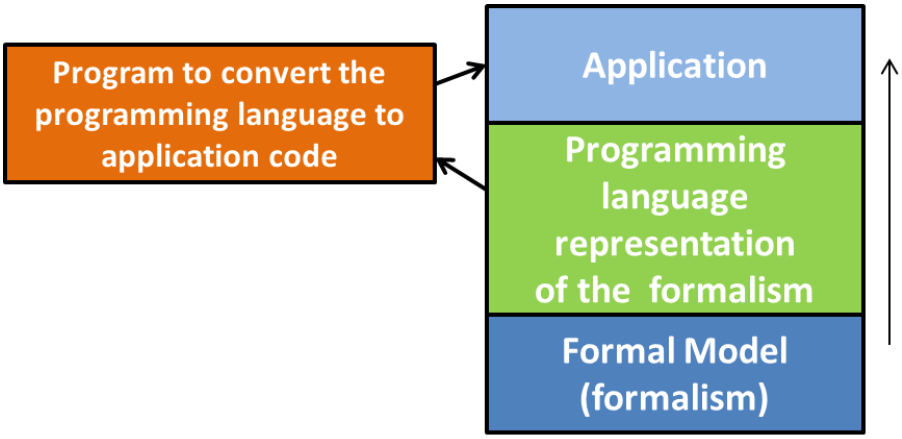
How to Build Applications with a Sound Foundation

Roger L. Costello
February 23, 2014

There are three steps to building applications with a sound foundation:

1. Create a simple, formal model (i.e., a formalism).
2. Express the formalism in a programming language.
3. Write a program which auto-generates the application from the programming language formalism.



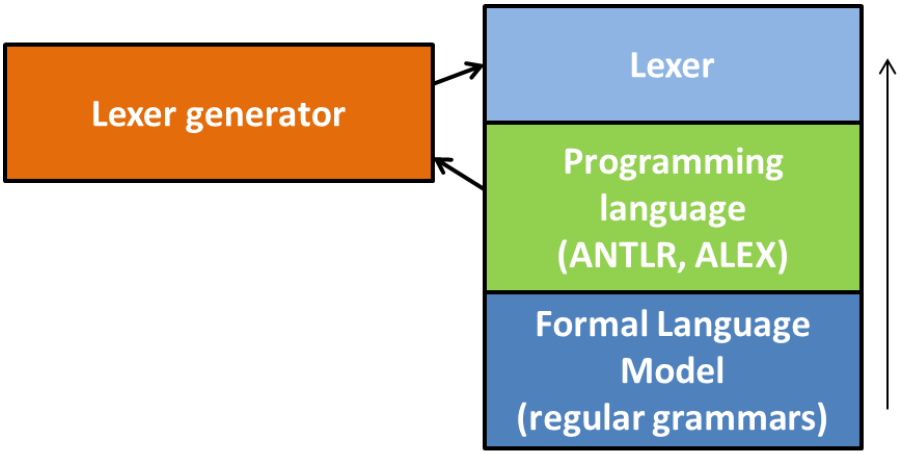
Applications created in this manner will have a sound foundation.

There are many advantages to this approach:

1. The formal model is developed clearly and thoroughly, without being burdened by irrelevant details such as programming language syntax.
2. In the diagram above, the orange box is a **program generator**. Generating programs rather than writing them by hand has numerous advantages [Grune]:
   a. The input to a program generator is of a much higher level of abstraction than the handwritten program would be. The programmer needs to specify less, and the tools take responsibility for much error-prone housekeeping. This increases the chances that the program will be correct.
   b. The use of program-generating tools allows increased flexibility and modifiability. For example, if during the design phase of a language a small change in the syntax is considered, a handwritten parser would be a major stumbling block to any such change. With a generated parser, one would just change the syntax description and generate a new parser.

c. Pre-canned or tailored code can be added to the generated program, enhancing its power at hardly any cost. For example, input error handling is usually a difficult affair in handwritten parsers; a generated parser can include tailored error correction code with no effort on the part of the programmer.
d. A formal description can sometimes be used to generate more than one type of program. For example, once we have written a grammar for a language with the purpose of generating a parser from it, we may use it to generate a syntax-directed editor.

The most mature computer science subject—compiler construction—has largely achieved its enormous success due to its judicious use of this approach. For example, lexical analyzers (lexers) are applications that have a strong formal foundation (see the discussion of regular grammars and regular expressions in books on formal languages); Java, Haskell, and many other languages provide a syntax for expressing regular grammars, and there are many tools that auto-generate lexical analyzers (see ANTLR for Java, ALEX for Haskell, and so forth).



Nobody creates lexers by hand. Developers stopped doing that 50 years ago. It's time for other branches of computer science to catch up to the maturity of compiler construction.

Recommendation

Don't create handwritten applications. Instead, do this:

- First, work with individuals (perhaps some teachers or some writers) who are able to examine a problem and see its simple underlying concepts and develop a simple formal model (formalisms).
- Next, express the formalisms in the syntax of your favorite programming language.
- Third, create a program generator that maps the programming language representation of the formalisms to application code.

As a bonus, most likely you will be able to re-use the formal model and the programming language representation with many applications.

How Did This Paper Come About?

I have always marveled at the beauty of succinct simple formal models, such as regular grammars and context-free grammars. But I never understood how to create applications built on top of formal models; that is, how to create applications with a solid foundation. As I was reading a book on compilers [Grune] the connection between formal models and applications suddenly dawned on me. Hence, this paper.

Acknowledgements

[Grune] The above list of advantages comes from the book *Modern Compiler Design* by Dick Grune et al.