bin/../lib/gcc/x86_64-linux-gnu/11.1.0/../../..
(.text+0x24): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

https://uniofgreenwich.github.io/ELEE1147_Exercises/Compilers/compilers.html