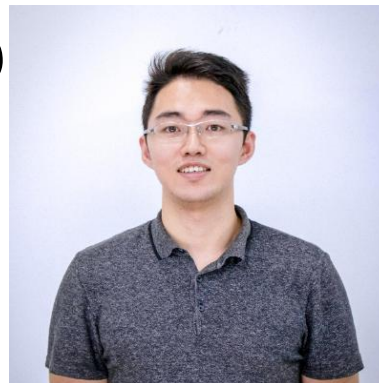


# Compilers

## Chapter 1: Introduction

- Lecturers:
  - Paul Kelly ([p.kelly@imperial.ac.uk](mailto:p.kelly@imperial.ac.uk))
  - Jamie Willis ([j.willis@imperial.ac.uk](mailto:j.willis@imperial.ac.uk))
  - Hongxiang Fan ([hongxiang.fan@imperial.ac.uk](mailto:hongxiang.fan@imperial.ac.uk))



- This course is about a particular class of programs called *language processors*, of which the best example is a compiler.

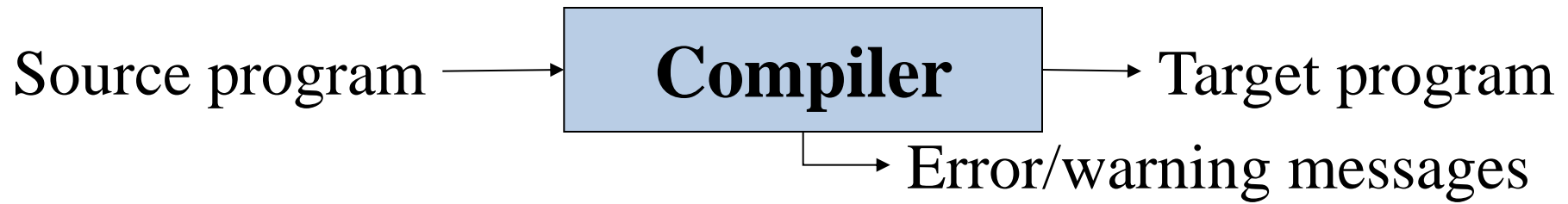
## What is a compiler?

- A program which *processes* programs, written in some programming language.
- A program which *writes* programs (in some language).
- A compiler *translates* programs written in one language into “equivalent” programs in another language.

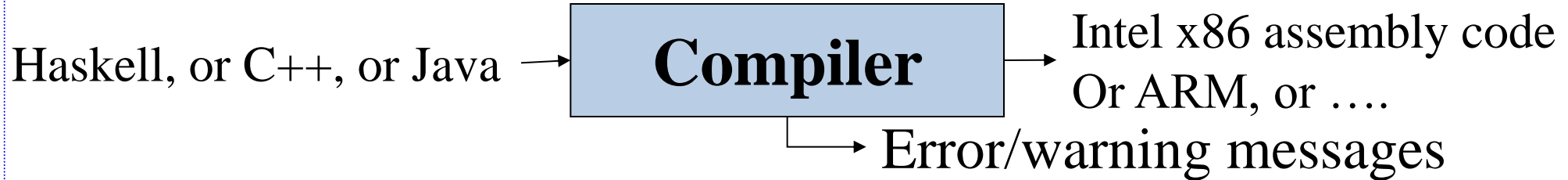
- This course is about a particular class of programs called *language processors*, of which the best example is a compiler.

## What is a compiler?

- A program which *processes* programs, written in some programming language.
  - A program which *writes* programs (in some language).
  - A compiler *translates* programs written in one language into “equivalent” programs in another language
- A tool to *enable you to program at a higher level*, by mapping high-level concepts to low-level implementation



*For example:*



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

# Basic compiler structure

**Input**

In some language

Eg C, Java, C#

**Analysis**

Construct an internal representation of the source language structure, and hence its meaning

Usually start by building a tree representation, but may build graph, eg to represent control flow

**Synthesis**

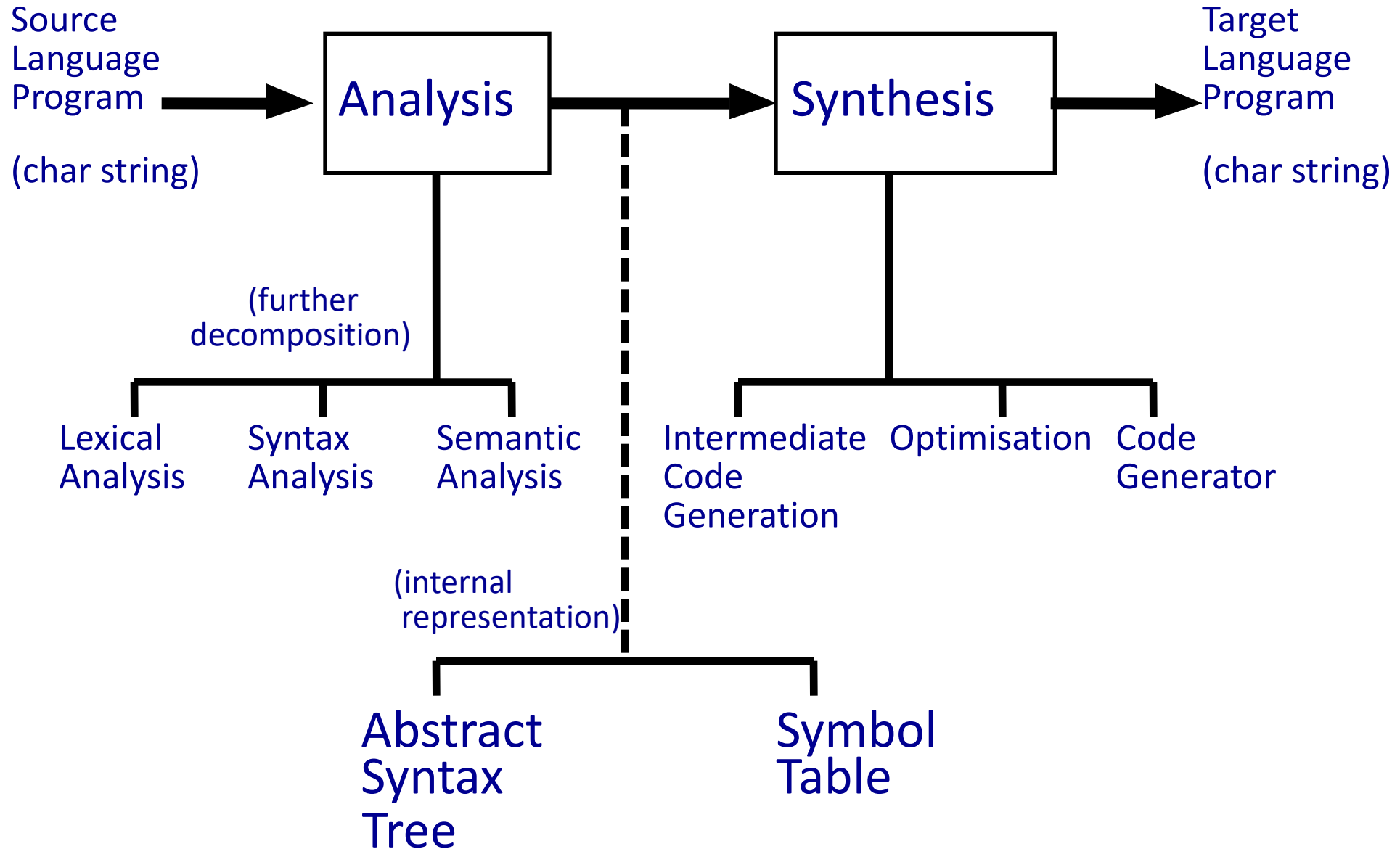
Use this internal representation to construct target language version

Analyse and transform the internal representation, then traverse it to produce output

**Output**

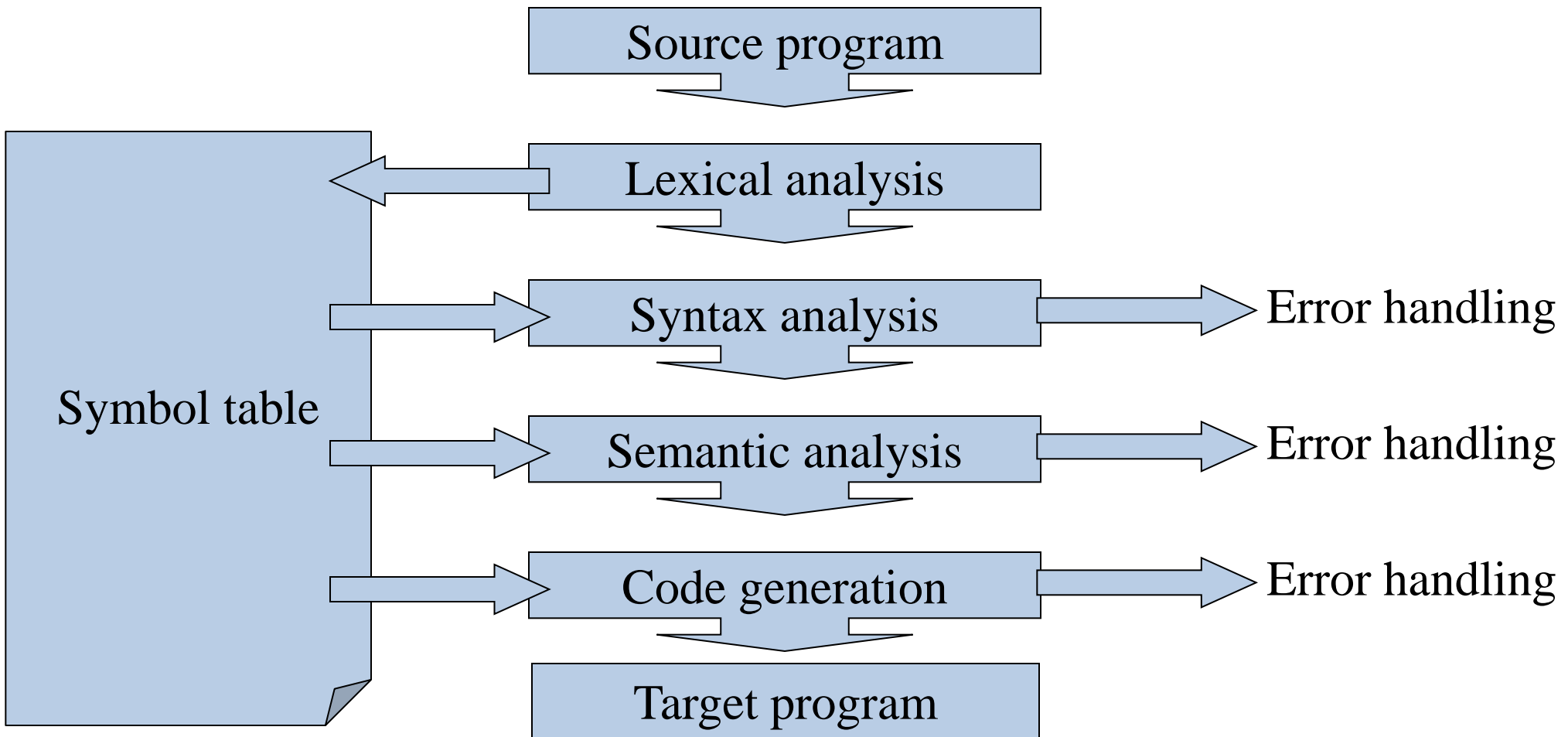
In the target language

Eg in Intel x86 assembler



# Compiler structure in more detail

# The phases of a compiler



Information from declarations is gathered in the symbol table, and is used to check how each variable is used, to reserve memory for it, and to generate code to access it

# Phases of a compiler - example

- *Input file “test.c”:*

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

- *Output file “test.s”:*

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

- Command: “gcc -S -O test.c”  
(the flag “-S” tells the compiler to produce assembly code, “-O” turns optimisation on).



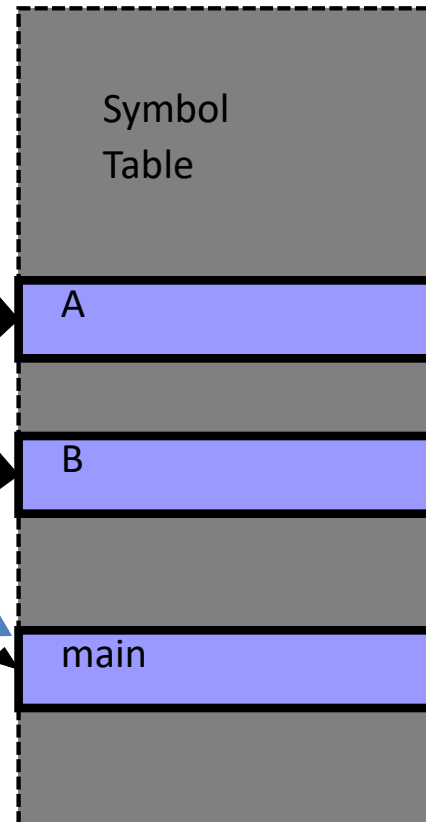
# Introduction to lexical analysis

- INPUT: sequence of characters

```
intspA; nlintspB; nlmain ( ) nl{ nlAsp=spB+123 ; nl} nl
```

- OUTPUT: sequence of tokens:

INTtok  
IDENTtok ( )  
SEMICOLONtok  
INTtok  
IDENTtok ( )  
SEMICOLONtok  
IDENTtok ( )  
LBRACKETtok  
RBRACKETtok  
LCURLYtok  
IDENTtok ( )  
EQtok  
IDENTtok ( )  
PLUSTok  
CONSTtok 123  
SEMICOLONtok  
RCURLYtok



User identifiers like A, B and main are all represented by the same lexical token (**IDENTtok**), which includes a pointer to a symbol table record giving the actual name.

# Introduction to Syntax Analysis (also known as “parsing”)

- Programming languages have grammatical structure specified by grammatical rules in a notation such as BNF (Backus-Naur Form)

- Example:

```
stat → 'print' expression |  
      'while' expression 'do' stat |  
      expression '=' expression
```

- The function of syntax analysis is to extract the grammatical structure—to work out how the BNF rules must have been applied to yield the input program.
- The output of the syntax analyser is a data structure representing the program structure: the **Abstract Syntax Tree** (AST).

# Returning to our C example:

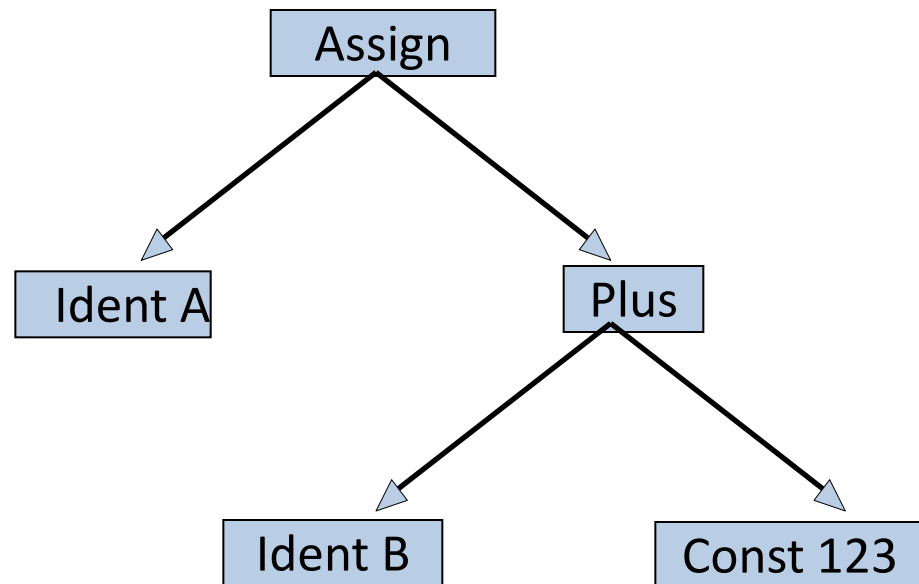
- Input characters:

“.....A = B+123.....”

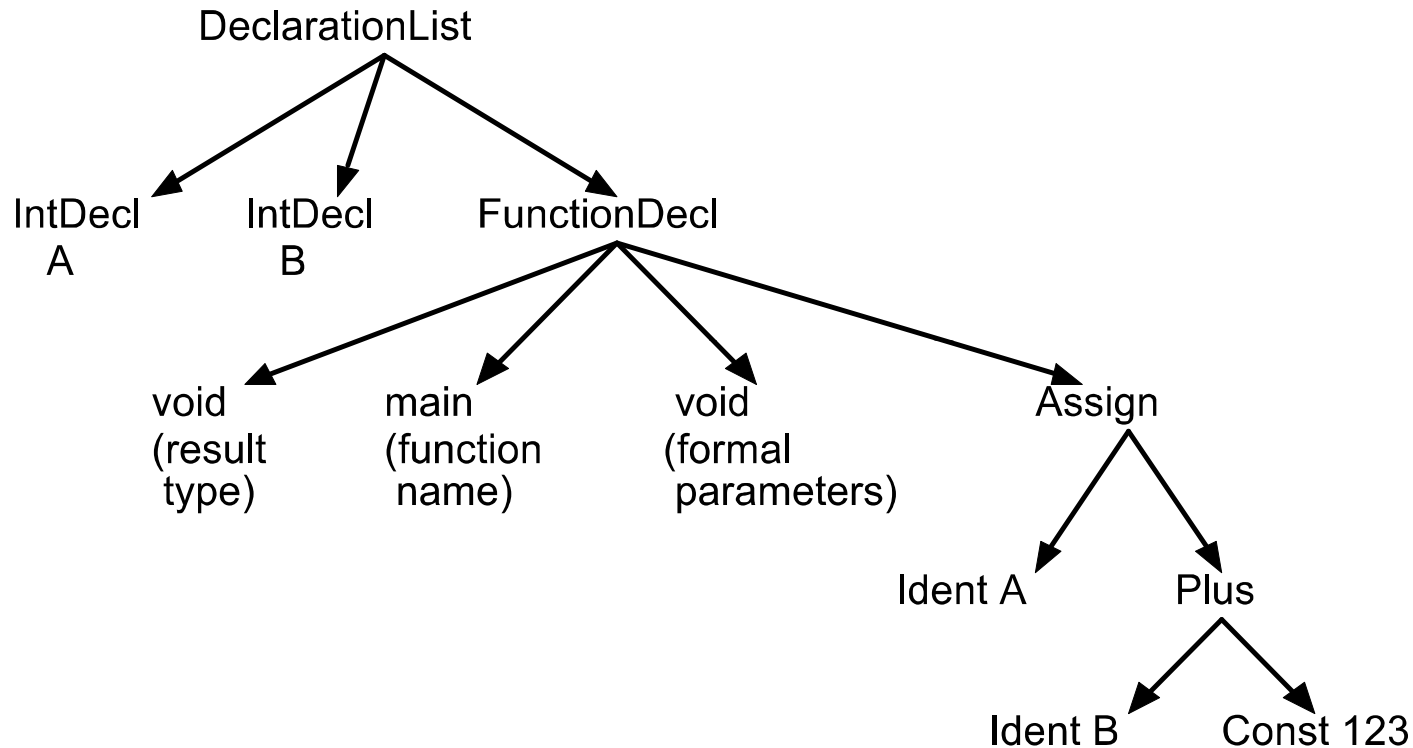
- Lexical tokens:

[IDENTtok A, EQtok, IDENTtok B,  
PLUStok, CONSTtok 123]

- Abstract syntax tree:



# AST for whole C example:



- The AST is implemented as a tree of linked objects
- The compiler writer must design the AST data structure carefully so that it is easy to build (during syntax analysis), and easy to use (e.g. during code generation).

# You try: experimenting with real compilers

- Create a file “file.c” containing a simple C function
- Under Linux, type the command:

```
gcc -O -S file.c
```

- This tells Gnu C compiler ‘gcc’ to produce optimised translation of the C program and leave result in “file.s”
- Examine “file.s”
- You might try

```
gcc -O -S -fomit-frame-pointer file.c
```

(This simplifies the output slightly)

- On Windows using Microsoft Visual Studio, try

```
cl /Fa /Ox /c test.c
```

(The output is written to “test.asm”)

- **Better still: <http://gcc.godbolt.org/>**

Compiler Explorer

https://godbolt.org

COMPILER EXPLORER

Add... More

Benchmark your code online at [Quick Bench!](#)

Sponsors [intel](#) [PC-lint](#) [Solid Sands](#) Share Other Policies

C++ source #1

x86-64 clang 11.0.0 (Editor #1, Compiler #1) C++

x86-64 clang 11.0.0 Compiler options...

```
1 int A;
2 int B;
3 void test_fun()
4 {
5     A = B+123;
6 }
```

```
1 test_fun():                                # @test_fun()
2     push    rbp
3     mov     rbp, rsp
4     mov     eax, dword ptr [B]
5     add     eax, 123
6     mov     dword ptr [A], eax
7     pop     rbp
8     ret
9
10 A:
11     .long   0                                # 0x0
12
13 B:
14     .long   0                                # 0x0
```

Output (0/0) x86-64 clang 11.0.0 - cached (8512B)

Clang compiler, x86, Optimisation level zero (i.e. “-O0”)

The screenshot shows the Compiler Explorer interface. On the left, the C++ source code is displayed: `1 int A[1024];`, `2 int B[1024];`, `3 void test_fun()`, `4 {`, `5 for (int i=0; i<1024; ++i)`, `6 A[i] = B[i]+123;`, `7 }`. The main window shows the assembly output for x86-64 clang 11.0.0. The assembly code is as follows: `1 test_fun(): # @test_fun()`, `2 push rbp`, `3 mov rbp, rsp`, `4 mov dword ptr [rbp - 4], 0`, `5 .LBB0_1: # =>This Inner Loop Header: Depth=1`, `6 cmp dword ptr [rbp - 4], 1024`, `7 jge .LBB0_4`, `8 movsxd rax, dword ptr [rbp - 4]`, `9 mov ecx, dword ptr [4*rax + B]`, `10 add ecx, 123`, `11 movsxd rax, dword ptr [rbp - 4]`, `12 mov dword ptr [4*rax + A], ecx`, `13 mov eax, dword ptr [rbp - 4]`, `14 add eax, 1`, `15 mov dword ptr [rbp - 4], eax`, `16 jmp .LBB0_1`, `17 .LBB0_4:`, `18 pop rbp`, `19 ret`, `20 A:`, `21 .zero 4096`, `22`, `23 B:`, `24 .zero 4096`. The bottom status bar indicates 'Output (0/0) x86-64 clang 11.0.0 - cached (135468)'.

Clang compiler, x86, Optimisation level zero (i.e. “-O0”)

January 25 (<https://godbolt.org/z/1h1drv9Y3> )

Compiler Explorer

https://godbolt.org

COMPILER EXPLORER

Add... More

Get cool gear in the Compiler Explorer shop

Sponsors intel PC-lint Solid Sands

Share Other Policies

C++ source #1

x86-64 clang 11.0.0 (Editor #1, Compiler #1) C++

x86-64 clang 11.0.0 Compiler options...

```
1 int A[1024];
2 int B[1024];
3 void test_fun()
4 {
5     for (int i=0; i<1024; ++i)
6         A[i] = B[i]+123;
7 }
```

```
1 test_fun(): # @test_fun()
2     push    rbp
3     mov     rbp, rsp
4     mov     dword ptr [rbp - 4], 0 # "i" is stored on stack at frame offset -4
5     .LBB0_1: # =>This Inner Loop Header: Depth=1
6     cmp     dword ptr [rbp - 4], 1024 # Check if i exceeds 1024
7     jge     .LBB0_4
8     movsxd  rax, dword ptr [rbp - 4] # Load B[i] (each int is 4 bytes long)
9     mov     ecx, dword ptr [4*rax + B] # Add 123!
10    add     ecx, 123
11    movsxd  rax, dword ptr [rbp - 4] # Store result to A[i]
12    mov     dword ptr [4*rax + A], ecx # Increment i
13    mov     eax, dword ptr [rbp - 4]
14    add     eax, 1
15    mov     dword ptr [rbp - 4], eax
16    jmp     .LBB0_1
17 .LBB0_4:
18    pop     rbp
19    ret
20 A:
21     .zero   4096 # Reserve space for array A (and associate symbol "A" to the start address)
22
23 B:
24     .zero   4096
```

Output (0/0) x86-64 clang 11.0.0 - cached (135468)

Clang compiler, x86, Optimisation level zero (i.e. "-O0")



Compiler Explorer

https://godbolt.org

COMPILER EXPLORER

Add... More

Get cool gear in the [Compiler Explorer shop](#)

Sponsors [intel](#) [PC-lint](#) [Solid Sands](#) Share Other Policies

C++ source #1

```
1 int A[1024];
2 int B[1024];
3 void test_fun()
4 {
5     for (int i=0; i<1024; ++i)
6         A[i] = B[i]+123;
7 }
```

x86-64 clang 11.0.0 (Editor #1, Compiler #1) C++

x86-64 clang 11.0.0 -O1

Output... Filter... Libraries Add new... Add tool...

```
1 test_fun(): # @test_fun()
2     mov     rax, -4096
3 .LBB0_1:    # =>This Inner Loop Header: Depth=1
4     mov     ecx, dword ptr [rax + B+4096]
5     add     ecx, 123
6     mov     dword ptr [rax + A+4096], ecx
7     add     rax, 4
8     jne     .LBB0_1
9     ret
10 A:
11     .zero   4096
12
13 B:
14     .zero   4096
```

Output (0/0) x86-64 clang 11.0.0 - 563ms (12959B)

Clang compiler, x86, Optimisation level “-O1”

Compiler Explorer

https://godbolt.org

COMPILER EXPLORER

Get cool gear in the Compiler Explorer shop

Sponsors intel.PC-lint Solid Sands

Share Other Policies

C++ source #1

```
1 int A[1024];
2 int B[1024];
3 void test_fun()
4 {
5     for (int i=0; i<1024; ++i)
6         A[i] = B[i]+123;
7 }
```

x86-64 clang 11.0.0 (Editor #1, Compiler #1) C++

x86-64 clang 11.0.0 -O1

Output... Filter... Libraries + Add new... Add tool...

```
1 test_fun(): # @test_fun()
2     mov     rax, -4096
3 .LBB0_1:    # =>This Inner Loop Header: Depth=1
4     mov     ecx, dword ptr [rax + B+4096]
5     add     ecx, 123
6     mov     dword ptr [rax + A+4096], ecx
7     add     rax, 4
8     jne     .LBB0_1
9     ret
10 A:
11     .zero   4096
12
13 B:
14     .zero   4096
```

"i" is stored in register rax, except we store i-4096 so we can compare to zero and avoid the cmp instruction

Output (0/0) x86-64 clang 11.0.0 - 563ms (12959B)

Clang compiler, x86, Optimisation level "-O1"

January 25 (<https://godbolt.org/z/W5h4a9P3T>)

The screenshot displays the Compiler Explorer interface. On the left, the C++ source code is shown with line numbers 1 through 7. The code defines two arrays, A and B, each of size 1024, and a function test\_fun() that iterates over both arrays, adding the value of B[i] to A[i] plus 123. The right pane shows the assembly output for x86-64 clang 11.0.0 at optimization level -O2. The assembly includes labels for the loop header (.LCPI0\_0), the inner loop body (.LBB0\_1), and the end of the function (.LBB0\_1). The assembly code uses x86-64 instructions like mov, movdqa, padd, and add to implement the loop logic. The output also shows the memory layout for arrays A and B, with A starting at address 4096 and B starting at address 4144.

```
1 int A[1024];
2 int B[1024];
3 void test_fun()
4 {
5     for (int i=0; i<1024; ++i)
6         A[i] = B[i]+123;
7 }
```

```
1 .LCPI0_0:
2     .long    123                # 0x7b
3     .long    123                # 0x7b
4     .long    123                # 0x7b
5     .long    123                # 0x7b
6 test_fun():                     # @test_fun()
7     mov     rax, -4096
8     movdqa  xmm0, xmmword ptr [rip + .LCPI0_0] # xmm0 = [123,123,123,123]
9 .LBB0_1:                         # =>This Inner Loop Header: Depth=1
10    movdqa  xmm1, xmmword ptr [rax + B+4096]
11    padd    xmm1, xmm0
12    movdqa  xmm2, xmmword ptr [rax + B+4112]
13    padd    xmm2, xmm0
14    movdqa  xmmword ptr [rax + A+4096], xmm1
15    movdqa  xmmword ptr [rax + A+4112], xmm2
16    movdqa  xmm1, xmmword ptr [rax + B+4128]
17    padd    xmm1, xmm0
18    movdqa  xmm2, xmmword ptr [rax + B+4144]
19    padd    xmm2, xmm0
20    movdqa  xmmword ptr [rax + A+4128], xmm1
21    movdqa  xmmword ptr [rax + A+4144], xmm2
22    add     rax, 64
23    jne     .LBB0_1
24    ret
25 A:
26     .zero   4096
27
28 B:
```

Clang compiler, x86, Optimisation level “-O2”

January 25 (<https://godbolt.org/z/Y8jaGEW6b>)

Compiler Explorer

Watch C++ Weekly to learn new C++ features

Sponsors intel.PC-lint Solid Sands

Share Other Policies

C++ source #1

x86-64 clang 11.0.0 (Editor #1, Compiler #1) C++

x86-64 clang 11.0.0 -O2

```

1 int A[1024];
2 int B[1024];
3 void test_fun()
4 {
5     for (int i=0; i<1024; ++i)
6         A[i] = B[i]+123;
7 }

```

```

1 .LCPI0_0:
2     .long    123                # 0x7b
3     .long    123                # 0x7b
4     .long    123                # 0x7b
5     .long    123                # 0x7b
6 test_fun():                     # @test_fun()
7     mov     rax, -4096
8     movdqa  xmm0, xmmword ptr [rip + .LCPI0_0] # xmm0 = [123,123,123,123]
9 .LBB0_1:                         # =>This Inner Loop Header: Depth=1
10    movdqa  xmm1, xmmword ptr [rax + B+4096]
11    padd    xmm1, xmm0           # We add 123 to all 4 values
12    movdqa  xmm2, xmmword ptr [rax + B+4112]
13    padd    xmm2, xmm0
14    movdqa  xmmword ptr [rax + A+4096], xmm1 # We store the result to A
15    movdqa  xmmword ptr [rax + A+4112], xmm2
16    movdqa  xmm1, xmmword ptr [rax + B+4128]
17    padd    xmm1, xmm0
18    movdqa  xmm2, xmmword ptr [rax + B+4144]
19    padd    xmm2, xmm0
20    movdqa  xmmword ptr [rax + A+4128], xmm1
21    movdqa  xmmword ptr [rax + A+4144], xmm2
22    add     rax, 64              # The loop counter is incremented in steps of 64 = 4x4x4;
23    jne     .LBB0_1
24    ret
25 A:
26     .zero   4096
27
28 B:

```

We compute with four values in each register

We load 4 values from B into xmm1

We add 123 to all 4 values

We store the result to A

The loop is unrolled 4 times. The compiler schedules the instructions using a model of the processor pipeline

The loop counter is incremented in steps of 64 = 4x4x4;

- Four bytes per int
- Four ints per xmm
- Unrolled four times

Output (0/0) x86-64 clang 11.0.0 - 605ms (140488)

Clang compiler, x86, Optimisation level “-O2”

Compiler Explorer

Look after yourself, and, if you can, someone else too

Sponsors intel.PC-lint Solid

Compiler Explorer

C++ source #1

```
1 int A[1024];
2 int B[1024];
3 void test_fun()
4 {
5     for (int i=0; i<1024; ++i)
6         A[i] = B[i]+123;
7 }
```

x86-64 gcc 21.1.9 (Editor #1, Compiler #1) C++

x86-64 gcc 21.1.9 -fast

```
1 test_fun():
2     xor     eax, eax                                #5.5
3     vmovdqu ymm0, YMMWORD PTR .L_2il0floatpacket.0[rip] #6.21
4     ..B1.2:                                         # Preds ..B1.2 ..B1.1
5     vpaddd  ymm1, ymm0, YMMWORD PTR [B+rax*4]      #6.21
6     vpaddd  ymm2, ymm0, YMMWORD PTR [32+B+rax*4]   #6.21
7     vmovdqu YMMWORD PTR [A+rax*4], ymm1           #6.9
8     vmovdqu YMMWORD PTR [32+A+rax*4], ymm2         #6.9
9     add     rax, 16                                #5.5
10    cmp     rax, 1024                              #5.5
11    jb     ..B1.2                                   # Prob 99% #5.5
12    vzeroupper                                     #7.1
13    ret                                             #7.1
14 A:
15 B:
16 .L_2il0floatpacket.0:
17     .long   0x0000007b,0x0000007b,0x0000007b,0x0000007b,0x0000007b,0x0000007b,0x0000007b,0x0000007b
```

The Intel compiler uses wider vector instructions – 8 ints per ymm register

But only unrolls by a factor of 2

(0x7b = 123)

Output (0/0) x86-64 gcc 21.1.9 - 1623ms (60450B)

Intel compiler, x86, Optimisation level “-fast”

January 25 (<https://godbolt.org/z/nxqGs7Pdf>)

# Compilers are just one kind of language processor:

- Really useful software tools are useful because they are programmable
- If it's programmable it must have some kind of programming language
- Programming languages are often “domain-specific” – designed for a particular application area

- Domain-specific languages – examples:

- Tensorflow: deep learning
- P4: network packet forwarding
- Solidity: smart contracts
- GLSL: shaders for 3D graphics
- SQL: database queries
- Verilog: digital circuit design
- Matlab: prototyping numerical computations
- Simulink: modelling dynamical systems
- R: statistical data analytics
- Prolog: logic programming
- LaTeX: typesetting
- The FEniCS Project's Unified Form Language: solving PDEs
- ANTLR, Yacc: parser generation
- TableGen: the LLVM compiler's DSL for instruction selection

- Language processors- examples:

- FindBugs: finds Java bugs
- PyLint: bug and quality checking for Python
- Coverity, CodeSonar: find C++ (etc) bugs
- BitBlaze, Coverity/B: vulnerability analysis for binaries
- KLEE: symbolic execution engine
- JUnit: annotation-driven unit testing
- Mockito: mock object generation for test
- IDEs – Intellisense in Visual Studio, etc
- Binary-to-binary: eg x86 to ARM for Windows-on-ARM
- Valgrind, PIN, Mambo: dynamic binary rewriting

# Course structure

- Introduction to syntax analysis – and a complete toy compiler
  - Code generation
  - Generating better code by using registers
  - Register allocation
  - Optimisation and data flow analysis
  - Loop optimisations
- Syntax analysis (sometimes called “parsing”)
  - Semantic analysis: is the program “legal”?
  - Runtime (memory) organisation: stacks and heaps



Paul



Jamie and Hongxiang

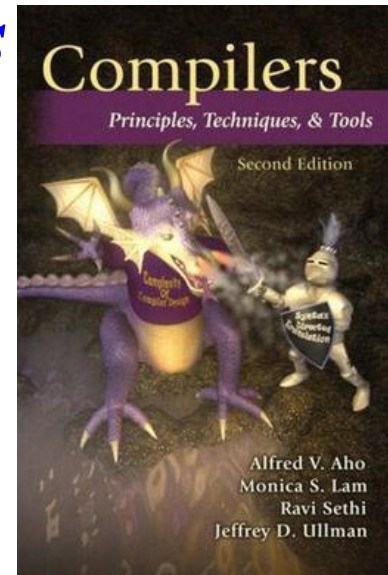
# Textbook - philosophy

- There are many textbooks on compilers, some good
- The purpose of lecture course is to give you enough understanding of the area to be able to use a serious textbook effectively
- Textbook should be worth more to you *after* the course than during it!
- Choose an authoritative book that goes substantially beyond this course

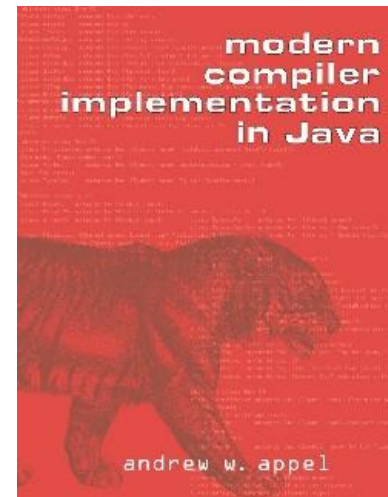


# Somewhat recommended textbooks

- *Compilers: Principles, Techniques, and Tools (second edition, 2006)* by Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Monica Lam.
  - The new(ish) edition of the definitive book by pioneers in the subject. Often called the **Dragon book** because of the picture on the front. Compiler engineers regularly refer to “standard Dragon book stuff”.
- *Modern Compiler Implementation in Java (second edition, 2005)* by Andrew Appel
  - Useful source for specific advice on how to build your compiler, including simple and more sophisticated techniques



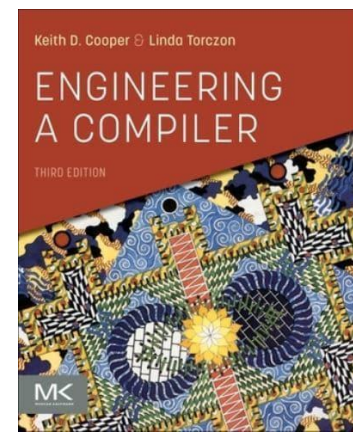
[https://en.wikipedia.org/wiki/Compilers:\\_Principles,\\_Techniques,\\_and\\_Tools](https://en.wikipedia.org/wiki/Compilers:_Principles,_Techniques,_and_Tools)



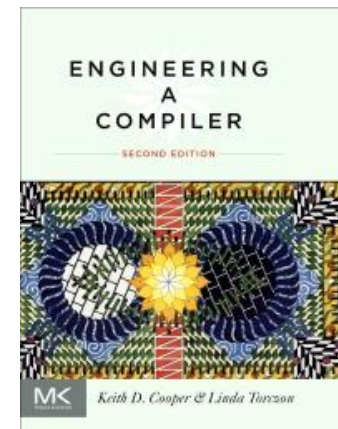
<https://www.cs.princeton.edu/~appel/modern/java/>

# The most recommended textbook

- *Engineering a compiler*, by Keith Cooper and Linda Torczon, Morgan Kaufmann/Elsevier (3<sup>rd</sup> edition, 2022). About £75.
  - “This book is the best compiler engineering guide ever I read.” (review, Amazon.com)
  - “This book has a good introduction guiding the beginning compiler student into understanding basic concepts and gradually revealing the more intimidating stuff, but the authors took great care not to scare the beginners away and instead offers great indepth explanations into how concepts and implementation merge. Its an overall good book!” (review, Amazon.com)
  - “*First, all the algorithms are consistent with the latest research. Second, the explanations are exceptionally clear, especially compared to other recent books. Third, there's always enough extra context presented so that you understand the choices you have to make, and understand how those choices fit with the structure of your whole compiler*”. (review, Jeff Kenton, comp.compilers 3/12/03)
  - “If you are a beginner "do not buy this book”” (review, Amazon.com)



<https://www.elsevier.com/books/engineering-a-compiler/cooper/978-0-12-815412-0>



(The 2<sup>nd</sup> edition is actually adequate for this course)

# How to enjoy, learn from and pass this course:

- **Textbook** – start early, read the first couple of chapters
- Make **notes** during lectures
- **Tutorial exercises** are used to introduce new examinable material
- Tutorials are designed to reinforce and integrate lecture material; it's designed to help you pass the exam
- Go look at the **past papers** - *now*
- Use the **live Q&A classes** to get feedback on your solutions
- You are **assumed to have studied the past exam papers**
- Substantial lab exercise should bring it all together
- Ask questions! Use **EdStem!**
- There are hundreds of compilers courses at universities around the world, often more advanced than this one – with slides on the web that will help with your questions

# Beyond the curriculum....

- **The first compilers (for Fortran, COBOL etc) were designed to put programmers out of work**
- **But that's not what happened**
- **Why?**

- **See Jevons Paradox [https://en.wikipedia.org/wiki/Jevons\\_paradox](https://en.wikipedia.org/wiki/Jevons_paradox)**

# Beyond the curriculum....

- **LLMs can generate code**
- **LLM agents can automate business logic without code**
- **Will AI put programmers out of work?**

# Beyond the curriculum....

- **Programs can have type errors**
- **A good type system should prevent type errors**
  - **In a language with strong, static type checking, we should have a guarantee that no type error can occur at runtime**
- **You can write a compiler in a language with strong, static type checking**
- **That is, you can write a program that generates code**
- **Can we guarantee that the code we generate contains no type errors?**