Lec 1 Intro

Compilers

they translate code from one lang to another

Python → JS
C → x86

We want :

- Correctness — same behaviour as original code inlended] focus
- Code quality — is the compiled code good ? speed, size, etc.]
- Efficiency — is compile fast ?
- Useability — interface with human - helpful

Some history :

1943	Plankalkül	first high level lang	Konrad Zuse
1951	Formules	first self-hosting compiler (compiles itself)	Carrado Böhm
1952	A-0	term "compiler"	Grace Hopper
1952	Autocode		
1957	FORTRAN		
1958	ALGOL58		
1962	Garbage collector		

⋮

many active research

Course Structure

- Structuring a compiler
- Algorithms & Data structures
  - parsing
  - typechecking
  - register allocations

Focus :
- Imperative language
- Code generation , optimisation

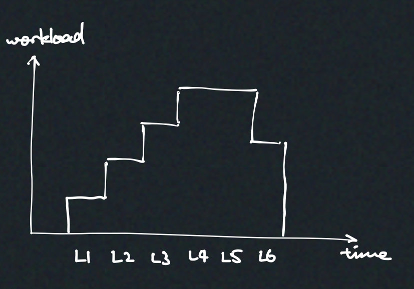
Not course content but useful :
- Software engineering
  - unit testing
  - writing specs
  - code revision
- Design
- Git

Logistics

- Go to lecture
- Grading

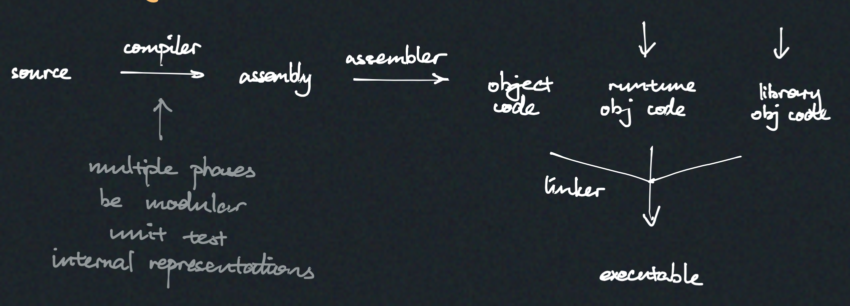
assignments	each	total
L1 - L4	100 pts	400
L3 code review	50 pts	50
L6 proposal	50 pts	50
L5 - L6	150 pts	300
Writtens 1-4	50 pts	200
Exams	NONE	

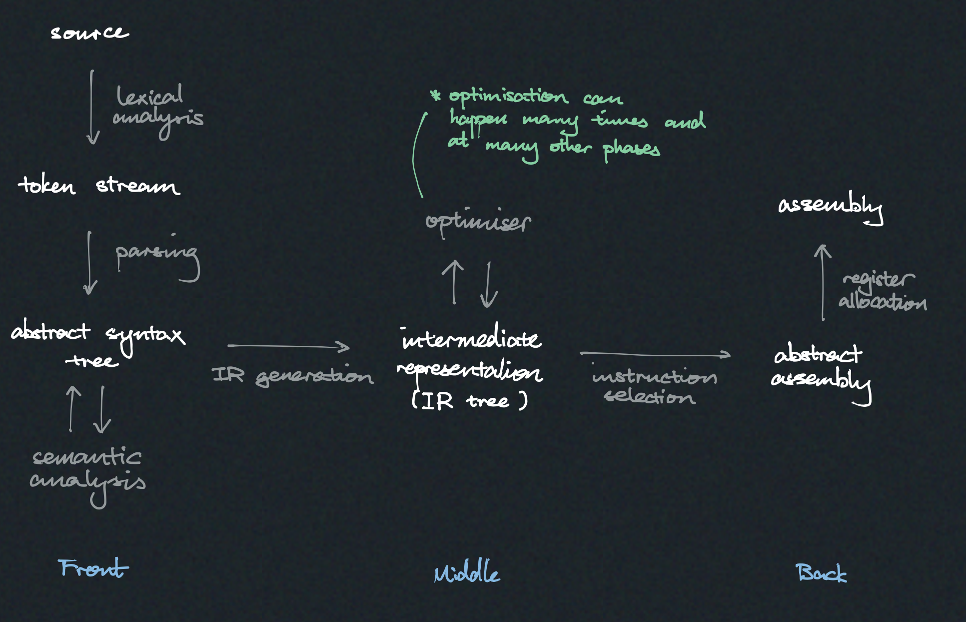
- Labs
  - L1 straight - line
  - L2 conditionals , loops
  - L3 functions
  - L4 memory
  - L5 optimisations] with report
  - L6 open]



- Source lang : C0
- Target lang : x86-64
  ... many instructions
  but move alone is Turing complete
  in fact CPU memory management is enough so 0 instruction sufficient to implement working compiler
- This is a partner class
- Language support : SML , Ocaml , Rust
- Programming
  - GitHub for code
  - Autolab for autograding
    - unlimited submissions
    - local tests available
  - Labs depend on previous labs
    bugs carry over
- Theme this sem : famous mathematian
  team names should fit the theme
- 6 late days total, max 2 days for each lab
- Rules
  - Do
    - Use debugger , profiler
    - Use standard library
    - ChatGPT for test case okay
  - Ask
    - Use library close to what we're implementing

Compilers high level





Instruction Selection

Input to this phase : IR tree ← assume valid programme
Output of .. : abstract assembly

Ex.

IR tree representing ... → abstract assembly

IR tree representing ...	abstract assembly
x = 5	x = 5
return ((x + x) + (x * 2))	t_1 = x + x
	t_2 = x * 2 ← not optimised (can simply return 20)
	ret = t_1 + t_2
	return

Def IR tree is some datastructure representing textual prog ramme

programme	p ::= s_1, ..., s_n	
statements	s ::= x = e	
	| return e	
expressions	e ::= c	int constant
	| x	variable
	| $e_1 \oplus e_2$	binary ops
operations	$\oplus$ ::= +	
	| *	
	| -	
	| /	
	| ...	

Def Abstract assembly

programme	p ::= i_1, ..., i_n	
instructions	i ::= d ← s	move
	| s ← $s_1 \oplus s_2$	binary ops
	| return	return
operands	d, s ::= r	register (finite)
	| c	integer constants
	| t	temporary (possibly infinite)
	| x	variable (usually treated as tmp)