

50.051 Programming Language Concepts

W8-S2 Introduction to Compilers and Course Outline

Matthieu De Mari



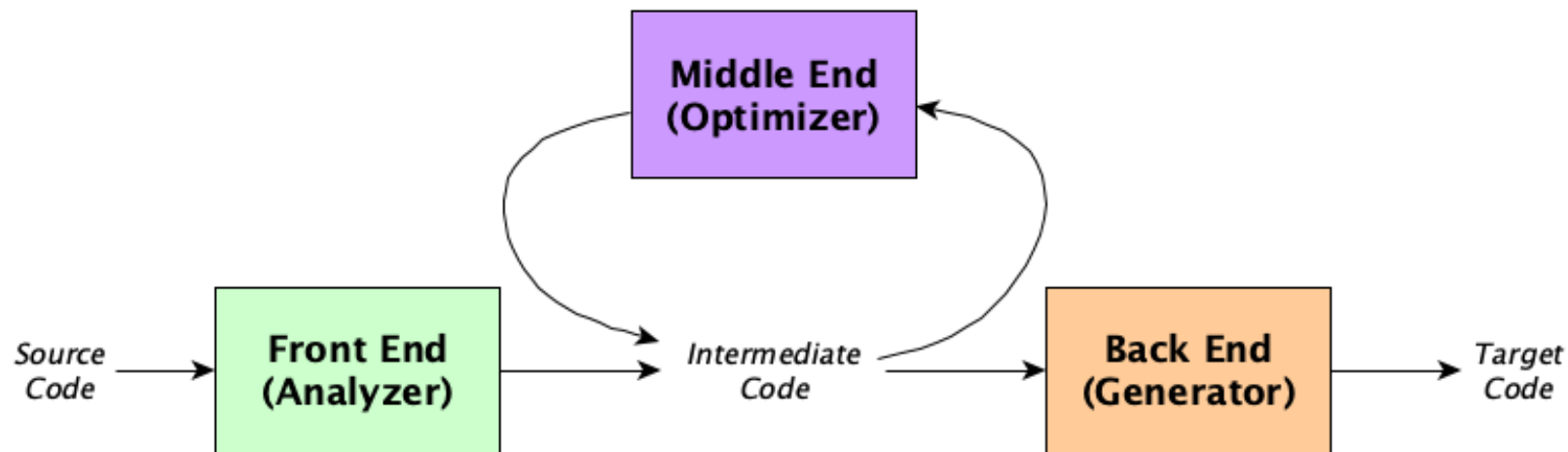
SINGAPORE UNIVERSITY OF
TECHNOLOGY AND DESIGN

The architecture of a compiler

Definition (the three parts of a typical compiler architecture):

A typical compiler architecture consists of three main components: the **front-end**, **middle-end**, and **back-end**.

Each component is responsible for a specific set of tasks and works together to generate the final executable code.

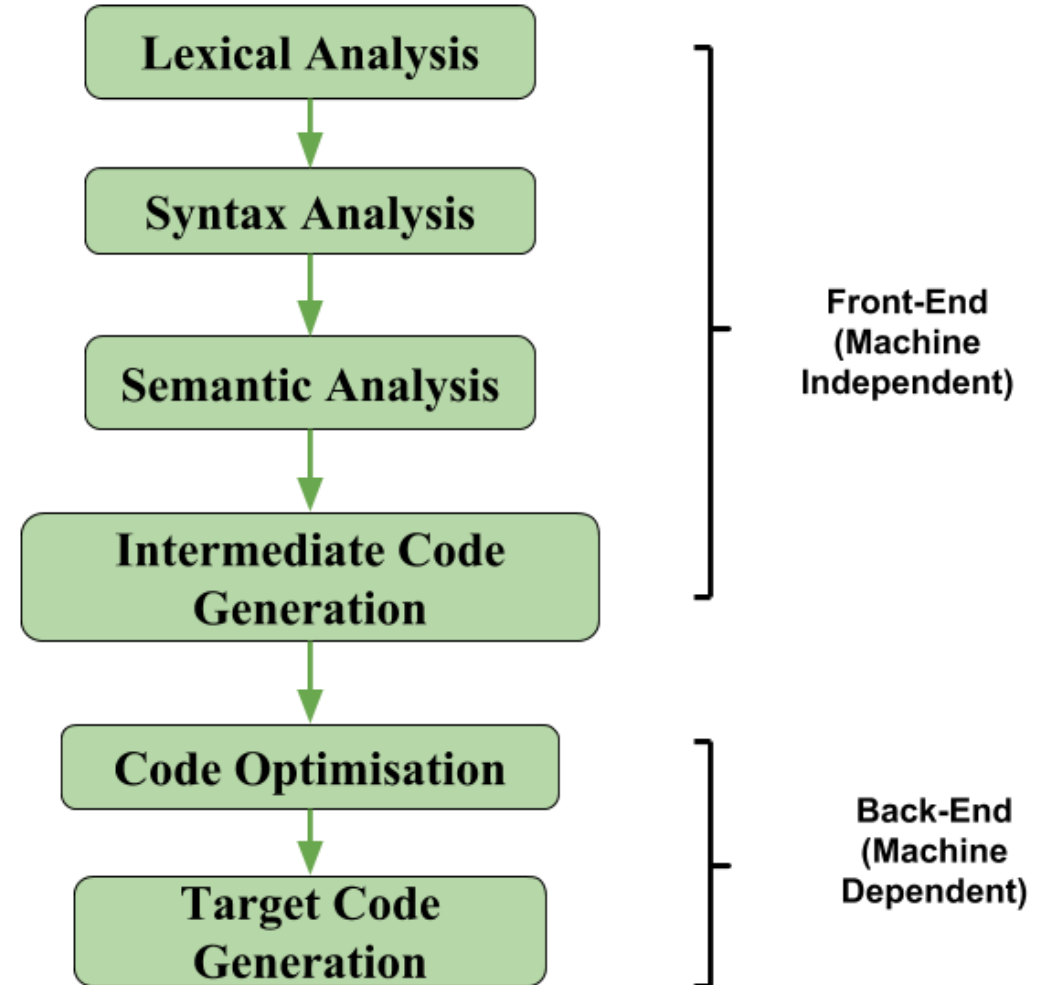


The architecture of a compiler

Definition (the three parts of a typical compiler architecture):

A typical compiler architecture consists of three main components: the **front-end**, **middle-end**, and **back-end**.

Each component is responsible for a specific set of tasks and works together to generate the final executable code.



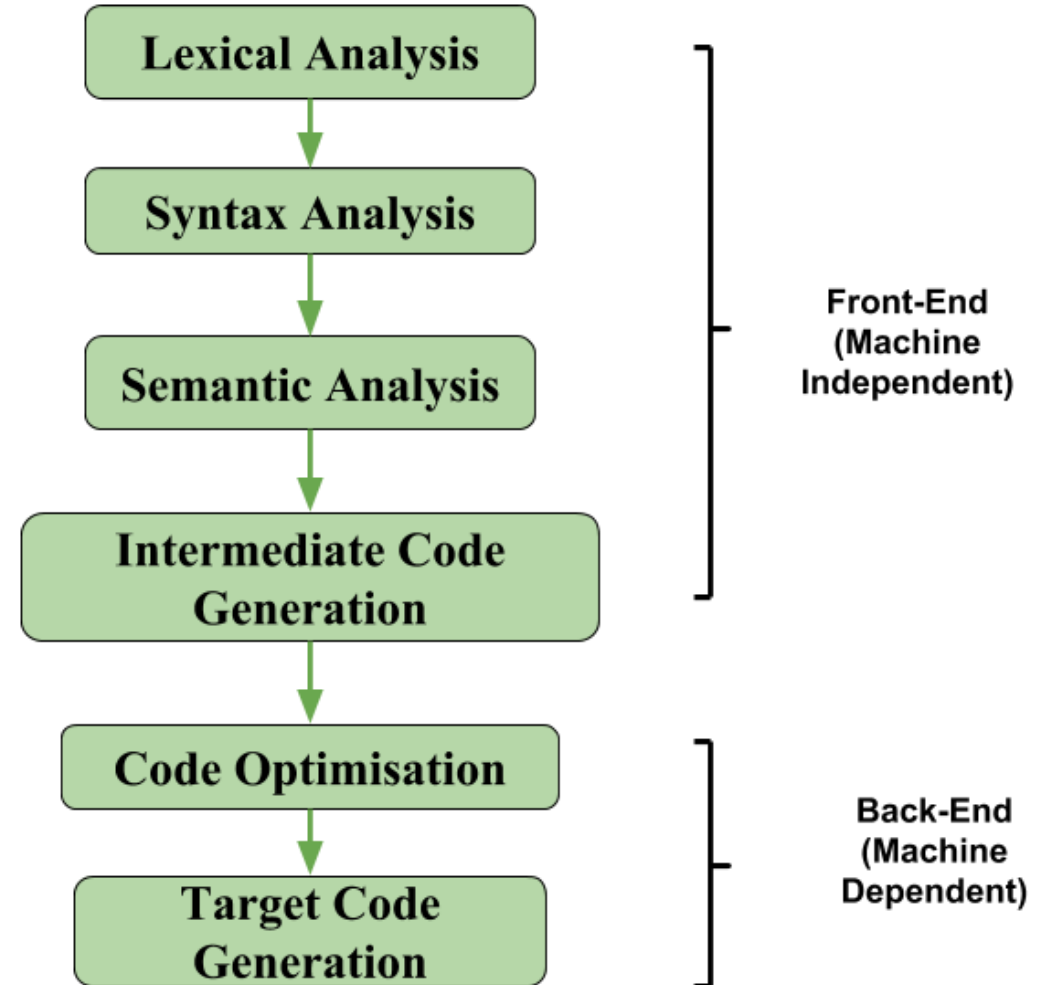
The front-end of a compiler

Definition (The front-end part of a compiler):

The **front-end of a compiler** is responsible for analysing the source code, and converting it into a form that can be used by the rest of the compiler.

It involves tasks, such as:

- **Lexical analysis,**
- **Syntax analysis,**
- and **Semantic analysis.**



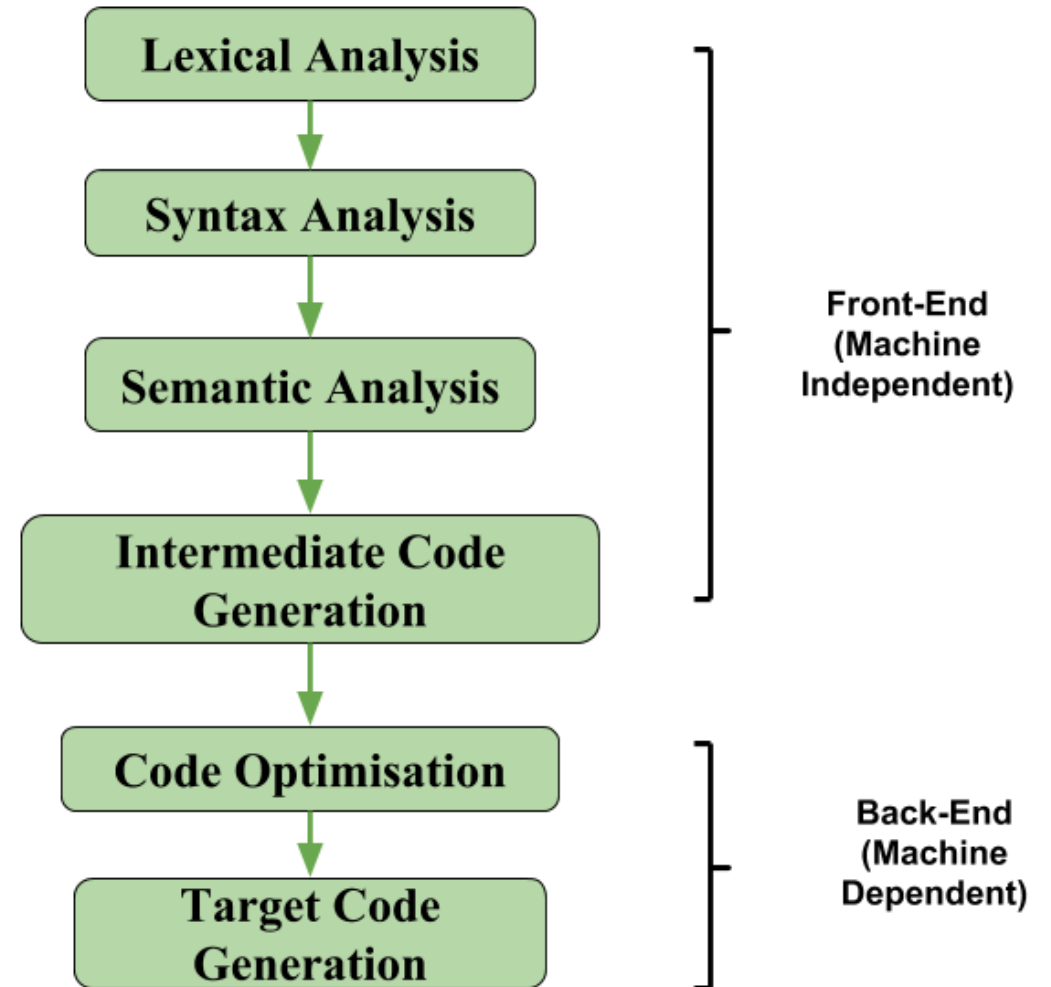
The middle-end of a compiler

Definition (The middle-end part of a compiler):

The **middle-end of a compiler** follows the front-end analysis and it consists of a series of operations and transformations to optimize and improve its efficiency.

It involves tasks, such as:

- **Intermediate code generation**
- **Code optimization,**
- and **Data-flow analysis.**

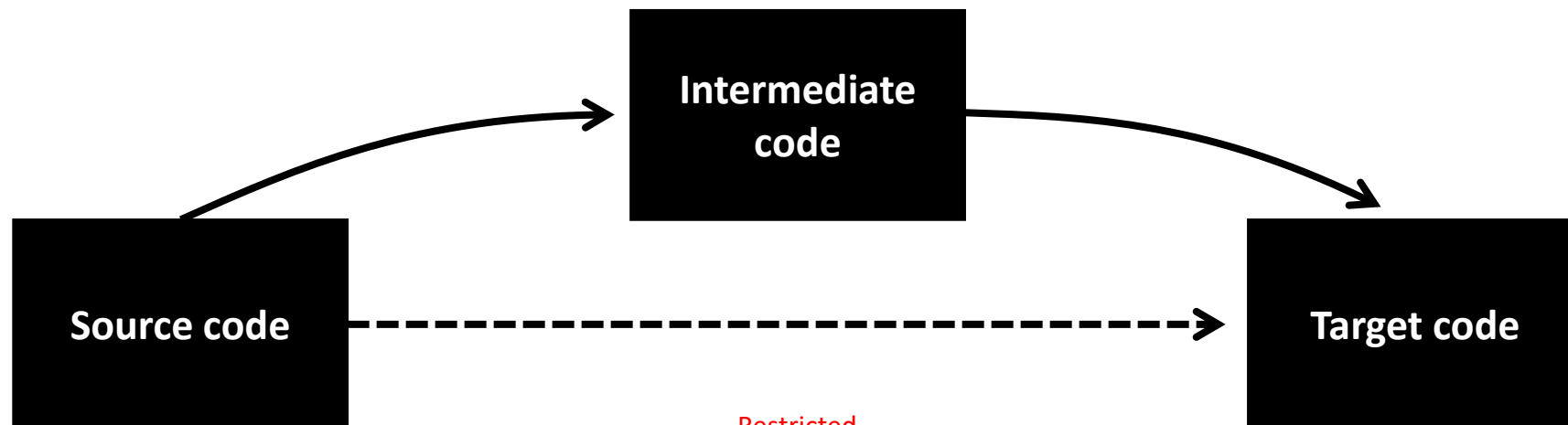


Intermediate code generation

Definition (Intermediate code generation):

Intermediate code generation is the process of transforming the source code into a code that is more abstract and closer to machine language.

Making a direct jump from source code to target code might prove difficult.



Intermediate code generation

Definition (Intermediate code generation):

Intermediate code generation is the process of transforming the source code into a code that is more abstract and closer to machine language.

Making a direct jump from source code to target code might prove difficult.

For this reason, an intermediate code representation is often easier to manipulate and allows for optimization.

The intermediate code is not specific to any particular hardware or operating system and can be easily transformed into the final machine code during the backend phase of the compilation process.

Code optimization

Definition (**Code optimization**):

Code optimization is a critical process that involves analysing and modifying the intermediate code to make it run faster and consume fewer resources. Some of the optimizations typically include:

1. **Dead code elimination:** identify and remove **dead code** that is not executed during program execution.
This allows to reduce the size of the final compiled code.

Dead code

Definition (dead code):

Dead code is code that is never executed during the runtime of a program, often following from bad code logic or design.

```
#include <stdio.h>

int main() {
    int x = 10;
    if (x < 5) {
        printf("x is less than 5\n");
    }
    // This code below is dead code, as the condition above will always evaluate to false
    else if (x < 8) {
        printf("x is less than 8\n");
    }
    else {
        printf("x is greater than or equal to 8\n");
    }
    return 0;
}
```

Dead code

Question: Can you see why the code shown on the right is containing some dead code?

```
#include <stdio.h>

int main() {
    int x = 10;
    if (x < 5) {
        printf("x is less than 5\n");
    }
    // This code below is dead code, as the condition above will always evaluate to false
    else if (x < 8) {
        printf("x is less than 8\n");
    }
    else {
        printf("x is greater than or equal to 8\n");
    }
    return 0;
}
```

Dead code

Answer: The *else if* statement is the dead code, as the preceding if statement will always evaluate to false since x is initialized to 10.

```
#include <stdio.h>

int main() {
    int x = 10;
    if (x < 5) {
        printf("x is less than 5\n");
    }
    // This code below is dead code, as the condition above will always evaluate to false
    else if (x < 8) {
        printf("x is less than 8\n");
    }
    else {
        printf("x is greater than or equal to 8\n");
    }
    return 0;
}
```

Dead code

Definition (dead code):

Dead code is code that is never executed during the runtime of a program, often following from bad code logic or design.

```
#include <stdio.h>

int main() {
    int x = 10;
    if (x < 5) {
        printf("x is less than 5\n");
    }
    // This code below is dead code, as the condition above will always evaluate to false
    else if (x < 8) {
        printf("x is less than 8\n");
    }
    else {
        printf("x is greater than or equal to 8\n");
    }
    return 0;
}
```

Code optimization

Definition (**Code optimization**):

Code optimization is a critical process that involves analysing and modifying the intermediate code to make it run faster and consume fewer resources. Some of the optimizations typically include:

- 2. Constant folding:** replace multiple expressions that involve constant values with their computed values.
This allows to reduce the number of computations performed during program execution.

Constant folding

As an example of constant folding...

In the code below, the expression $x = 2 + 3$ can be folded into a simpler expression $x = 5$ during compilation, eliminating the need to perform the addition operation at runtime.

```
#include <stdio.h>

int main() {
    int x = 2 + 3; // This expression is constant folded at compile-time
    printf("The value of x is %d\n", x); // The output will be "The value of x is 5"
    return 0;
}
```

Code optimization

Definition (**Code optimization**):

Code optimization is a critical process that involves analysing and modifying the intermediate code to make it run faster and consume fewer resources. Some of the optimizations typically include:

3. Control flow optimization: optimize the instructions and the order of instructions being executed.

This allows, for instance, to reduce the number of necessary branch instructions, removing parts that could almost be considered **dead code** (even though they did not follow from bad logic this time).

Control flow optimization

As an example of control flow optimization...

The program below uses an *if-else statement* to check if the value of *x* is less than the value of *y*.

```
#include <stdio.h>

int main() {
    int x = 10, y = 20;
    if (x < y) {
        printf("x is less than y\n");
    } else {
        printf("x is greater than or equal to y\n");
    }
    return 0;
}
```


Control flow optimization

As an example of control flow optimization...

The program below uses an *if-else statement* to check if the value of *x* is less than the value of *y*.

Control flow optimization analyses the code and can determine that *x* will always be less than *y*, as they are constants.

It can simplify the code to eliminate the unnecessary *if-else statement* and simplify the code even further.

```
#include <stdio.h>

int main() {
    int x = 10, y = 20;
    printf("x is less than y\n");
    return 0;
}
```

Code optimization

Definition (**Code optimization**):

Code optimization is a critical process that involves analysing and modifying the intermediate code to make it run faster and consume fewer resources. Some of the optimizations typically include:

- **Dead code elimination,**
- **Constant folding,**
- **Control flow optimization,**
- **Also, Function Inlining (substituting secondary functions into main), Loop Unrolling (replaces a loop with a fixed number of iterations with a series of unrolled iterations), etc.**

Code optimization

Definition (Code optimization):

Code optimization is a critical process that involves analysing and modifying the intermediate code to make it run faster and consume fewer resources. Some of the optimizations **typically** include:

- **Dead code elimination,**
- **Constant folding,**
- **Control flow optimization.**
- **Also, Function Inlining (substituting secondary functions into main), Loop Unrolling (replaces a loop with a fixed number of iterations with a series of unrolled iterations), etc.**

(More on this in Week 10).

Intermediate code representations

Definition (intermediate code representations):

The intermediate code generated during this phase is often represented in a language that is easier to manipulate than the original source code.

Some examples of intermediate code representation languages that we will investigate in Week 11, are:

- **Three-address code,**
- **Virtual machine code,**
- and **Abstract syntax trees.**

Three-address code representation

Definition (Three-address code representation):

Three-address code is a low-level intermediate code representation used by compilers to facilitate optimization and code generation.

It is called “**three-address**” because each instruction in the code **can have at most three operands**.

A typical three-address code instruction has the following format:

$$\textit{operand1} = \textit{operand2} \textit{operator} \textit{operand3}$$

Three-address code representation

For instance, the C code below can be transformed...

...into its equivalent three-address code representation.

```
#include <stdio.h>

int main() {
    int x = 10;
    int y = x + 5;
    printf("The value of y is %d\n", y);
    return 0;
}
```

```
t1 = 10
t2 = t1 + 5
y = t2
printf("The value of y is %d\n", y)
```

Virtual Machine code representation

Definition (Virtual Machine code representation):

Virtual Machine code (VM code) is an intermediate representation of code that is designed to be executed on a virtual machine rather than on the hardware directly.

It is typically used by compilers that target virtual machines (e.g. the Java Virtual Machine or the .NET Common Language Runtime).

It is a stack-based representation, where each operation is performed on values that are pushed onto and popped off of a stack. Each VM instruction typically consists of an opcode and zero or more operands.

Virtual Machine code representation

For instance, the C code on the right can be transformed...

...into its equivalent VM code representation, below.

```
#include <stdio.h>

int main() {
    int x = 10;
    int y = x + 5;
    printf("The value of y is %d\n", y);
    return 0;
}
```

```
push 10          ; Push the value 10 onto the stack
push 5           ; Push the value 5 onto the stack
add              ; Pop the top two values off the stack, add them, and push the result onto the stack
pop y           ; Pop the top value off the stack and store it in the variable y
push y           ; Push the value of y onto the stack
push "The value of y is %d\n" ; Push the format string onto the stack
printf          ; Pop the format string and the value of y off the stack and print them to the console
push 0          ; Push the value 0 onto the stack
return          ; Return from the function
```


Abstract syntax trees code representation

Definition (Abstract Syntax Tree representation):

The **Abstract Syntax Tree (AST) representation** is a tree representation of the syntactic structure of a program that is generated by a compiler during the front-end phase.

Each node in the tree then corresponds to a syntactic construct in the program, such as an operation, expression or statement.

While a lot more advanced, the tree structure makes it easy for compilers to analyse and manipulate the program syntax and perform optimizations.

Abstract syntax trees code representation

For instance, the C code below can be transformed...

```
#include <stdio.h>

int main() {
    int x = 10;
    int y = x + 5;
    printf("The value of y is %d\n", y);
    return 0;
}
```

Abstract syntax trees code representation

For instance, the C code below can be transformed...
...into its equivalent abstract syntax tree code representation.

```
Program
|- Function: main
  |- Declaration: x (type: int)
    |- Literal: 10
  |
  |- Declaration: y (type: int)
    |- Binary Expression: +
      |- Identifier: x
      |- Literal: 5
  |
  |- Function Call: printf
    |- String Literal: "The value of y is %d\n"
    |- Identifier: y
```

Intermediate code representations

Definition (intermediate code representations):

The intermediate code generated during this phase is often represented in a language that is easier to manipulate than the original source code.

Some examples of intermediate code representation languages that we will investigate in Week 11, are:

- **Three-address code,**
- **Virtual machine code,**
- and **Abstract syntax trees.**

(More on this during Week 11).

Data-flow analysis

Definition (**Data-flow analysis**):

Data-flow analysis is another task performed during the middle-end stage, which analyses the flow of data in the program and identifies opportunities for optimization.

In data-flow analysis, the program is modelled as a set of variables and operations that manipulate each variable. Each variable has a defined value at each point in the program, and the analysis tracks how the value of each variable changes as the program executes.

Typically, it tracks how variables are defined and used throughout the program and performs optimizations such as dead code elimination (again!) and constant propagation (also!).

Data-flow analysis

For this code, in C...

```
#include <stdio.h>

int main() {
    int x = 10;
    int y = 5;
    int z = 0;

    if (x > y) {
        z = x - y;
    } else {
        z = y - x;
    }

    printf("The value of z is %d\n", z);
    return 0;
}
```

...the Data-flow analysis gives

Block 1:

```
x = 10
y = 5
z = 0
```

Block 2:

```
t1 = x > y
if t1 goto Block 3
goto Block 4
```

Block 3:

```
z = x - y
goto Block 5
```

Block 4:

```
z = y - x
```

Block 5:

```
printf("The value of z is %d\n", z);
```

Block 6:

```
return 0
```

Data-flow analysis

- In this data-flow analysis, the program is divided into basic blocks, each representing a sequence of instructions with no branches or jumps (apart for jumping to other blocks).
- Using this data-flow analysis, the compiler can perform optimizations such as constant propagation and dead code elimination.

Block 1:

```
x = 10  
y = 5  
z = 0
```

Block 2:

```
t1 = x > y  
if t1 goto Block 3  
goto Block 4
```

Block 3:

```
z = x - y  
goto Block 5
```

Block 4:

```
z = y - x
```

Block 5:

```
printf("The value of z is %d\n", z)
```

Block 6:

```
return 0
```

Data-flow analysis

For instance...

1. Since the value of x is never modified, the compiler can replace all occurrences of x with the constant value 10. Same for y .
2. Both values of x and y being constants, the compiler can perform the comparison $x > y$ and computation of z at compilation time.
3. This eventually eliminates the need for the *if* conditional statement and reduces the program to its minimal form.

Block 1:

```
x = 10  
y = 5  
z = 0
```

Block 2:

```
t1 = x > y  
if t1 goto Block 3  
goto Block 4
```

Block 3:

```
z = x - y  
goto Block 5
```

Block 4:

```
z = y - x
```

Block 5:

```
printf("The value of z is %d\n", z)
```

Block 6:

```
return 0
```


Data-flow analysis

- In this data-flow analysis, the program is divided into basic blocks, each representing a sequence of instructions with no branches or jumps (apart for jumping to other blocks).
- Using this data-flow analysis, the compiler can perform optimizations such as constant propagation and dead code elimination.
- *(More about this on Week 11).*

Block 1:

```
x = 10  
y = 5  
z = 0
```

Block 2:

```
t1 = x > y  
if t1 goto Block 3  
goto Block 4
```

Block 3:

```
z = x - y  
goto Block 5
```

Block 4:

```
z = y - x
```

Block 5:

```
printf("The value of z is %d\n", z)
```

Block 6:

```
return 0
```

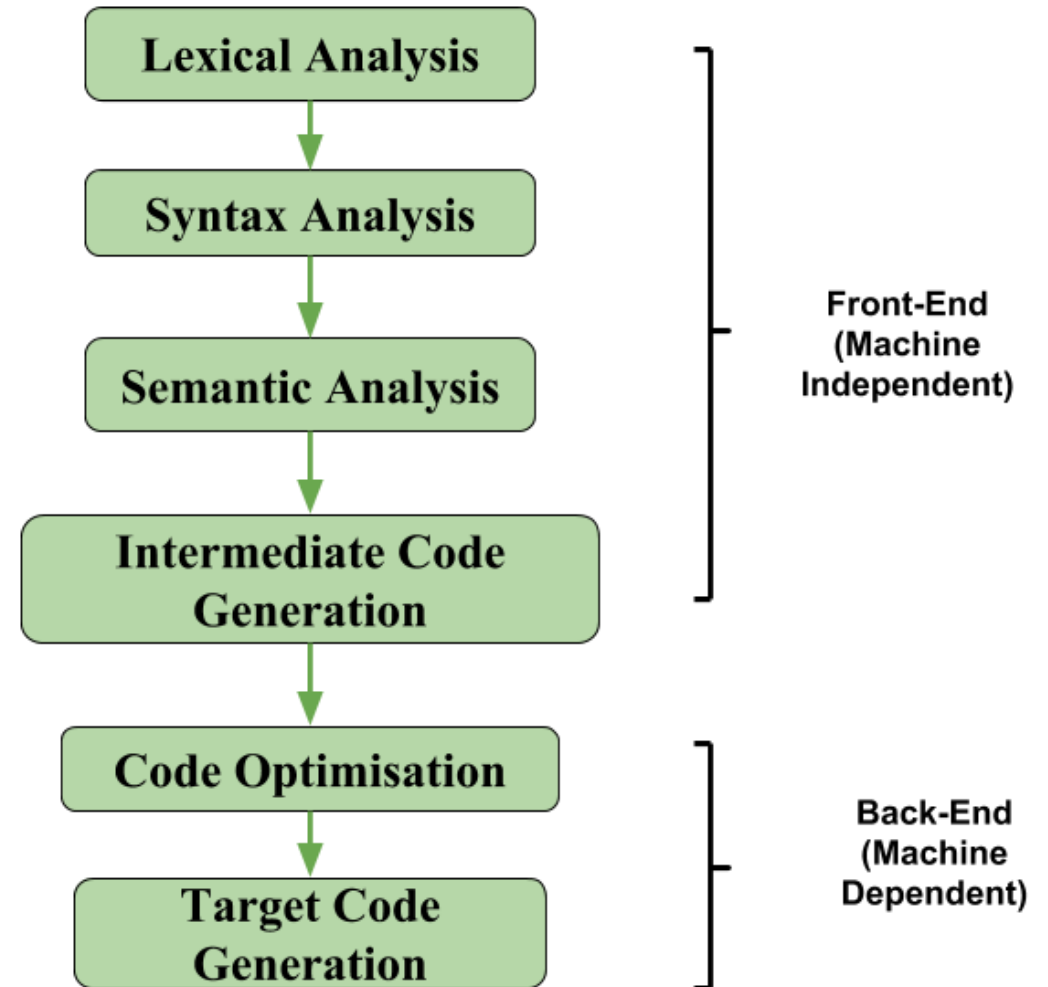
The back-end of a compiler

Definition (The back-end part of a compiler):

The **back-end of a compiler** takes the optimized and transformed code generated by the middle-end and translates it into machine code that can be executed.

It involves tasks, such as:

- **Instruction selection/ordering,**
- **Register allocation,**
- **and Code generation.**



Instruction Selection

Definition (**Instruction Selection**):

During **instruction selection**, the compiler chooses the specific machine code instructions (mov, add, jmp, etc) to use to implement each operation in the program.

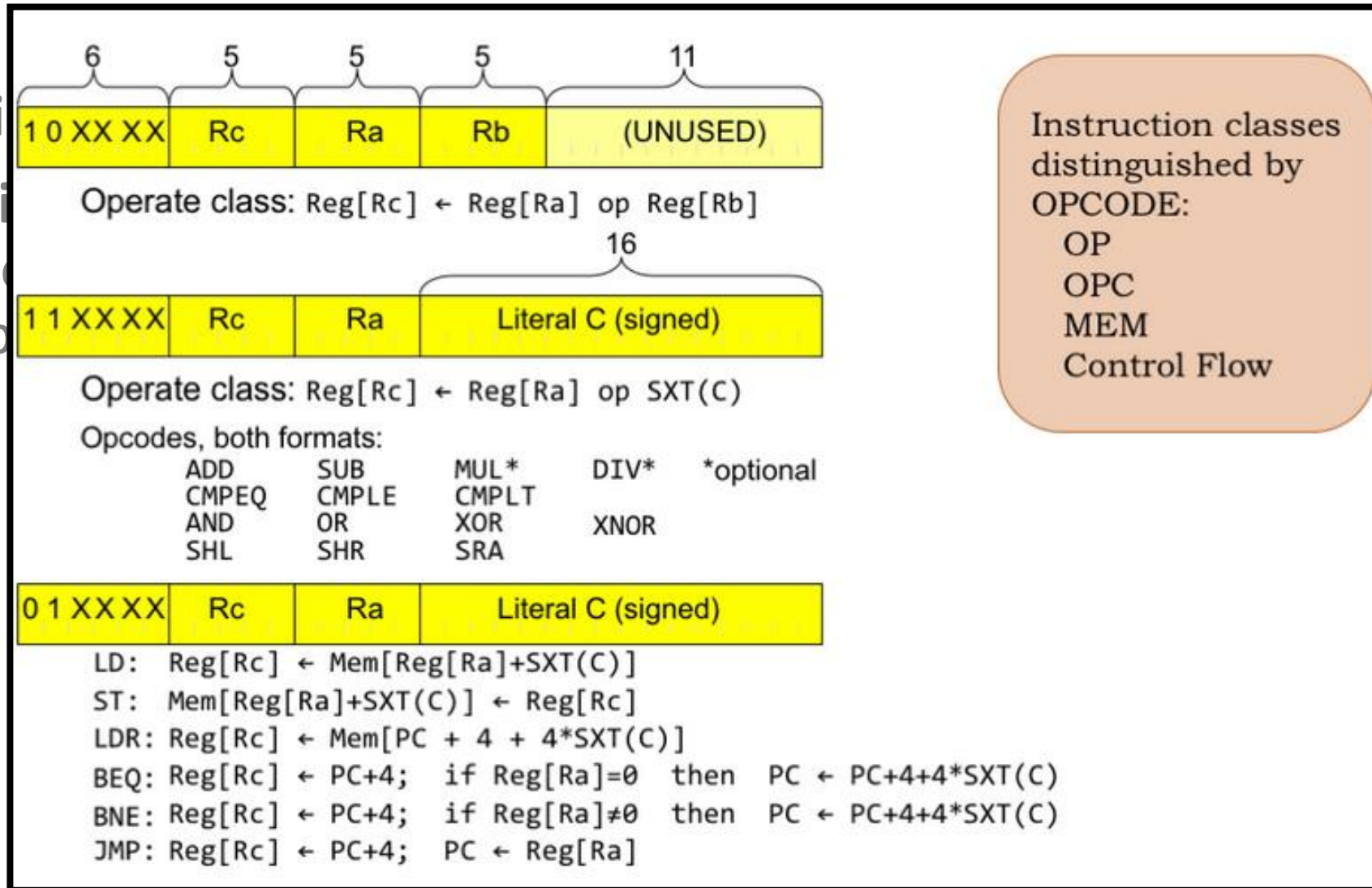
During this step, it might perform additional transformations on the generated code to improve performance and reduce code size.

Typically, there might be multiple equivalent ways to choose and order instructions

Instruction Selection

Definition

During instruction
machine
each operation



Instruction classes
distinguished by
OPCODE:
OP
OPC
MEM
Control Flow

ment

Instruction Selection

Definition (**Instruction Selection**):

During **instruction selection**, the compiler chooses the specific machine code instructions (mov, add, jmp, etc) to use to implement each operation in the program.

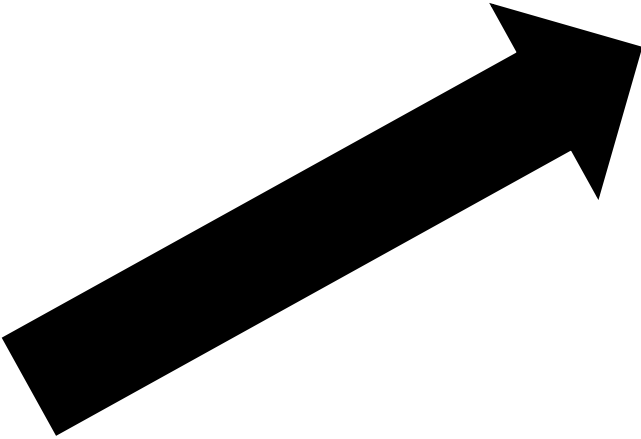
During this step, it might perform some final transformations on the generated code to improve performance and reduce code size.

Typically, there might be multiple ways to choose and order machine code instructions. Some might be equivalent, others might allow to save memory/register space, etc.

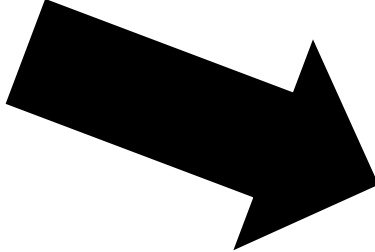
Instruction Selection

```
#include <stdio.h>

int main() {
    int x = 7;
    int y = 2;
    int z = 5;
    int w = 3;
    int result = (x + y) * (z - w);
    printf("The result is %d\n", result);
    return 0;
}
```



```
mov r1, [x]
mov r2, [y]
add r3, r1, r2
mov r4, [z]
mov r5, [w]
sub r6, r4, r5
mul r7, r3, r6
mov [result], r7
```



```
mov r1, [x]
mov r2, [y]
mov r4, [z]
mov r5, [w]
sub r6, r4, r5
add r3, r1, r2
mul r7, r3, r6
mov [result], r7
```

Instruction Ordering/Scheduling

Definition (**Instruction Ordering/Scheduling**):

The **instruction ordering** (or **scheduling**) phase is responsible for rearranging the order of instructions in the program to improve performance.

This typically involves reordering instructions to take advantage of the parallelism and pipelining features of modern processors (with many cores running in parallel, for instance).

The instruction scheduler takes into account the dependencies between instructions and the specific features of the processor.

(Too advanced, out-of-scope).

Register Allocation

Definition (**Register Allocation**):

During the **register allocation** phase, the compiler assigns program variables to machine registers in an efficient way.

As seen in [50.002 Computation Structures](#), this is important because register access is much faster than memory access and can then improve the program performance.

The register allocator must take into account the constraints imposed by the target architecture, such as the number and type of available registers.

(Could have been interesting, but probably out-of-scope).

Code generation

Definition (**code generation**):

The **code generation** phase is responsible for generating machine code based on the intermediate code representation of the program produced by the middle-end part of the compiler.

This involves translating the intermediate representation into a low-level machine language that can be executed by the CPU (or GPU!).

In its most advanced versions, the code generator must take into account the specifics of the target architecture, such as the instruction set, memory hierarchy, and addressing modes.

And then, after the machine code is ready, we execute it, as explained in [50.002 Computation Structures!](#)

Recall your machine code from 50.002!

The machine code, to be executed by the CPU has

- **Decomposed the source code into basic instructions understandable by the CPU (or GPU),**
- **Identified registers to use to store (and free) variables for each of these operations.**

Need a refresher? Have a look at your [50.002 Computation Structures!](#)

```
movl 10, Reg1 ;10 into Reg1
movl 5, Reg2  ;5 into Reg2
add Reg3, Reg1, Reg2 ;add Reg1 + Reg2 store in Reg3
ret  Reg3 ;return value in Reg3
```

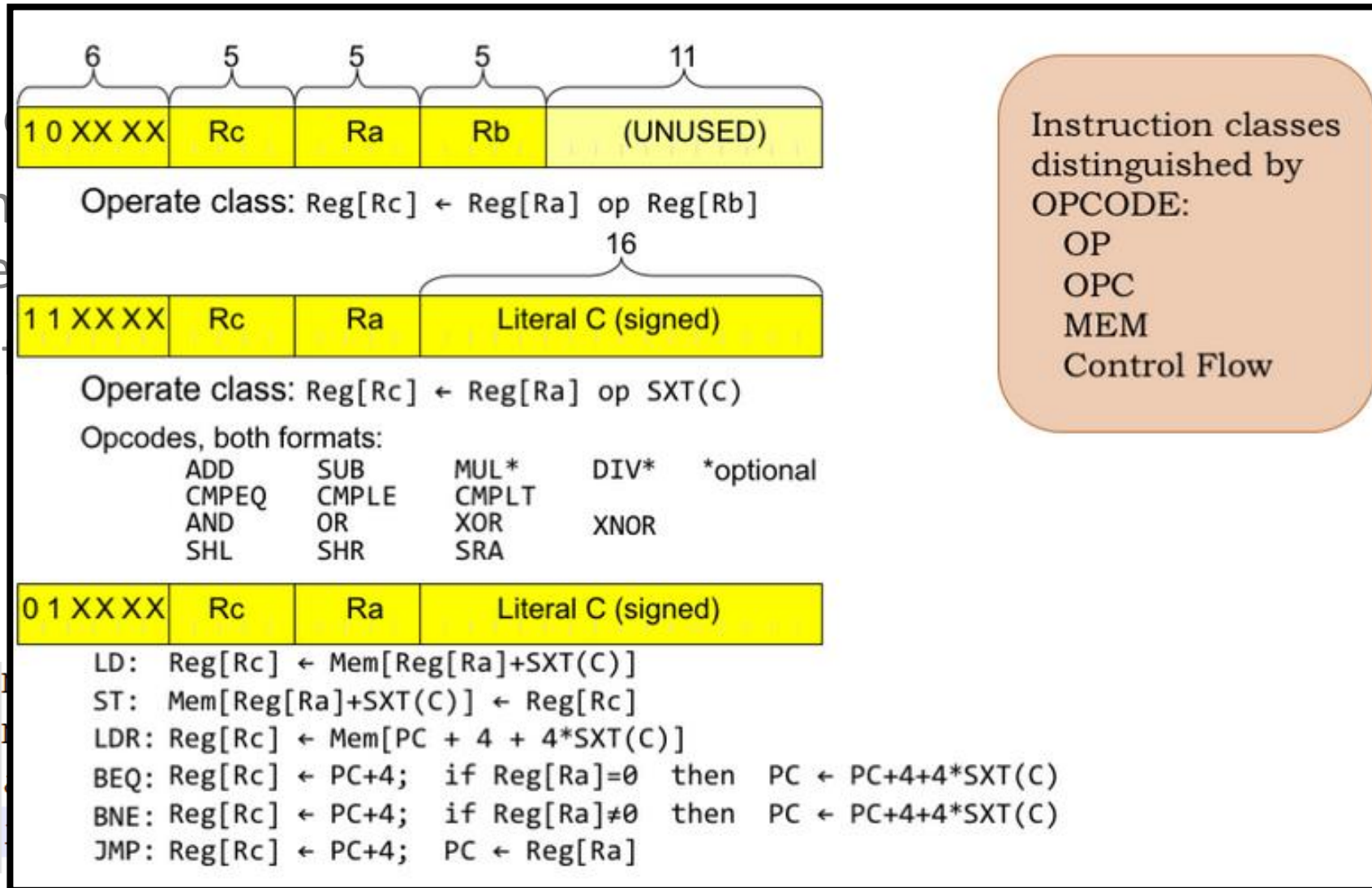
Recall your machine code from 50.002!

The machine code

- Decompose by the

- Identify these

Need a



ndable

of

ctures!

3

Source code (C)

```
int main() {  
    int x = 10;  
    int y = 5;  
    int z = x + y;  
    return z;  
}
```

Restricted

Front-end

Lexical Analysis
Syntax Analysis
Semantic Analysis

After front-end, intermediate code (Three-address) and optimizations

```
t1 = 10  
t2 = 5  
t3 = t1 + t2  
z = t3  
return z
```

Machine instructions after code generation and execution

```
movl 10, Reg1  
movl 5, Reg2  
add Reg3, Reg1, Reg2  
ret Reg3
```

Restricted

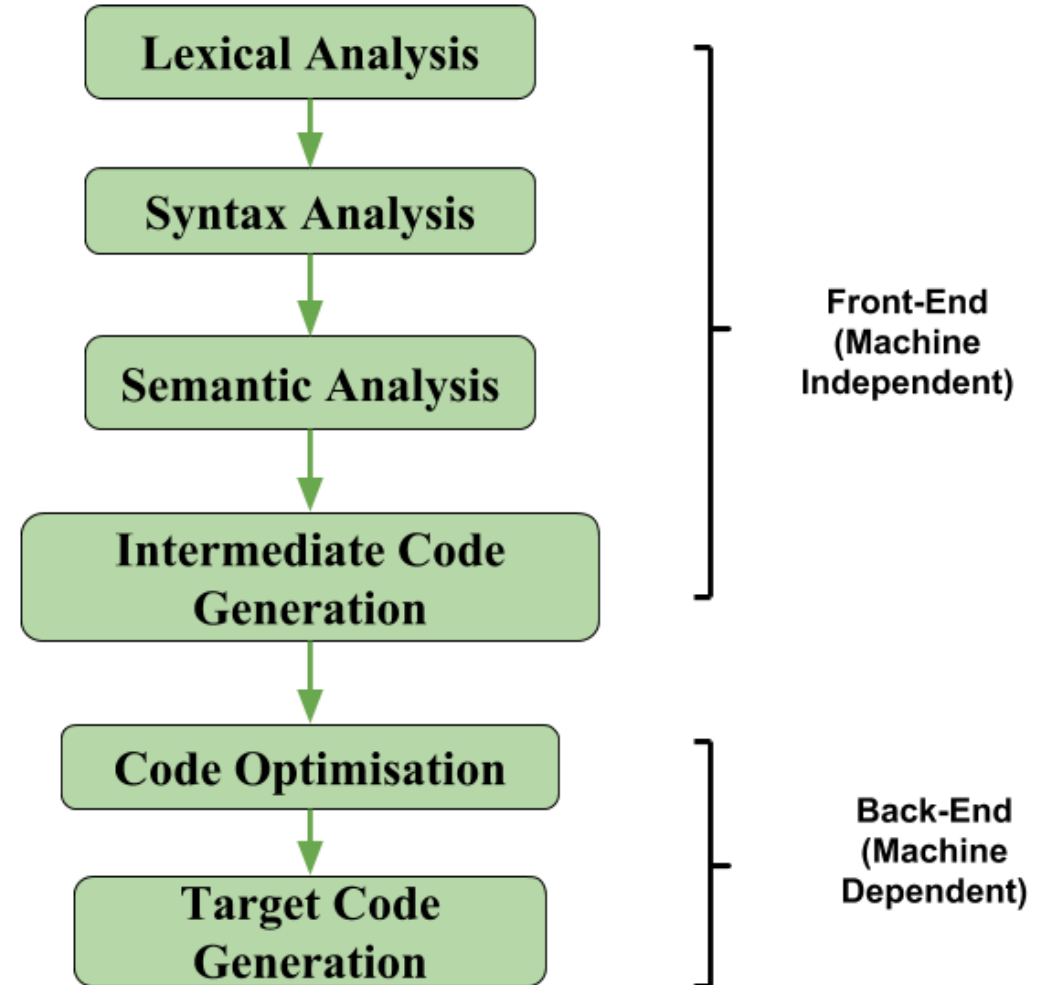
The back-end of a compiler

Definition (The back-end part of a compiler):

The **back-end of a compiler** takes the optimized and transformed code generated by the middle-end and translates it into machine code that can be executed.

- **Instruction selection/ordering,**
- **Register allocation,**
- and **Code generation.**

(Briefly discussed in W12-13).



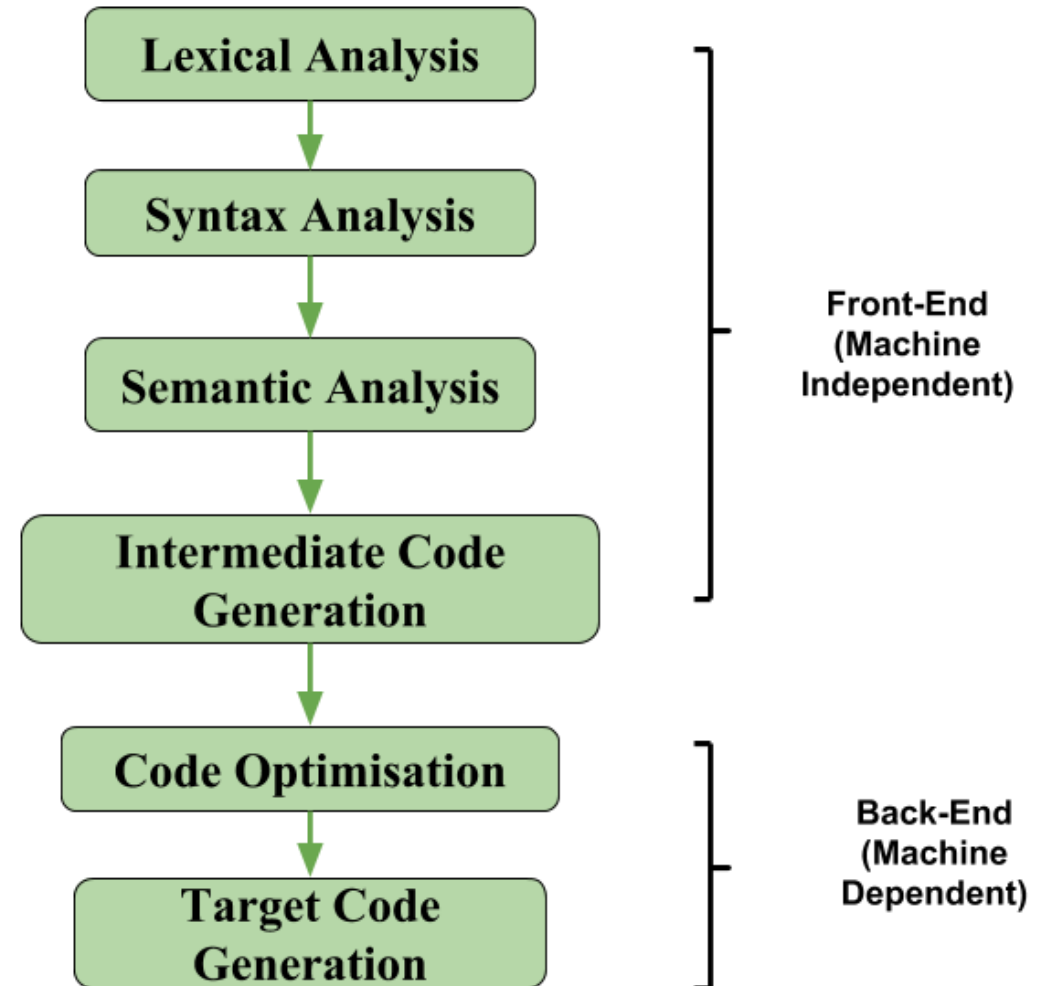
The architecture of a compiler

Definition (the three parts of a typical compiler architecture):

A typical compiler architecture consists of three main components: the **front-end**, **middle-end**, and **back-end**.

Each component is responsible for a specific set of tasks and works together to generate the final executable code.

Hopefully, this gives us a roadmap!



Quiz

What is the middle-end phase of a compiler?

- a. The phase that analyses and optimizes an abstract syntax tree or another intermediate code representation.
- b. The phase that generates machine code from the intermediate representation (IR) code.
- c. The phase that performs lexical analysis and tokenization of the source code.
- d. The phase that performs typos checking and reporting.

Quiz

What is the middle-end phase of a compiler?

- a. The phase that analyses and optimizes an abstract syntax tree or another intermediate code representation.**
- b. The phase that generates machine code from the intermediate representation (IR) code.
- c. The phase that performs lexical analysis and tokenization of the source code.
- d. The phase that performs typos checking and reporting.

Quiz

What is the purpose of instruction selection in a compiler?

- a. To perform static analysis of the program to identify possible runtime errors.
- b. To generate an optimized intermediate representation (IR) that can be used for code generation.
- c. To map high-level or intermediate-level language constructs to low-level machine instructions that can be executed by the processor.
- d. To optimize the code for specific target architectures and instruction sets.

Quiz

What is the purpose of instruction selection in a compiler?

- a. To perform static analysis of the program to identify possible runtime errors.
- b. To generate an optimized intermediate representation (IR) that can be used for code generation.
- c. To map high-level or intermediate-level language constructs to low-level machine instructions that can be executed by the processor.**
- d. To optimize the code for specific target architectures and instruction sets.

Quiz

What is register allocation in a compiler?

- a. The process of mapping variables to their literal values.
- b. The process of transforming high-level language constructs into low-level machine code.
- c. The process of selecting the best set of machine instructions to implement a program.
- d. The process of mapping variables to hardware registers in the target architecture (CPU or GPU).

Quiz

What is register allocation in a compiler?

- a. The process of mapping variables to their literal values.
- b. The process of transforming high-level language constructs into low-level machine code.
- c. The process of selecting the best set of machine instructions to implement a program.
- d. **The process of mapping variables to hardware registers in the target architecture (CPU or GPU).**

Quiz

**The compiler is a computer program, and this program has necessarily been compiled at some point.
Which language has then been used to write the C compiler?**

- a. HTML and PHP...!
- b. A language older than C like Fortran?
- c. C itself?!
- d. Machine code only?!?

Quiz

**The compiler is a computer program, and this program has necessarily been compiled at some point.
Which language has then been used to write the C compiler?**

- a. ~~HTML and PHP~~ (lol)
- b. A language older than C like Fortran?
- c. C itself?!
- d. Machine code only?!?

Quiz

The compiler is a computer program, and this program has necessarily been compiled at some point.

Which language has then been used to write the C compiler?

- a. ~~HTML and PHP (lol)~~
- b. ~~A language older than C like Fortran? (No, because it would not resolve the problem of which language was used to compile this compiler in Fortran then)~~
- c. C itself?!
- d. Machine code only?!?

Quiz

**The compiler is a computer program, and this program has necessarily been compiled at some point.
Which language has then been used to write the C compiler?**

- a. ~~HTML and PHP (lol)~~
- b. ~~A language older than C like Fortran? (No, because it would not resolve the problem of which language was used to compile this compiler in Fortran then)~~
- c. **C itself?!**
- d. Machine code only?!?

Bootstrapping

Definition (**Bootstrapping**):

The latest version of the C compiler is typically written in C itself.

The source code for latest version of the C compiler is then compiled using an older version of the compiler that is already available.

This process is called **bootstrapping**, and it allows a C compiler to be built from scratch by reusing and improving an existing compiler.

Once a basic C compiler is built, it can be used to compile other programs written in C, including newer versions of the C compiler itself with more advanced features.

Bootstrapping

Definition (**Bootstrapping**):

The latest version of the C compiler is typically written in C itself.

The source code for latest version of the C compiler is then compiled using an older version of the compiler that is already available.

This process is called **bootstrapping**, and it allows a C compiler to be built from scratch by reusing and improving an existing compiler.

Once a basic C compiler is built, it can be used to compile other programs written in C, including newer versions of the C compiler itself with more advanced features.

→ Cool, but then what was the language used for the “**first**” compiler?

Quiz

The compiler is a computer program, and this program has necessarily been compiled at some point.

Which language has then been used to write the C compiler?

- a. ~~HTML and PHP (lol)~~
- b. ~~A language older than C like Fortran? (No, because it would not resolve the problem of which language was used to compile this compiler in Fortran then)~~
- c. **C itself?!**
- d. **Machine code only?!?!**

The mother of all compilers

→ Cool, but then what was the language used for the “**first**” compiler?

- The first C compiler was developed by **Dennis Ritchie** at Bell Labs in the early 1970s (this is the same person who invented the language).
- The first C compiler was written in **assembly language**.
- After that, every time a new version of the C compiler was needed, we would use **bootstrapping**.

The mother of all compilers

This “first” compiler had several basic features:

1. It supported the basic features of the C language, including data types, control structures, and functions.
2. It produced assembly language code as output, which could be assembled into machine code using a separate assembler.
3. It used a simple two-pass compiler architecture, where the first pass performed lexical analysis and created a symbol table, and the second pass performed syntax analysis and code generation.
4. It included a rudimentary optimizer that performed simple optimizations such as constant folding and algebraic simplification.
5. It had a limited set of error messages and debugging capabilities, which made it difficult to diagnose and fix errors in the code.

The mother of all compilers

Despite its limitations, this “first” C compiler was a significant achievement because it allowed C to become a popular language for system programming and operating systems development.

The availability of a reliable and efficient C compiler helped to make C a popular choice for operating system development, as well as for writing other higher-level programming compilers/interpreters.

Learn more about Dennis Ritchie and his founding (and very much underrated) work on C and compilers:

<https://www.youtube.com/watch?v=g3jOJfrOknA>

A bit of history on compilers

References and extra reading, if any (for those of you who are curious):

- Learn more about Dennis Ritchie and his founding work on compilers:
<https://www.youtube.com/watch?v=g3jOJfrOknA>
- Refresh your skills from 50.002 Computation Structures, and see how this course will provide the missing link!
<https://istd.sutd.edu.sg/undergraduate/courses/50002-computation-structures/>
- And, more importantly,
<https://natalieagus.github.io/50002/>

(That would be your homework for this week!)