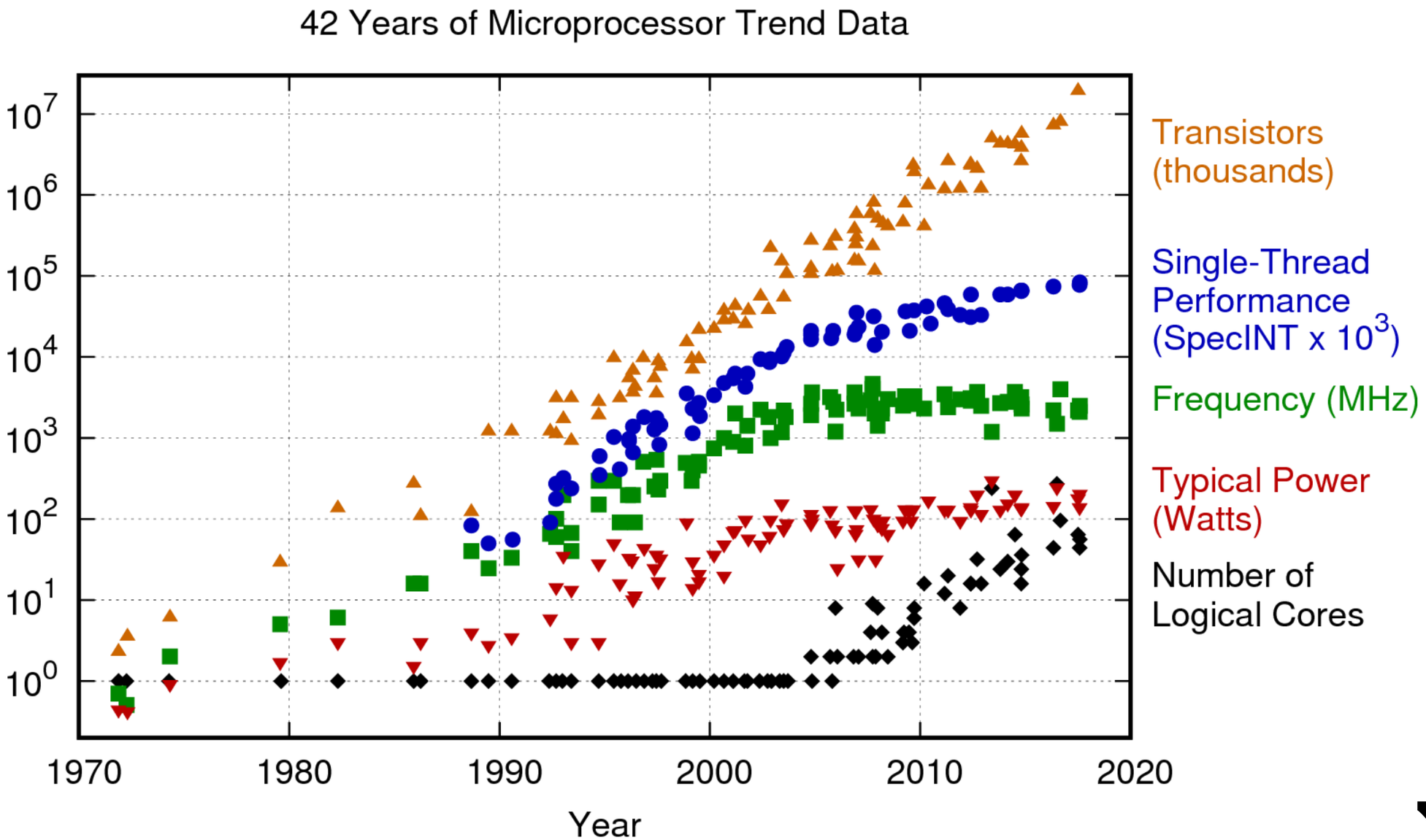
05: Scala, Spark & Big Data

2020 and beyond: advanced computing

- What languages and tools should we use in 2020 and beyond?
  - Tools that allow us to process massive amounts of data very quickly;
  - Languages that allow us to build and/or program such tools.
- Why not just use the same old tools and languages we've always used?
  - The amount of data that is typically involved in real life work is far too large to be handled with one computer.
  - Microprocessors haven't increased in raw speed in over 20 years!
  - Which is why all modern computers have multiple "cores"
  - See the next slide.

Microprocessors over the years



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

What do we notice?

- Around 2005, improvements in processors speeds started slowing down and the number of cores started increasing (from 1);
- Computers in 2020 are over a thousand times faster (single-thread speed) then they were when Java was developed in the late 80s.
  - Does this suggest anything to you?
- Let me give you a little history lesson.

Assemblers, compilers, interpreters…

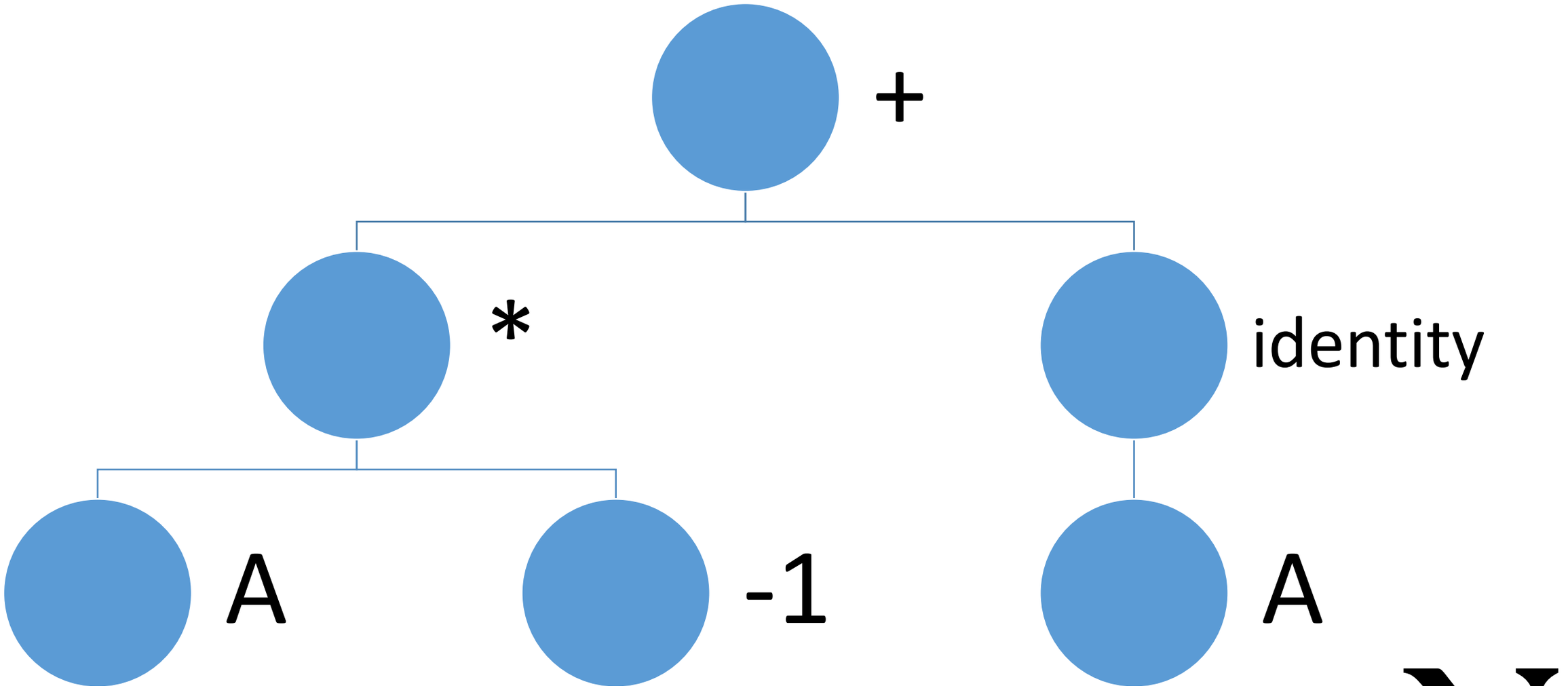
- In the early days, it was enough to be able to program a computer any way at all.
- Then, we started asking the compiler to do a bit more:
  - Figure out jumps based on *if, do, while, switch*, etc.;
  - Deal with the scope of variables (they weren't all global any more);
  - Handle problems (we call them exceptions now) rather than just terminating the program;
- Even so, the job of the compiler was to set up code which allowed the (fast) processor to do its thing—the compiler's job was then to just get out of the way.
  - Much of the emphasis in the 70s and 80s was in generating the correct machine instructions for the many different architectures.

Assemblers, compilers, interpreters…

- This was the world into which Java emerged.
  - Much of the effort was in making the compiler—and, in particular, the just-in-time byte code compiler (really, an interpreter)—as fast as possible.

- As compiler builders worked more and more on optimizing code, they noticed that they could get a lot of clues from the abstract syntax tree (AST) for the program.

An Abstract Syntax Tree



Optimization leads to smarter compilers

- We can see that the expression shown in the AST on the previous slide will evaluate to 0.
- Why would we want to generate code that we know is going to result in 0 anyway?
- But, in order to be sure that this evaluates to zero, we need to know a lot more:
  - All of the elements are immutable
  - All of the functions, including the identity function, are “pure” which is to say they always return the same value, given the same inputs.
  - That, in the domain to which *A* belongs (i.e. the “type”), that x + x * -1 = 0.
- In other words, once we know all that, we won’t even have to generate code to run this “program.”

So, why didn’t compilers always do this?

- Think back to that microprocessor speed slide:
  - We can do about **1000** times more work in the time that is acceptable to wait for the compiler than 40 years ago!
  - In those days, we couldn’t afford the time to do any more optimization.
  - But now, since we typically compile once for hundreds, if not thousands, of executions of a program, we should probably spend as much time on the compilation as possible.
- That’s the philosophy of functional programming and Scala, in particular.

Back to dealing with today's problems

- Massive amounts of data
  - But, given the nature of this data, it is usually possible to parallelize the processing of it because, typically, chunks of data are *independent*.
- Many concurrent inputs to a program:
  - Databases, users, web services.
  - It's really important that we design for multiple, asynchronous threads.
- Scala (**Sca**lable **La**nguage) is a language that is designed for these situations.
- And (Apache) Spark is a high performance computing platform which is built with Scala to, as much as possible, insulate programmers from the intricacies of parallelization and concurrency.
- Whether you're someone who thinks that Java and Python will eventually catch up to Scala, it doesn't matter. You need to understand how **functional programming** works because it is
  - **THE WAY OF THE FUTURE**.