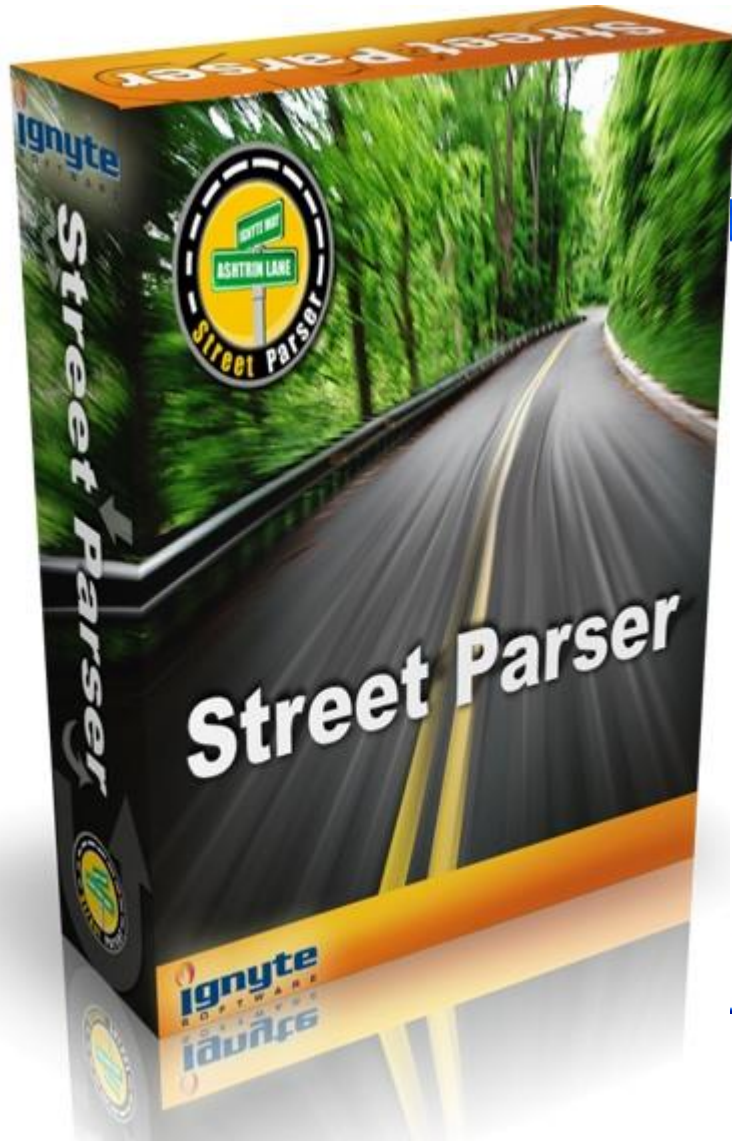


# E.g., Street Address → Standard Parts



**Address Parser Test Client**

Web Service Path

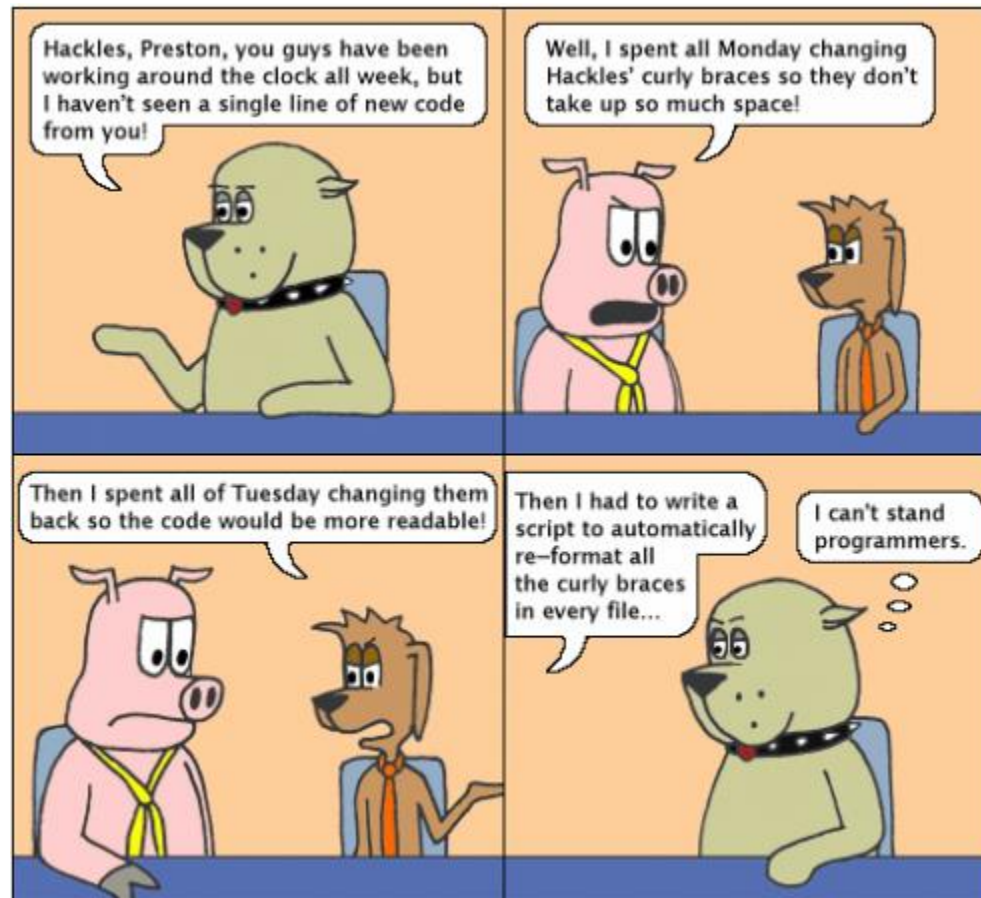
**Full Address:**

**Parsed Address**

|                                   |                                   |
|-----------------------------------|-----------------------------------|
| Street Number                     | <input type="text" value="1211"/> |
| Pre Directional                   | <input type="text" value="N"/>    |
| Street Name                       | <input type="text" value="MAIN"/> |
| Suffix                            | <input type="text" value="ST"/>   |
| Post Directional                  | <input type="text"/>              |
| Secondary Address Unit            | <input type="text" value="#B12"/> |
| Secondary Address Unit Designator | <input type="text" value="APT"/>  |

1211 NORTH MAIN STREET APT #B12

# E.g., { } → ...



# Why Study Compilers? <sub>1</sub>

- A great success story from theoretical computer science
- You may have to write a compiler or interpreter
- Parsers appear in a lot of applications
- Many applications contain little languages for customization and flexible control, e.g. word macros, scripts for graphics& animation, data layout description, ...
- Data formats are also formal languages, e.g., HTML, XML
- Translators (file converters) are often required
- Code optimization is still a challenging and active field of research
- Understanding compiler optimization can improve a programmer's code writing skills
- A programmer will eventually run into a compiler bug, limitation, or “quirk”
  - Understanding compilers will help understand what is wrong
- Besides, compilers are excellent examples of large and complex systems which can be specified rigorously, which can be implemented only by combining theory and practice

# Why Study Compilers? <sub>2</sub>

Excerpts from *Engineering a Compiler* by K.D. Cooper and L. Torczon:

**A good compiler contains a microcosm of computer science.** It makes practical application of greedy algorithms (register allocation), heuristic search techniques (list scheduling), graph algorithms (dead-code elimination), dynamic programming (instruction selection), finite automata and push-down automata (scanning and parsing), and fixed-point algorithms (data-flow analysis). It deals with problems such as dynamic allocation, synchronization, naming, locality, memory hierarchy management, and pipeline scheduling. Few software systems make as many complex and diverse components work together to achieve a single purpose. Working inside a compiler provides practical experience in software engineering that is hard to obtain with smaller, less intricate systems.

The lore of compiler construction includes many success stories. Formal language theory has led to tools that automate the production of scanners and parsers. These same tools and techniques find application in text searching, website filtering, word processing, and command-language interpreters. Type checking and static analysis apply results from lattice theory, number theory, and other branches of mathematics to understand and improve programs. Code generators use algorithms for tree-pattern matching, parsing, dynamic programming, and text matching to automatic the selection of instructions.

Building a successful compiler requires a blend of algorithms, engineering insights, and careful planning. Good compilers approximate the solutions to hard problems. They emphasize efficiency - in their own implementations and in the code they generate.

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：<https://d.book118.com/206225002212010134>