NTNU

Norwegian University of Science and Technology

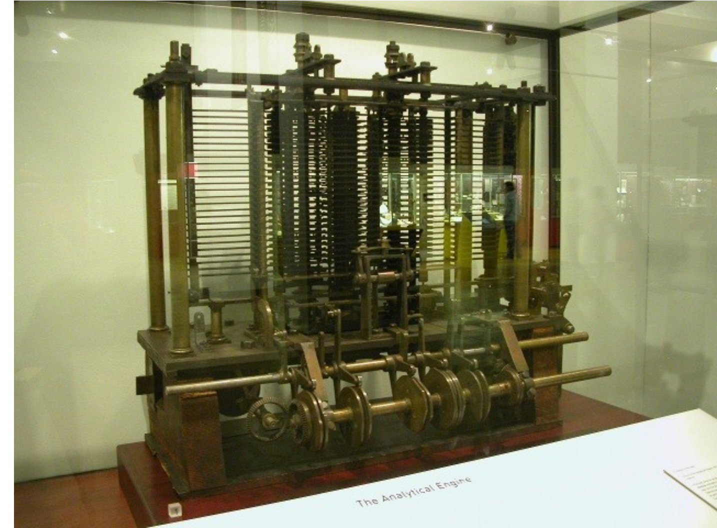
TDT4205 Overture

Hello, world

- I am Jan Christian Meyer
  (Jan.Christian.Meyer@ntnu.no, http://www.idi.ntnu.no/~janchris)
- I graduated from this very institution in 2004 (and again in 2012)
- My science is number crunching, I am not actually a compiler guru of the highest order
  (...but I use them a lot, and think they are very interesting...)
- TDT4205 track record:
  - Teaching Assistant 2006-2009
  - Assistant Prof. 2010-2011
  - Associate Prof. 2015- ?

(in other words, I can't tell you about all the latest inventions, but it will do for an introduction to our topic...)

In the beginning...



- ...there was Charles Babbage's *analytical engine*
- This doohickey was never finished, but Ada Lovelace was hired to translate a document on its construction
- She helpfully added notes on a method to make it calculate Bernoulli numbers, suggested that it might read punch cards, and be capable of composing music
- 170 years later, this sort of activity goes by the name of *programming*, that is
  - dressing up what the machine should do as some sort of calculation
  - translating it into a form where it makes the ACME device do what we want it to

Automating the manual labor

- Ada had to translate all the variables and numbers into machine state herself, and you would need to understand that mapping in order to provide input and read your answer.
- The second big idea came a century or so later, in Alan Turing's notion that a general device could be made to compute descriptions of specific ones, stored in an appropriate format.
- This raises the level of programmability, as we can hatch some systematic scheme in order to find suitable machine-friendly equivalents of vague ideas like “6”, “x”, and “finished”.
- If it's systematic enough, we can even make a program out of that scheme itself...

Providing abstractions

- That's where programming languages enter the picture.
- Quoting Babbage himself:

  “On two occasions I have been asked, – "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" […]
  I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.”

- The source of such confusion is more obvious with the complexity of more modern computers.
- Instead of turning gears, we are given little animated pixies who sometimes appear to understand which files to search for. Such illusions make it easy to forget that it's just a machine.

The first compiler

- In 1952, *Grace Murray Hopper* was working on the ENIAC system, and realized that some sequences of operations were useful in several of its programs
- This led her to develop the *A-0 programming system* ("Arithmetic language version 0")
  - Programs consisted of strings of numbers that identified locations of subroutines on storage tape, followed by their arguments
  - *i.e.* a statement like "sqrt (average (10, 15))" might have read like "231 10 15 61" if avg. and sqrt. were at tape locations 231 and 61, so that A0 could look them up and insert the logic
  - It was more like what we call a *linker* now, but it began the substitution of re-usable code words for fixed functions
  - People at the time were skeptical that a computer could even generate a program for itself to run

Prettier notations

- The possibility of substituting English-like words for numbers was not lost on people
  - Grace herself went on to work on what was to become COBOL
  - FORTRAN (“Formula Translator”) also emerged in this spirit
- The first FORTRAN compiler took 18 years of work
  - Not from start to finish, but as the time multiplied by the # of people
- More structured approaches to language description were coming out of linguistics around the same time
  - Further programming languages benefited from better definitions of words and grammar

Our first topic

- We will spend the first part of our course on this science of notations
  - How to recognize words composed of letters
  - How to recognize statements composed of words
  - How to recognize programs composed of statements
  - How to store them in a way that makes it clear how
    "sqrt (average (10, 15))" can be the same as "231 10 15 61"
    (or any other machine-specific choice of operations)
- This is a compiler's *front-end*
  - There are a bunch of formalisms and algorithms attached
  - Following those allow us to use tools that are based on them
  - It sometimes looks harder than necessary, but they help when things begin to get complicated

Machine-specific operations

- When Grace’s programming system represented its operations as tape indices, it obviously had to relate to the layout of the corresponding tapes
- Ada’s programs were similarly linked to the specific contraption, except its operations were built into nuts and bolts
- The operations *we* have are a combination:
  - Processor assembly instructions are wired in hardware
  - Operations that interact with the outside world go through the operating system

NTNU
Norwegian University of Science and Technology

Our second topic

- In order to sensibly discuss how an abstract program can be run on a physical computer, we need to dissect how these hardware and software parts interact
- This is part of a compiler's *back-end*

Automatic improvements

- In 1966, *Frances Elizabeth Allen* recognized that many useful ways to manipulate programs could be done automatically, by
  - Re-interpreting the program logic as graphs in various forms
  - Applying graph theory to those
- Apart from translating programs to machine code, people also expect a compiler to find a better translation than they could
- This is (obviously) only worth anything if the improved program still does what it was intended to do

The meaning of a program

- What a program is intended to accomplish can only be *implicitly* written into it
  - Computers can't know what "an average" is any more than a clock can know what time it is
  - Still, we expect the compiler to recognize that (a+b)/2 is the same as (a/2) + (b/2) *(...at least when it actually is :))*
- The assumptions about meaning are embedded in the definitions of our programming languages
- Compilers must be made so that they systematically *maintain* the assumptions without *understanding* them

Our third topic

- We will spend the second part of our course on identifying what kind of program ideas that can be turned into rules with no exceptions
  - It's really quite a useful list
  - Spoiler alert: it's still not quite as many as people tend to expect

Why we are here

- The illusions are so effective that we teach programming in terms of them.
  - It's perfectly possible to be a productive programmer while quietly believing that numbers and functions exist somewhere inside the box
  - In fact, accepting this sort of fairytale massively improves productivity
- In a great attack against productivity, this course is about exposing the illusion, and spoiling the trick. :)
- You can still enjoy looking at an illusion even if you know how it works – abstractions are useful tools to a programmer
- <u>Knowing what you are looking at fundamentally alters the way you see it, and that's what we are after.</u>
- In other words, even if you never touch the internals of a compiler again, there's an education in seeing how the abstractions evaporate under scrutiny.

As an aside:

(<u>**Not**</u> why we are here)

- Mildly to my regret, one recent class (ultimately) let me know that compiler construction has a reputation for being terrifyingly difficult
  - It’s easy to imagine why, the book is full of algorithms and tables and α-s and ω-s and Σ-s
  - If we were to go through this with zero tolerance for any misplaced ω, it would become a near-impossible memorization exercise
  - Let’s not do that, fear of mistakes ruins the learning process
- The point of covering the algorithms is not to recite them on command
  - It is to understand what they do, so that you can recognize them in action
- Our subject is hard in the sense that it takes a lot of patient effort
  - I maintain that it does <u>not</u> require award-winning mental powers, and offer my own lacklustre brain as evidence.

As an aside:

Why we are here

(in the auditorium)

- The book contains enough meticulous detail for a programmer's reference volume
  - Some people like to read details and gain an intuition
  - Others like to start from an intuition and read details
  - Others still jump between details and concepts in their own manner
- My goal with these lectures is to support the intuitive angle
  - I think that's how I can complement the text, nobody needs me to read aloud from the syllabus
- Hopefully, attending will save you some time and effort
  - You can judge for yourself whether or not I succeed, I am not prepared to play finger-wagging schoolmaster for an adult audience

Back on track:

What is a compiler?

- It's a program which takes another program as input, and translates it into a third program, often so that it may be run on some sort of machine.
- *i.e.*, all it does is take whatever the programmer has written in some language, and write it down in some other language.
- As alluded to, it's an automatic translator of sorts, but let's not call it that just yet.

What is an interpreter?

- It's a program which takes another program as input, and translates it into a third program, running it on some sort of machine.
- *i.e.*, all it does is take whatever the programmer has written in some language, and provide a running translation into some other language.
- This is also an automatic translator of sorts. With first appearances, the only difference from what we just called 'a compiler' is that nothing is written down, but we could surely make that happen too.
- There *is* a little more to it than that, though.

What is the difference?

- Briefly, the relation between *source and target languages.*
- The compiler has its work cut out at the time you decide that the program is ready. After it's finished, the result should be somehow self-contained with respect to the target language
- This means that no aspects of the source language are left, all behavior is expressed in the target language
- Interpreters have the luxury of knowing the state of all values and such, traded for the difficulty of not having read the entire source program before it stops.

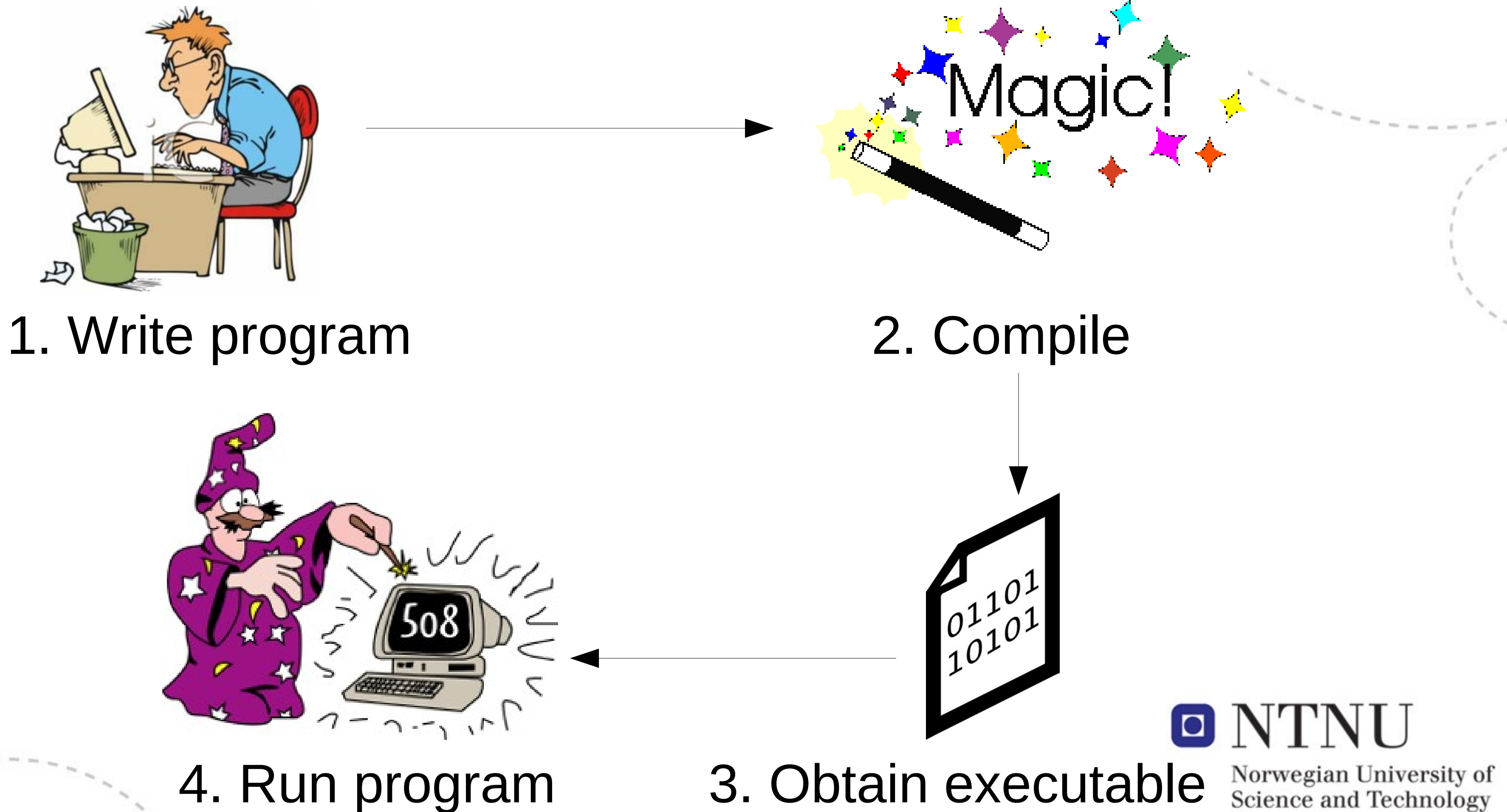
To add to the confusion

- Adding to the name salad, *translator* is a name often given to a program which converts one representation into an equivalent, other representation.
- The lines are a little fuzzy, but in general, one expects that a compiler will
  - analyze the source program
  - find some representation of its meaning
  - embed that meaning in a lower-level encoding where the concepts of the source language don't exist
- The last point defines our place in the landscape

Scope

- *Common/useful language abstractions* is a topic we cannot avoid, because we need something to compile.
  - Focus is on methodically mapping them to lower level abstractions; arguing which abstractions belong in a language, and how to use them is a topic for a class on programming language design
- *Run time systems* we must touch upon in order to formulate a translation scheme
  - Focus is on using what's provided by an operating system – deciding what's best to provide belongs to O/S and programming languages
- *Assemblers* we could do without, but using one saves us a bundle of work.
- *Linkers* and *loaders* are, again, essential trappings to get things running.
  - We will look at what they do, but not how to make them – that's the O/S again.

An iteration of the fix/compile/run cycle (as seen from far away)



Digging into step 2

- Ideally, compilation should have been the only unknown on the previous slide, but we'll have to look at how things are run in due time
- The production of an executable is a composite process in itself, though – let's start there.
- From the time you invoke "the compiler" until you get an executable, there is
  - Preprocessing
  - Lexical analysis
  - Syntax analysis, creation of intermediate representation
  - Semantic analysis
  - Lowering of intermediate representation
  - More semantic analysis / optimization
  - Code generation
  - Assembly
  - Linking

(This list is a little rough, don't memorize it as The Truth)

NTNU
Norwegian University of Science and Technology

From the top

- Preprocessing isn't that interesting – it's just substituting some text and slapping files together
- Therefore, the first salient point is *lexical analysis*

- Let us dive right in:

  When the compiler first sees a program…

  (To Be Continued)