Static Types

static types

- Some languages have static types
  - Types of key program elements — variables, functions — are known at compile time, before the program runs
  - Could be expressed directly in the program (int x)
  - Could be inferred from other parts of the program (x = 7 + 2)

- Contrast with dynamically typed languages that don't express types in the program
- Types of program elements are not determined until runtime
  - Python, Perl, LISP
  - Not the same as the language not having types!

type checking

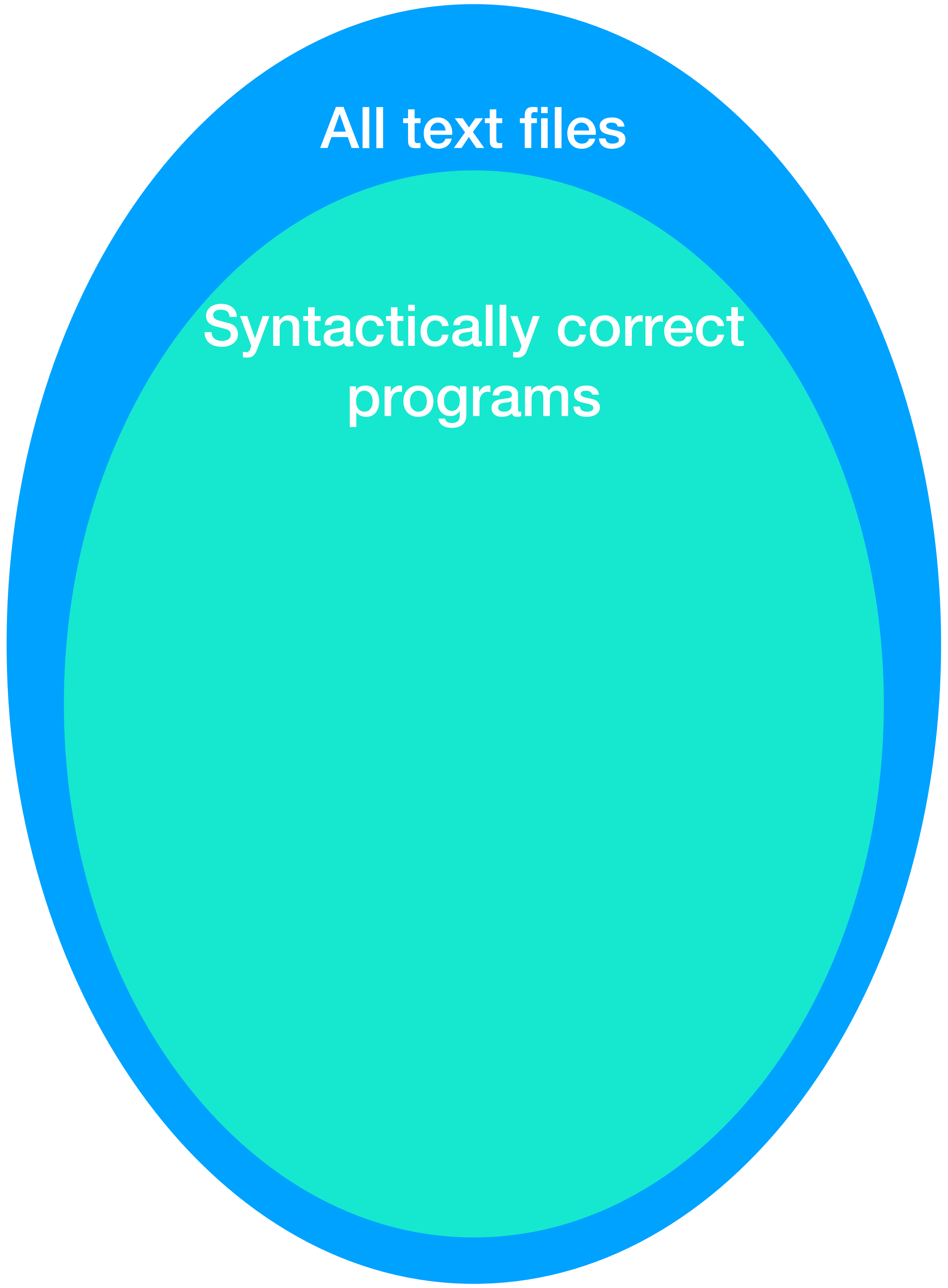
- Static types give compilers the power to do **static type checking**
  - Use type information to catch and prevent bugs
  - Intuition: prove at *compile* time that certain errors cannot occur at *run time*

- Think of this as a generalization, or more powerful version, of what we already do in our compiler

type checking

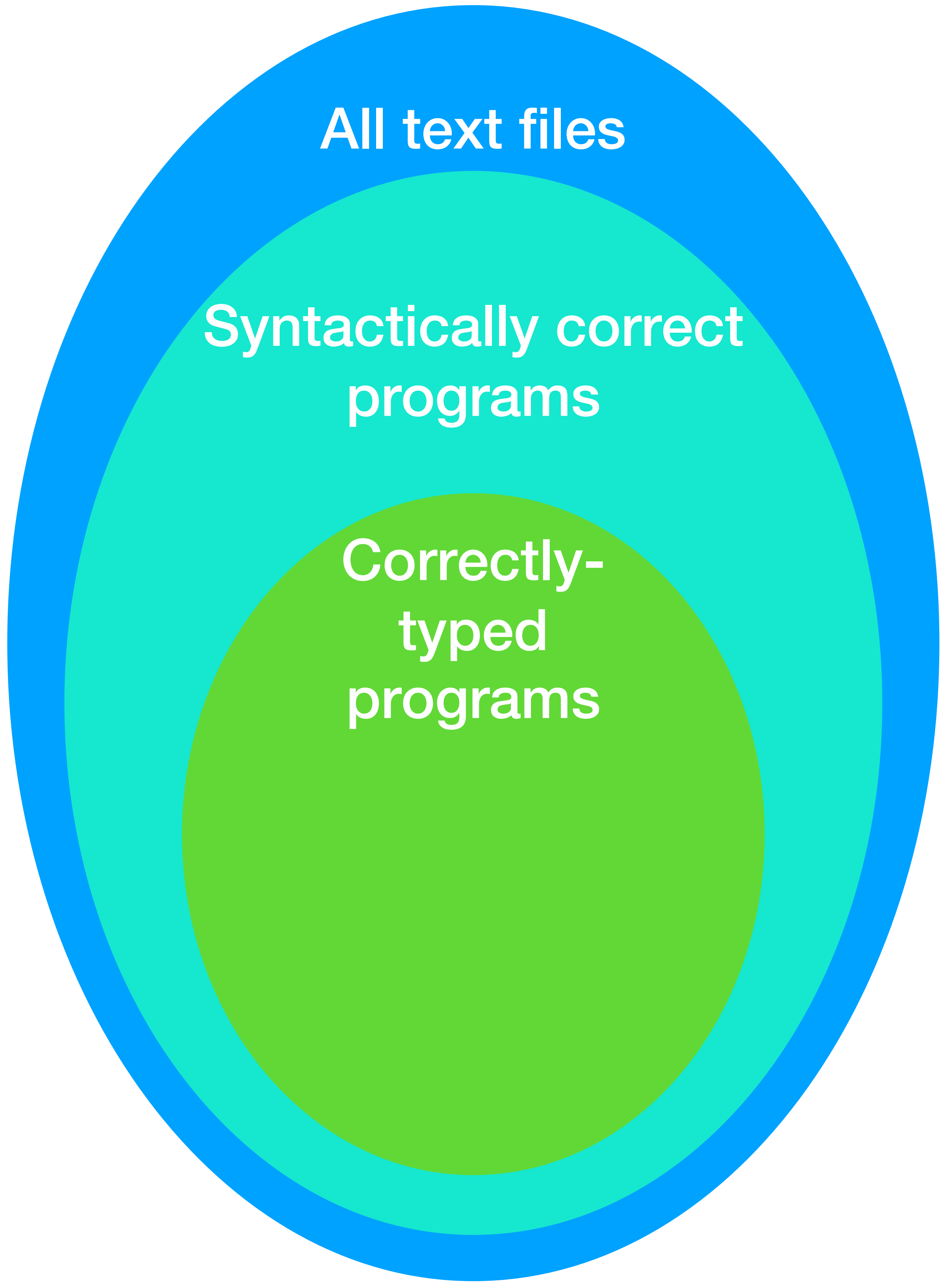
- Parsing identifies when a program is *syntactically correct*

- But syntactically correct programs can still have problems!



type checking

- Parsing identifies when a program is *syntactically correct*
- But syntactically correct programs can still have problems!
- A **correctly-typed** program obeys additional rules:
  - e.g., all arithmetic expressions use compatible types
  - e.g., all functions are called with the correct type of arguments



correctly typed ≠ correct

- Saying a program is **correctly typed** is saying something specific:
  - Certain run-time errors cannot happen

```
int main() {
  foo('a');
}

void foo(int * p) {
  print(* p);
}
```

correctly typed ≠ correct

- Saying a program is **correctly typed** is saying something specific:
  - Certain run-time errors cannot happen
- Does not mean a program is correct!
  - Other run-time errors can still happen
  - Other bugs can still happen
- Could be caught by dynamic type checks
  - e.g., Java catches null de-references

```
int main() {
  int * x = null;
  foo(x);
}

void foo(int * p) {
  print(* p);
}
```

correctly typed ≠ correct

- Saying a program is **correctly typed** is saying something specific:
  - Certain run-time errors cannot happen

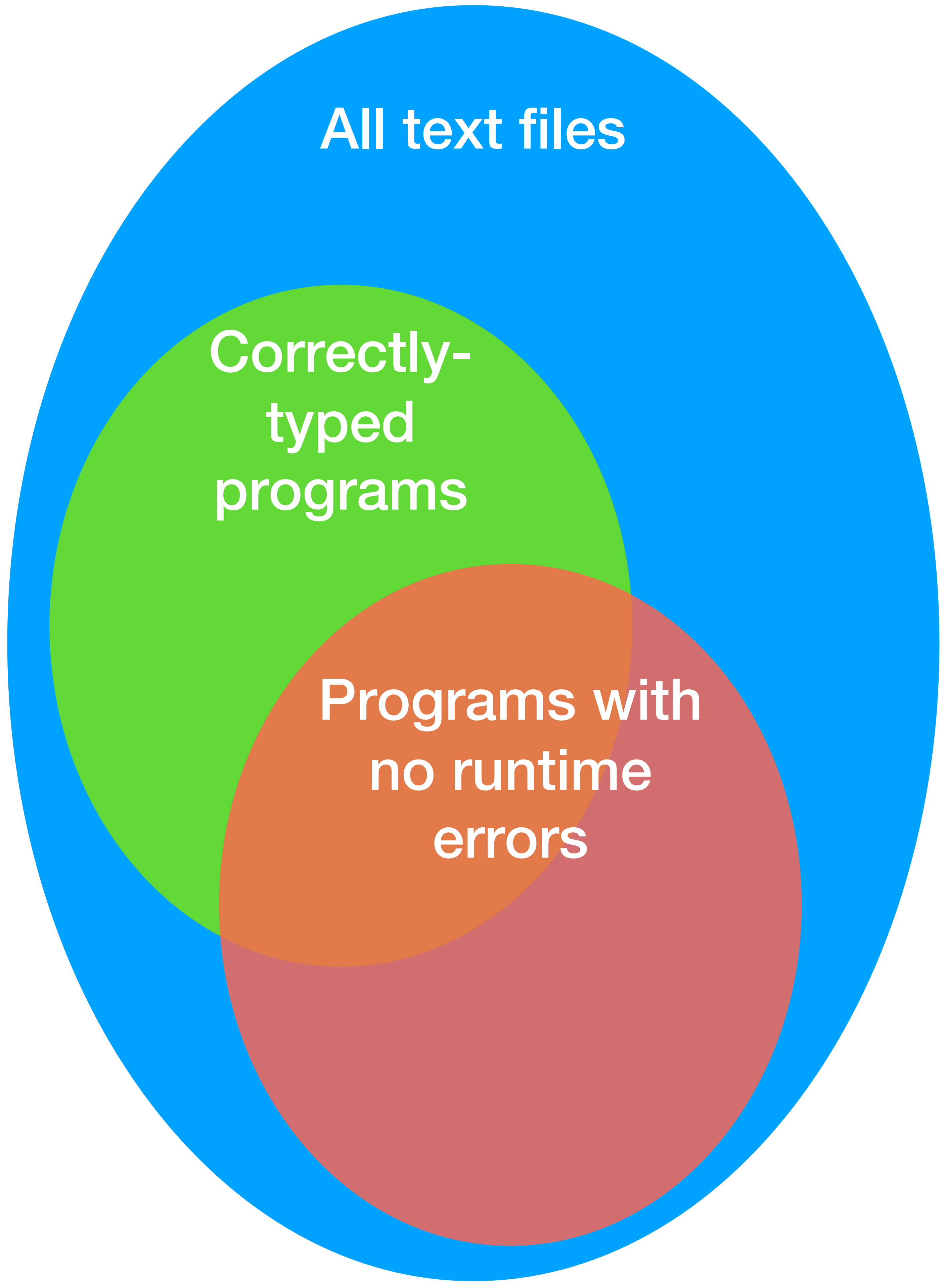
- A program that is **not** correctly typed may still be “safe”
  - If the compiler allowed it to run, it would not have a runtime error

- An equivalent Python program would not have an error

```
int main() {
  int a = 2;
  foo(‘x’, a);
}
void foo(int * p, int b) {
  if (b != 2)
    print(* p);
}
```

correctly typed ≠ correct

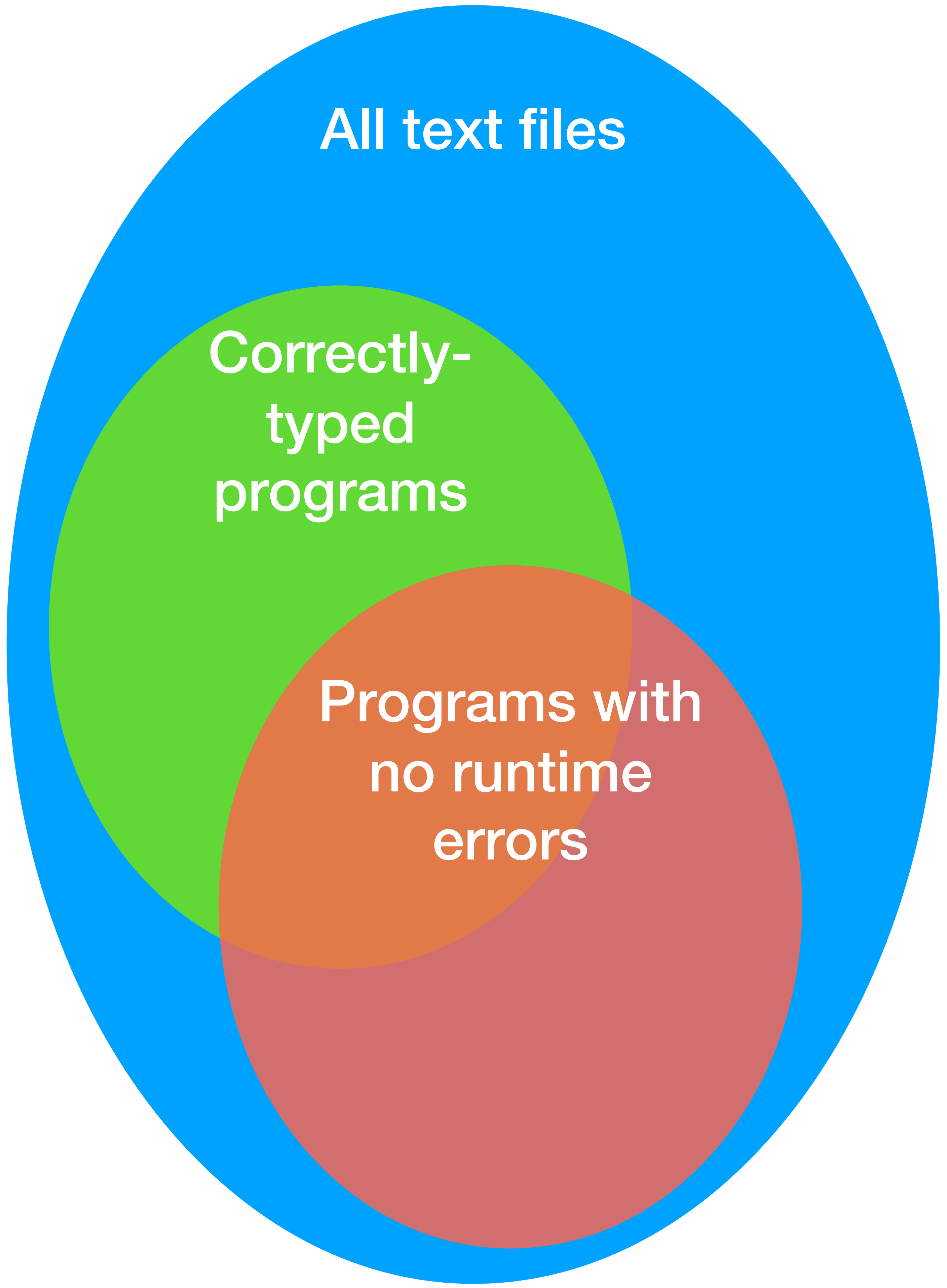
- Saying a program is **correctly typed** is saying something specific:
  - Certain run-time errors cannot happen
- A program that is **not** correctly typed may still be "safe"
  - If the compiler allowed it to run, it would not have a runtime error



correctly typed ≠ correct

- Saying a program is **correctly typed** is saying something specific:
  - Certain run-time errors cannot happen

- So why do this?
  - The set of run-time errors ruled out by a correctly typed program may be large and important!



example tradeoffs

- Different languages make different tradeoffs between static typing, dynamic typing, and no checks

- C/C++
    - Static typing ensures arithmetic operations are compatible, function calls are compatible
    - No (or very few) runtime checks for things like array out of bound, null dereferences
- Java
    - Static typing ensures arithmetic operations are compatible, function calls are compatible
    - Runtime checks ensure that array accesses are in bounds, pointers are not null

power of static types

- Static types say: **certain run time errors cannot occur**
- How do you decide?
  - Static types make stronger guarantees → fewer runtime errors can occur
  - But static type systems cannot guarantee everything!
    - Guarantee that a program terminates → not possible!

what else are static types good for?

- Help programmers structure code
  - Types provide a form of documentation
- Help IDEs work better
  - Types give IDEs more information about a program for tools like code completion
- Prove very strong properties about programs
  - The "set of values" that a type constrains can be very limiting indeed!
  - e.g., the range of an integer value (e.g. [0, 100] can be a type)
  - (Some times can even use undecidable type systems for things like theorem proving)