Compilers and computer architecture
1. Introduction

Ian Mackie

Email: **i.mackie@sussex.ac.uk**
Office hours: by appointment

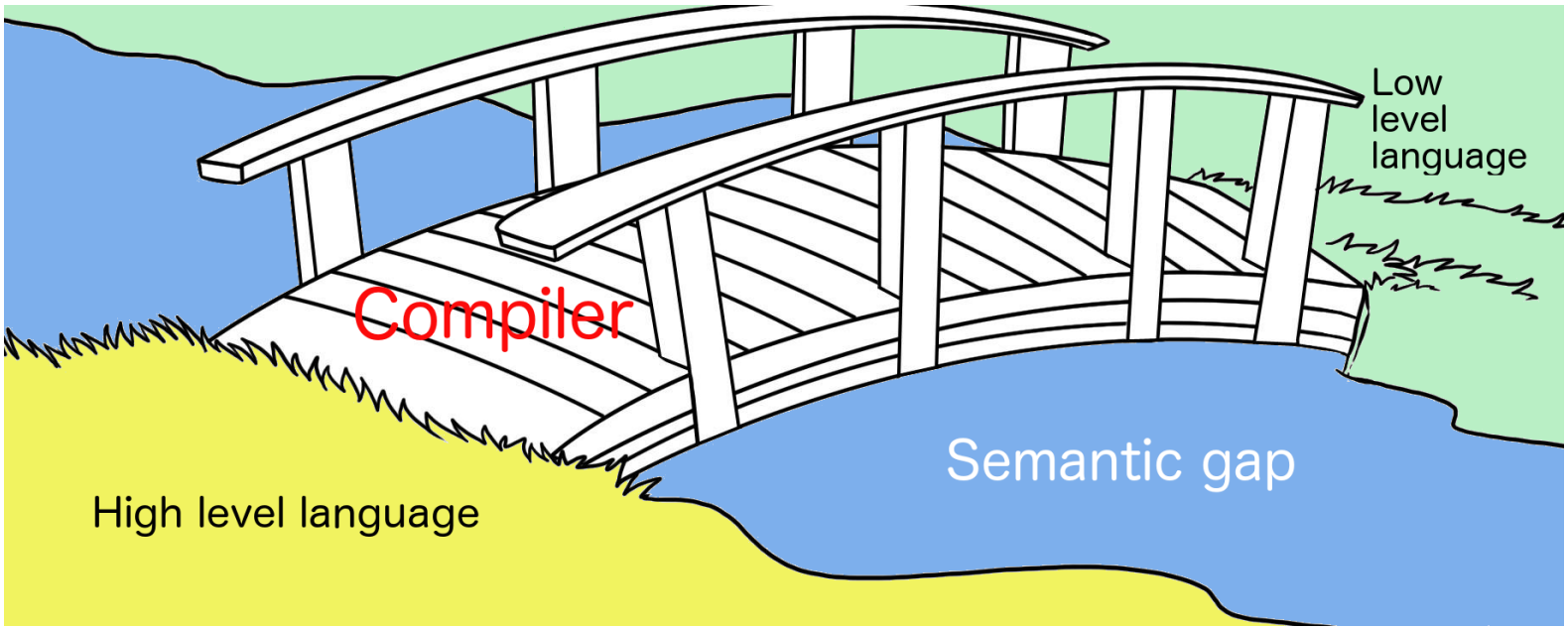
September 2022

Overview

- What is a compiler?
- Why study compilers?
- Compiler structure:
  - Lexical analysis
  - Syntax analysis
  - Semantic analysis, type-checking
  - Code synthesis

Overview

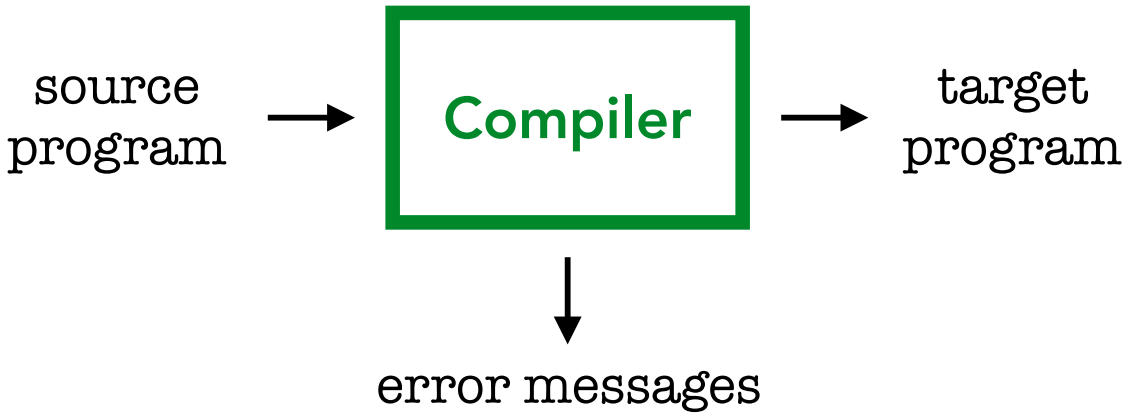
What is a compiler?



A compiler translates programs from one programming language to programs in another programming language — such as machine language — while preserving the meaning.

(What does “preserve” and “meaning” mean in this context?)

Overview
What is a compiler?



Typically, the input language (or *source language*) is more high-level than the output language (or *target language*)

Examples:

- **Source:** Java, **Target:** JVM bytecode.
- **Source:** JVM bytecode, **Target:** ARM/x86 machine code
- **Source:** TensorFlow, **Target:** GPU/TPU machine code.

Example translation: source program

Here is a little program. (What does it do?)

```
int testfun (int n) {
  int res = 1;
  while (n > 0) {
      n--;
      res *= 2;
  }
  return res;
}
```

Using **`clang -S`** this translates to the following x86 machine code ...

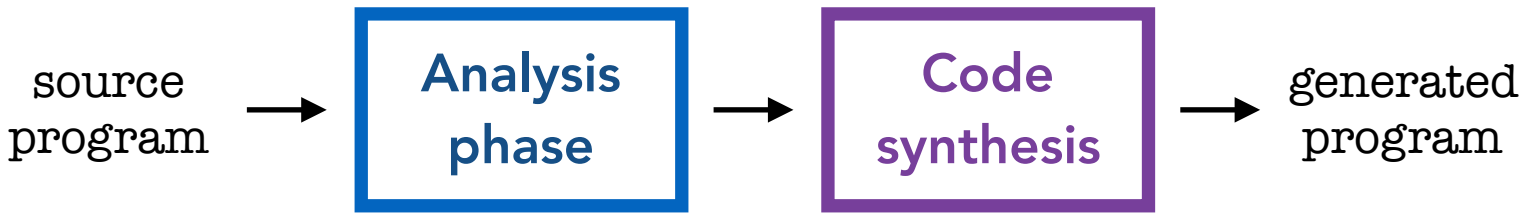
Example translation: target program

```
_testfun:                                       ## @testfun
        .cfi_startproc

                                                ## BB#0:
        pushq   %rbp
Ltmp0:
        .cfi_def_cfa_offset 16
Ltmp1:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
Ltmp2:
        .cfi_def_cfa_register %rbp
        movl    %edi, -4(%rbp)
        movl    $1, -8(%rbp)
LBB0_1:                                         ## =>This Inner Loop Header: Depth=1
        cmpl    $0, -4(%rbp)
        jle     LBB0_3


                                                ## BB#2:
                                                ##   in Loop: Header=BB0_1 Depth=1
        movl    -4(%rbp), %eax
        addl    $4294967295, %eax               ## imm = 0xFFFFFFFF
        movl    %eax, -4(%rbp)
        movl    -8(%rbp), %eax
        shll    $1, %eax
        movl    %eax, -8(%rbp)
        jmp     LBB0_1
LBB0_3:
        movl    -8(%rbp), %eax
        popq    %rbp
        retq
        .cfi_endproc
```

Why study compilers?

- To be a good programmer, you need to understand what happens ‘under the hood’ when you write programs in a high-level language.
- To better understand low-level languages — such as assembler, C/C++, Rust, and Go (of prime importance, *e.g.* for operating systems, embedded code, and generally code that needs to be fast).
- Many large programs (*e.g.*, Office) embed a programming language. The skill quickly to write an interpreter or compiler for such embedded languages is invaluable.
- But also: compilers are one of the all time great examples of software engineering (modularisation).

The structure of a compiler

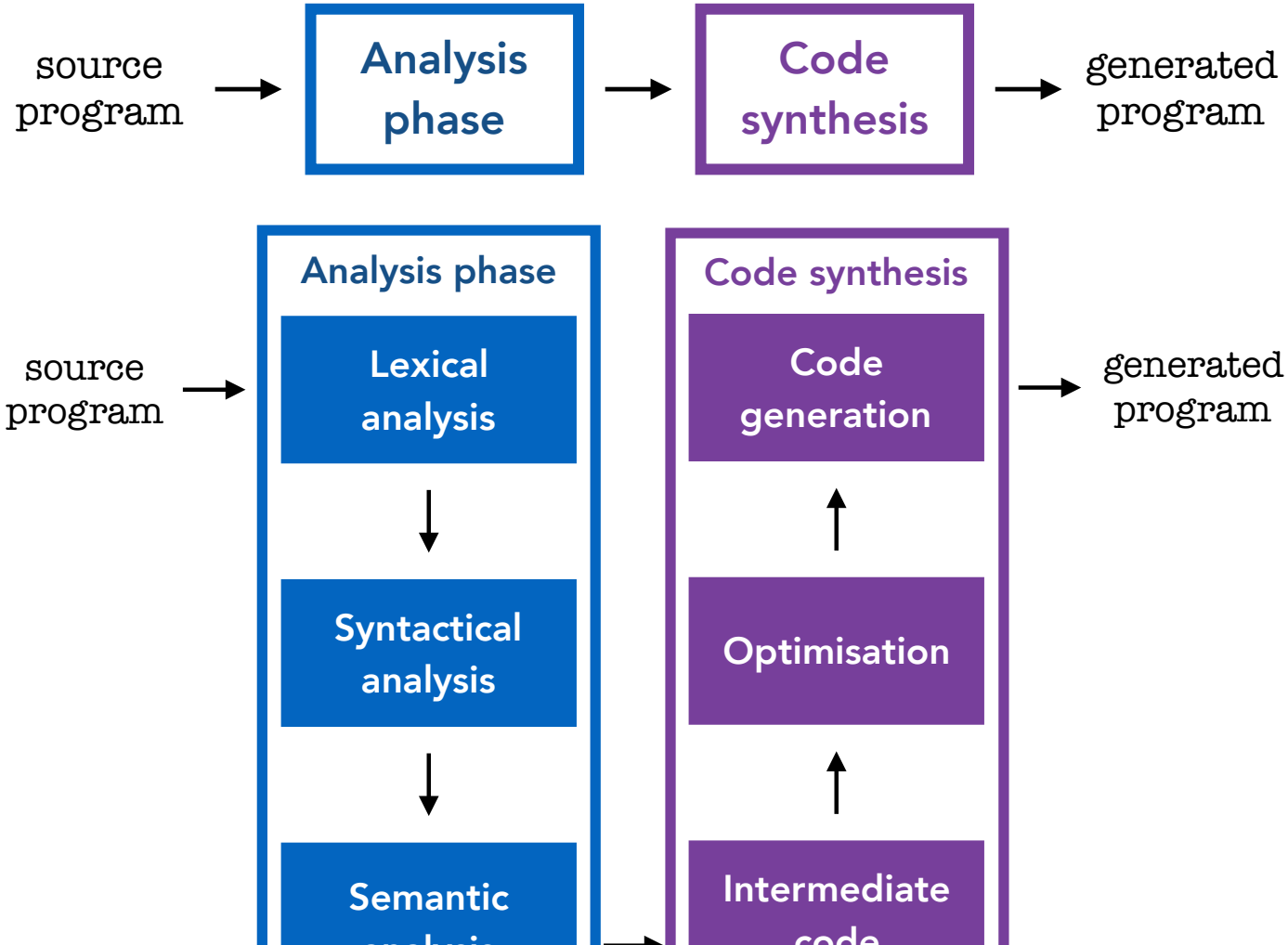


- In the analysis phase two things happen:
  - Analysing if the program is well-formed (*e.g.*, checking for syntax and type errors).
  - Creating an efficient representation of the source program structure — an **abstract syntax tree** (AST), and symbol table — for further processing.
- The code synthesis phase then produces the executable program from the AST.

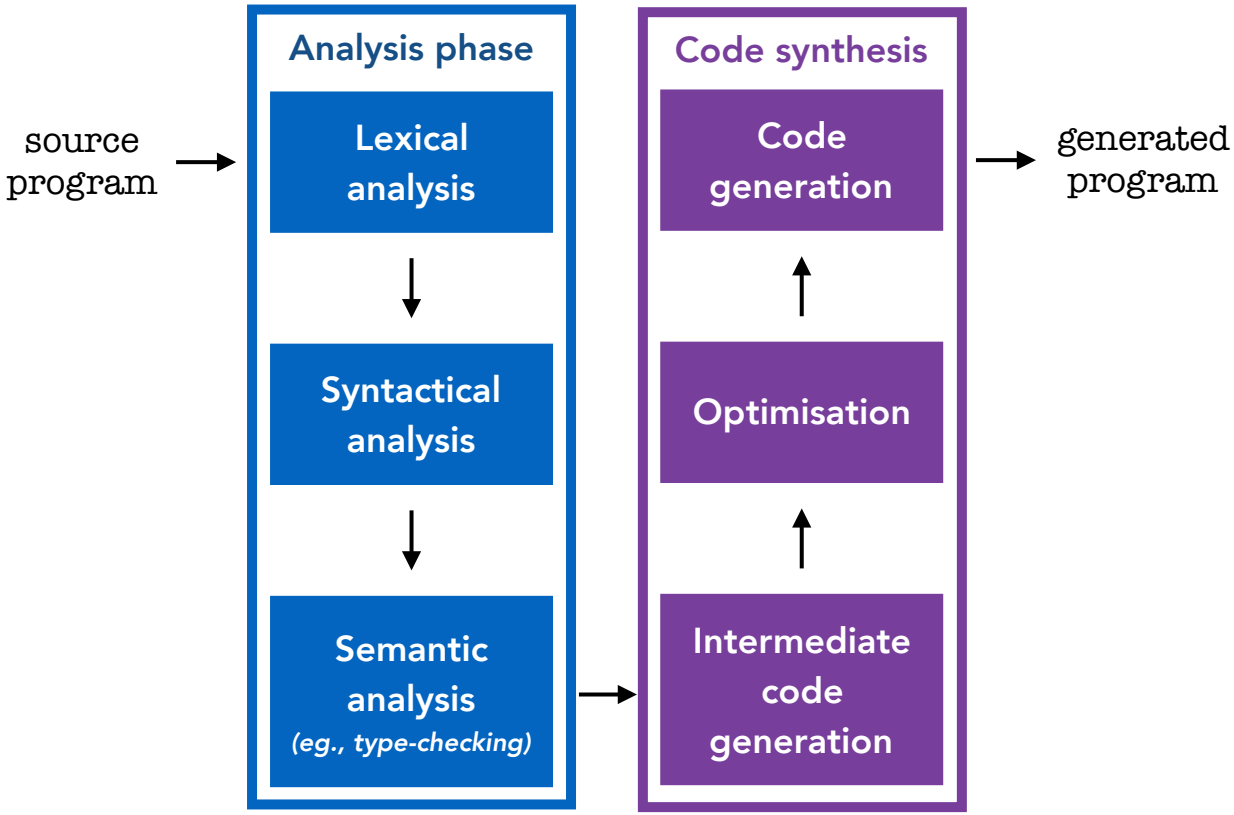
The structure of a compiler

Refining the picture

A compiler actually breaks the hard problem of compilation into several smaller problems, making it easier to retarget compilers (changing source or target language).



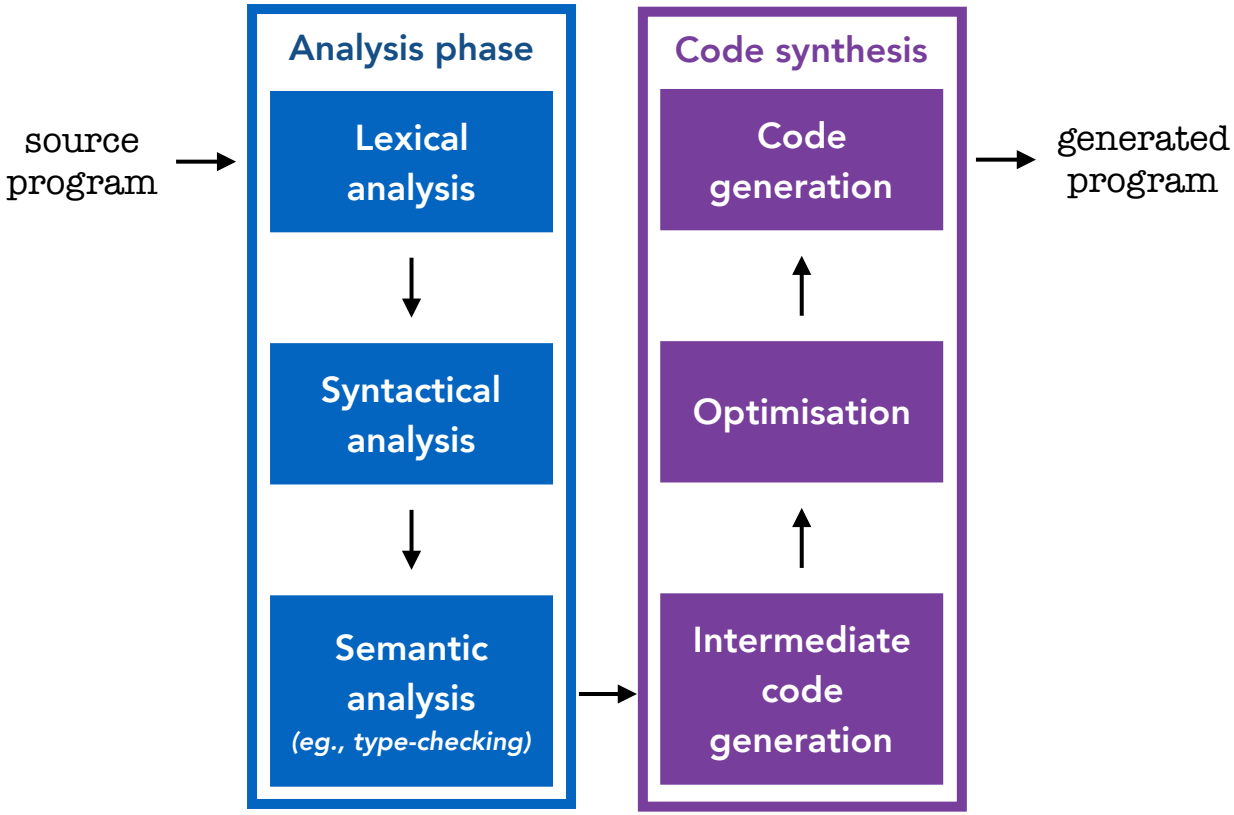
Compilation phases



Question: when do these phases happen?

- In the past: all happend at ... compile-time.
- Now: some happen at run-time, in Just-in-time compilers (JITs) — affecting choice of algorithms, and performance.

Compilation phases

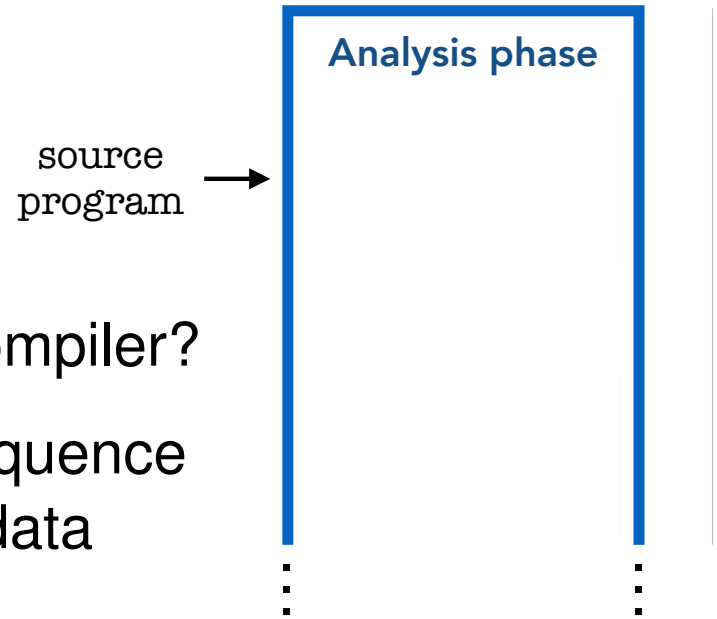


Question: what do these phases have in common?

The phases are purely functional: they take one input, and return one output. Languages such as Haskell, O'Caml, F#, Rust or Scala are ideal for writing compilers.

Phases

Analysis phase



Question: What is the input to a compiler?

A string (i.e., usually a very long sequence of characters). Not a very efficient data structure to work with . . .

Compilers generate code from a more convenient data structure called “abstract syntax trees” (ASTs).

We construct the AST in two phases:

- **Lexical analysis:** convert the input into a list of ‘tokens’.
- **Syntactical analysis:** assemble AST from the token list.

Phases

Lexical analysis

In the lexical analysis, a string is converted into a list of tokens.
Example: The program

```
int testfun( int n ){
  int res = 1;
  while (n>0){
    n--;
    res *= 2;
  }
  return res;
}
```

could be represented as the list

```
T_int, T_ident("testfun"), T_lparen,
T_int, T_ident("n"), T_rparen,
T_lbrace, T_int, T_ident ("res"),
T_eq, T_num(1), T_semicolon, T_while, ...
```

Phases

Lexical analysis

```
T_int, T_ident("testfun"), T_lparen,
T_int, T_ident("n"), T_rparen,
T_lbrace, T_int, T_ident ("res"),
T_eq, T_num(1), T_semicolon, T_while, ...
```

Another list, of more complicated things.
How is this an improvement?

- Instead of characters, a list of *language primitives*.
- Nothing in the list which *isn't* a language primitive.
- Abstracts away irrelevant details — e.g., syntax of keywords, whitespace, comments.

A step from 'letters' to 'words', reducing the work of the next stage: *parsing*.

Phases

Syntax analysis (parsing)

Parsing converts the program — the list of tokens —
into a tree: the AST of the program.

- Compare to the Document Object Model of a webpage
- Syntax-checking (type-checking) and code-generation can be done by doing a traversal of the AST.

But how is a program a tree?

Phases

Syntax analysis (parsing)

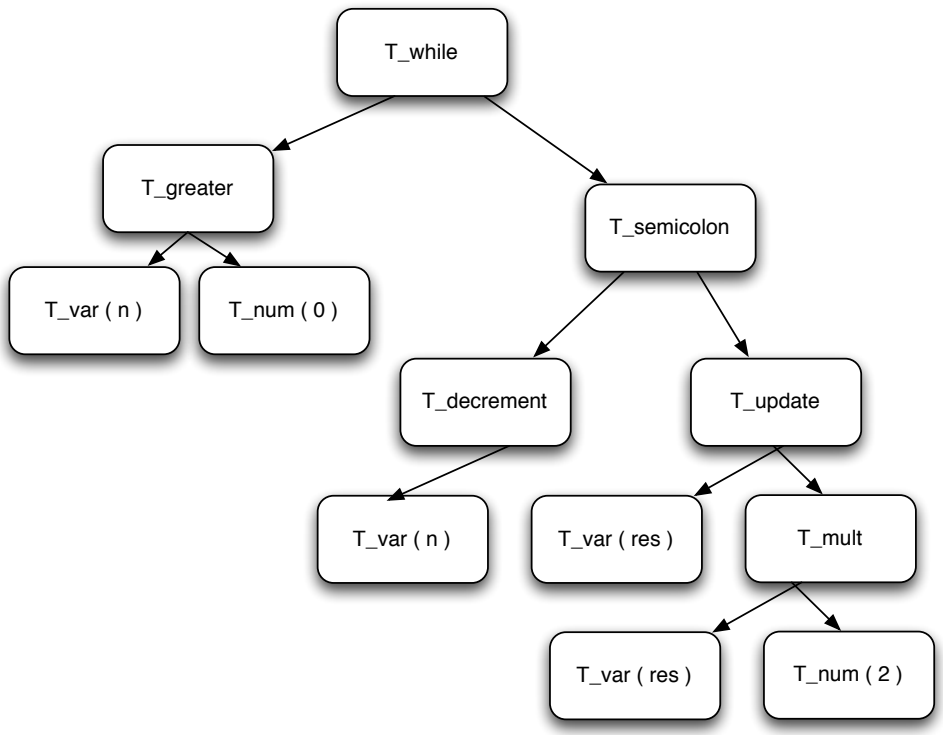
Example.

Source code: `while (n>0) { n--; res *= 2; }`

Token list:

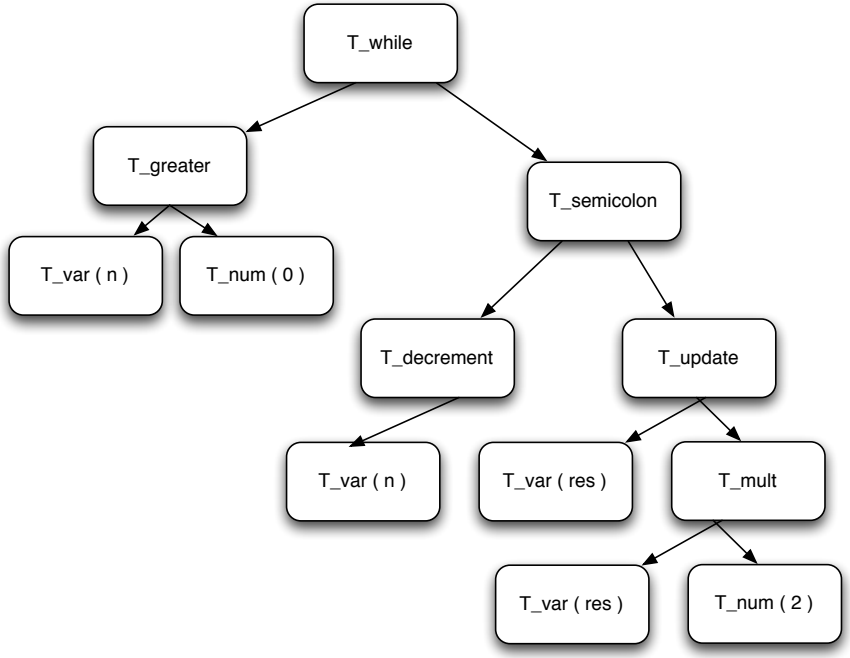
```
T_while, T_lparen, T_ident("n"), T_greater,
T_num(0), T_rparen, T_lbrace, T_ident("n"),
T_decrement, T_semicolon, T_ident("res"),
T_multeq, T_num(2), T_semicolon, T_rbrace
```

AST:



Phases

Syntax analysis (parsing)

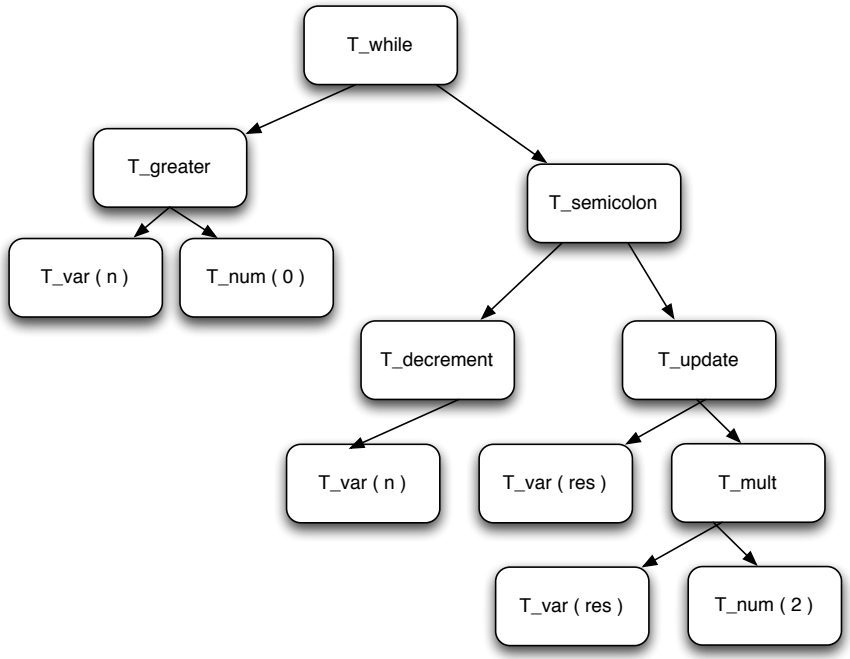


Lexical analysers and parsers can be automated, using **parser generators** (*e.g.*, **`flex`**, **`yacc`**, **`bnfc`**, **`antlr`**).

- Sophisticated — in practise they generate better lexers and parsers than most programmers would be able to.
- But: to understand them, it is a helpful to write a toy lexer and parser. (And someone has to write/maintain them!)

Phases

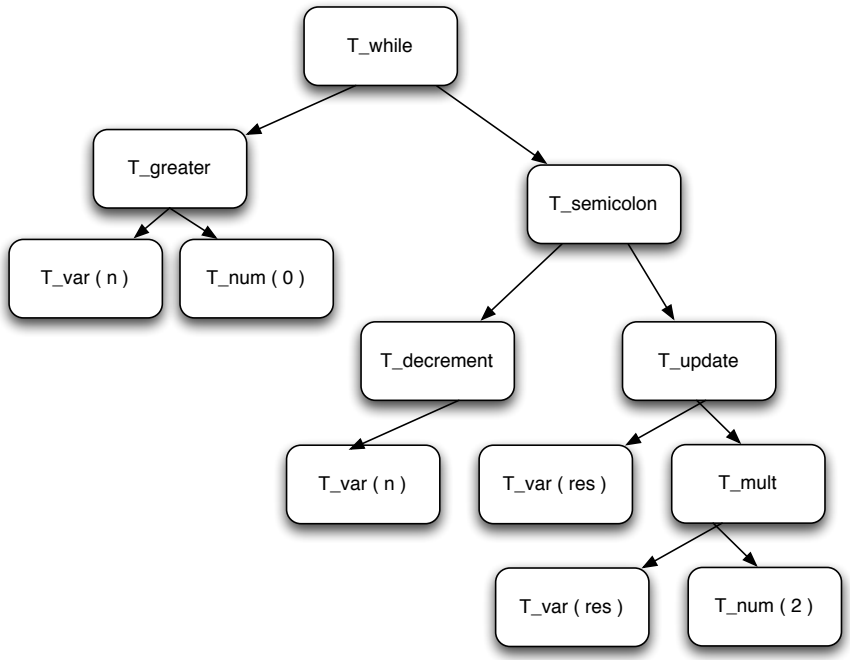
Syntax analysis (parsing)



- The performance of the compiler strongly depends on the AST — a lot of optimisation goes here for instustrial strength compilers.
- The compiler writer must design the AST data structure to be easy to build and traverse.

Phases

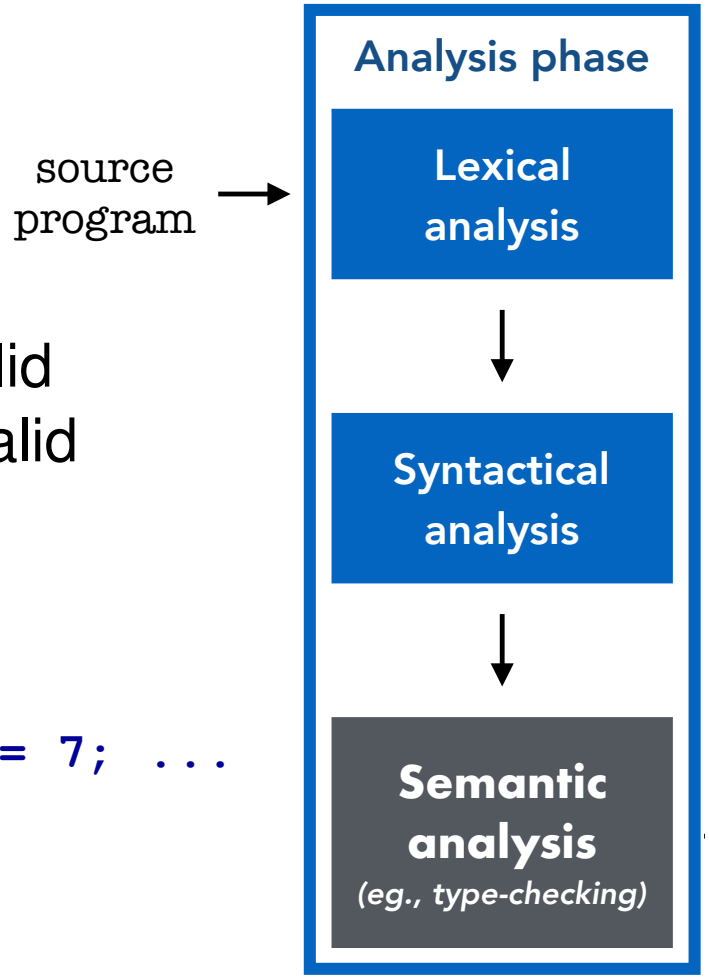
Syntax analysis (parsing)



In constructing the AST, we also verify the correctness of the syntax of the program.

- Rules for constructing the AST are essentially the rules that determine the set of syntactically valid programs
- Uses the theory of formal languages — *e.g.*, context free grammars — which we will study in detail.

Phases
Semantic analysis



Parsing can reject syntactically invalid programs, but not **semantically** invalid programs

Examples.

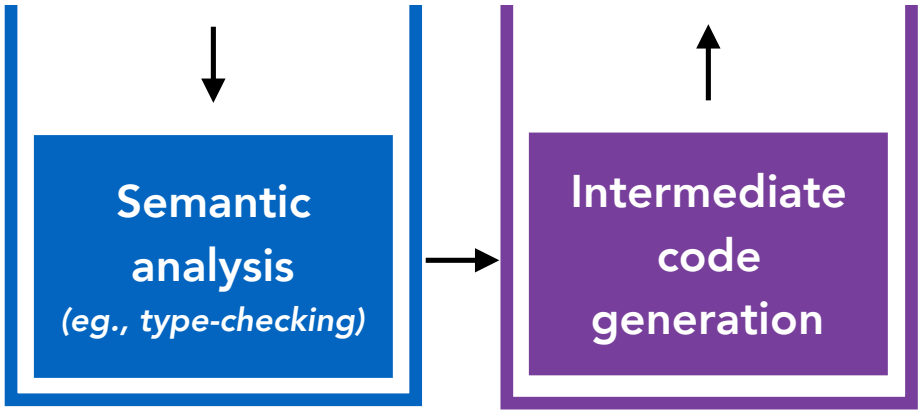
```
void main () { i = 7.5; int i = 7; ...

if (3 + true > "hello") ...
```

They are caught with semantic analysis. The key technology are types. Modern languages like Scala, Rust, Haskell, Ocaml, F# employ type inference.

Phases

Intermediate code generation

- There are many different CPUs (including different versions of a single line of chips), each with its own machine language



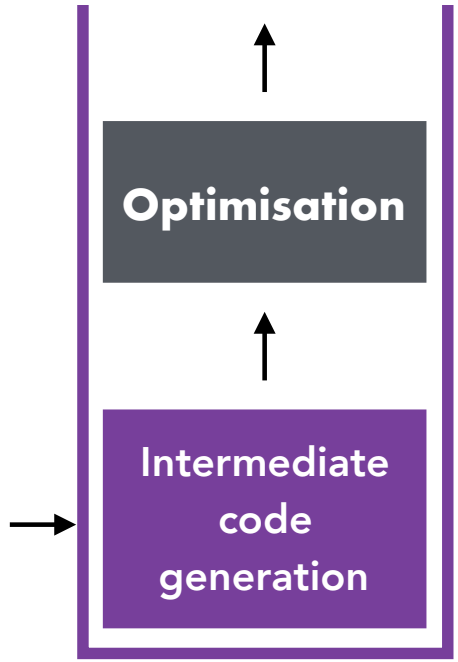
- To make compilation more portable, we define an **intermediate representation** (or IR), to:
  - Translate the AST of the source program to the IR.
  - Do compiler optimisations in the intermediate language.
  - Translate the intermediate representation to the target machine language.

An IR can capture the structure of almost all machine languages $\Longrightarrow$ simple machine code generation. We can target new machines by re-implementing the last step.

Phases
Optimiser

Translating a program often introduces various inefficiencies — *e.g.*, by making the program

- run more slowly
- more memory-intensive
- consume more power
  (important for mobile devices!)



Optimisers try to remove these inefficiencies by transforming the IR, without changing its meaning.

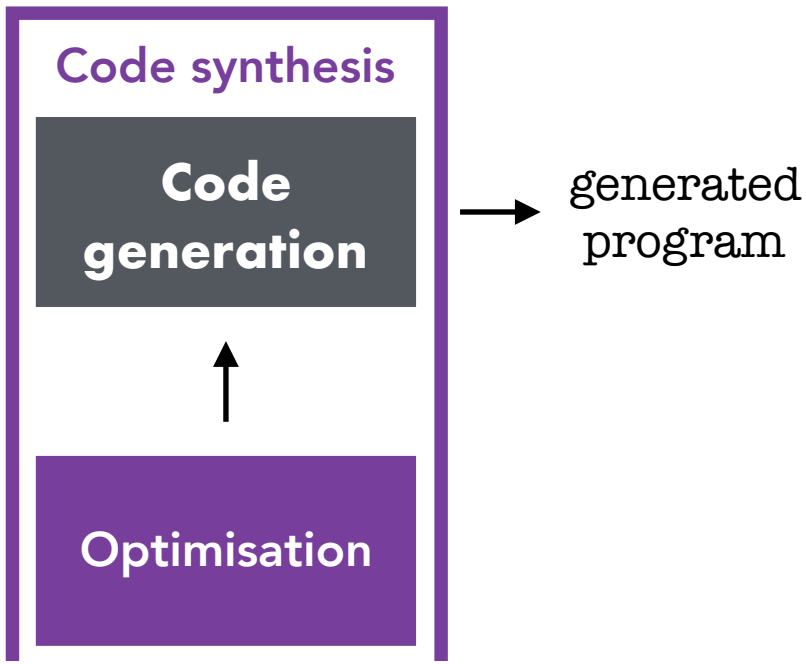
- This is **HARD** to do (**NP**-hard, or worse!) and usually leads to modest improvements
  - For JITs, it gets worse: some of it is performed at run-time!
- Some optimisations are easy (*e.g.*, inlining short functions), but even these must be done with care

Phases

Code generation

This straighforward phase translates the generated intermediate code to machine code.

As machine code and the IR are much alike, this 'mini-compiler' is simple and fast.

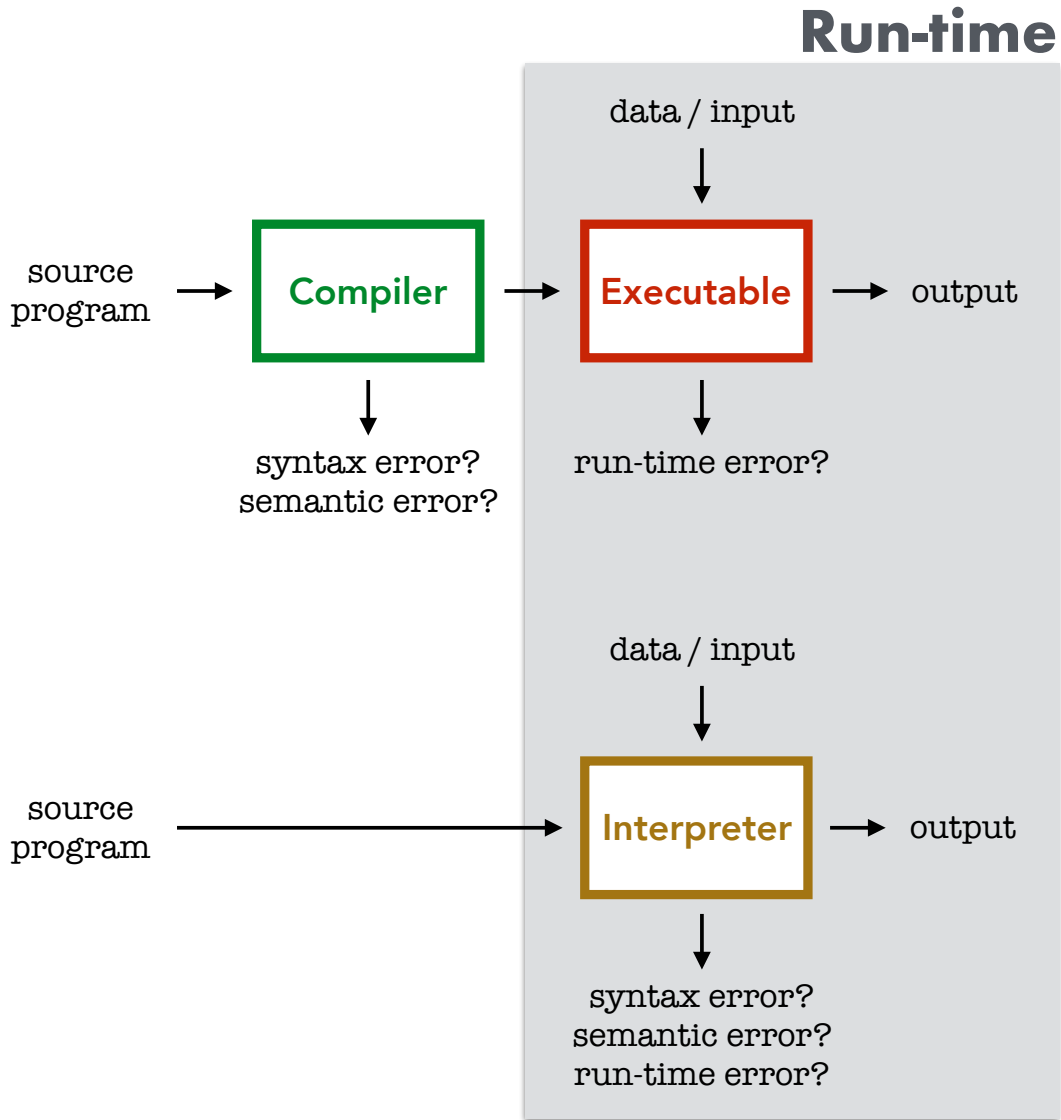


Compilers vs interpreters

Interpreters are a second way to run programs.

- Code generated by compilers can be faster, by processing the entire program at once, with results which can be re-used for each execution.
- Interpreters are better for interaction, and can be much simpler than compilers.

We won't say much more about interpreters in this course.



Questions?