Lisp is easy in theory. If you go back to Lisp 1.5 (or something on that order) with no macros, it’s easy in practice as well. Lisp with macros is not quite so easy.

Smalltalk is also quite easy to parse. The language itself has extremely minimal syntax. Essentially everything is just sending messages to objects. Even (for example) creating an object is done by sending a message to an object (in case you care how the initial object go created: it simply has to pre-exist for a Smalltalk system to exist).

Pascal (the original Jensen and Wirth one) has more keywords and more punctuation. This makes both the parse and (especially) the lexer quite a bit larger—but the language is still designed specifically so it can be parsed using recursive descent, so although it takes more code than Lisp or Smalltalk, the code involved is still quite simple.

Turbo Pascal goes still further in roughly the same direction—more punctuation and keywords, so the parser gets \*bigger\*, but it’s still fundamentally fairly simple.

I won’t bore you by trying to name them all, but there are quite a few more in this direction—languages that are fairly easy to parse with relatively simple LR(0) or LALR(1) parsers that are fairly easy to write or generate.

Then we get to the ones that are harder.

Although they’re not encountered often in practice, C has a few cases where syntax alone isn’t sufficient to determine the meaning of a statement—one exact set of characters could mean either of two different things, depending on how some of the identifiers in that statement were previously defined. You must build a symbol table parse it correctly.

C++ goes quite a bit further in the same direction again. Instead of a few obscure things that almost never arise in practice requiring a symbol table to parse correctly, you run into this almost constantly. Modern C++ goes a step further though: you also need to store some state to even be able to tokenize input correctly. For example, by looking at >> in isolation, there’s no way to tell whether this is one token or two. You need to know something about the current parsing state (whether you’re parsing a template declaration) to know whether it’s one token or two.

Finally, we get to the red-headed step-grandparent of them all: Fortran. Most Fortran parsers use back-tracking, because there are points in a Fortran program when (even with full symbol information) it’s impossible to be sure what sort of statement you’re parsing until you’ve basically finished parsing it. For example, compare this:

1. DO 10 I = 1,10

…to this:

1. DO 10 I = 1.10

(note, this is fairly well-known example—pretty much a classic, as a matter of fact).

The first of these is a “DO” loop—Fortran’s equivalent of what most languages call a for loop, so that first one is roughly equivalent to `for (int i=0; i<10;i++)` in a language like C or Java.

The second is where things get hairy though. Since that’s a decimal point between the 1 and the 10, we don’t have a DO loop. What we have is a simple assignment. Specifically, it assigns the value 1.10 to a variable named `DO 10 I`.

It’s impossible to say for sure whether this is a do loop or not until we get to the second value after the comma. If there isn’t one, we re-start parsing the statement from the beginning.

Of course, that’s not the only way to do things—you can (for example) basically create two sets of parser state, and parse the statement both ways (in parallel), creating the correct state for each. Then when you’ve parsed enough to know which of those is correct, you keep the state for the branch that was correct, and discard the one for the dead branch.