IN HOLLYWOOD'S TERMS, this is the chapter where the widow in arrears is tied to
the railroad track by the heinous landlord, young Jack overtakes the onrushing
locomotive to rescue the distressed widow, gets the widow a second mortgage
on the Web, sends the villainous landlord to rehab, and organizes a men's retreat
for the boys back at the ranch.

Or, if you prefer, this is where Luke Skywalker defeats the Dark Side of the
Force and finds that dad is Darth Vader, which just goes to show you.

After much object development at the mother ship, we have arrived at the flag­
ship, and indeed the largest object in our compiler object fleet, quickBasicEngine.
A brilliant editor of mine has said, in so many words, that programmers are homeys
who be chillin' when they see code. What I think he meant was that I need to
supplement a theoretical discussion with at least a mad dash through the over­
all solution architecture of quickBasicEngine and its roughly 10,000 lines of code.
I will postpone discussion of the onboard Nutty Professor interpreter that is
included in quickBasicEngine until the next chapter. Here, I will cover the parser
algorithms as generated from the BNF definition of the language described in
Chapter 4.

This chapter will show how the parsing procedures can be manually, but
rapidly, cranked out as multiple-algorithm implementations using a simple set
of rules. You will see how the compiler generates individual instructions as
objects and how this allows us to associate as much data as is appropriate to
each instruction, including data that ties the instruction to the source code to
aid in debugging. Just because we're implementing a legacy language, there is
no reason for using retrograde methods from the dawn of man.

I will also introduce the fascinating topic of compiler optimization, demon­
strating how constant expressions are evaluated by default during parsing and
how the compiler eliminates unnecessary operations in a safe manner. Finally,
I will present an end-to-end example of the compilation and execution of a very simple program ('Hello world,' everyone's favorite).

The Recursive-Descent Algorithms

The word *Algorithms* is plural in this section's title because a different algorithm is needed for each production in the BNF. A separate algorithm must be constructed for each production in the BNF grammar. Each parser method in our compiler needs to pass a series of parameters expressing state or reference state in *Common Declarations*, and the most important single fact in the state is the position of the next token.

Before we look at the "meta" algorithm for actually coding individual recursive-descent procedures for the grammar symbols in your BNF, let's review recursive-descent algorithms in general.

Recursive-Descent Approaches

*Recursive descent* is one of the oldest parsing algorithms. It is not a compiling algorithm, per se, because it has little to do with scanning or code generation. It has to do with recognizing the language.

Hero computer scientist Niklaus Wirth, the creator of the Pascal language,¹ said that recursive descent must be used for block-structured languages such as Pascal. This is extreme, but as a manual parsing method, it is the most understandable.

Two general approaches to parsing exist: top-down and bottom-up. In the top-down, or goal-oriented, method of recursive descent, you decide on an overall goal or task and break it down into smaller tasks. In bottom-up algorithms, you instead run through the series of scanned tokens with auxiliary data structures, and enter a variety of higher states as these symbols are seen to build higher structures. Basically, in top-down algorithms, you start with *program* and go down to *token*; in bottom-up algorithms, you start with *token* and go up to *program*.

Both approaches can be automated by parser generators, but on the whole, bottom-up is better automated because of its complex data structures. Top-down recursive descent is easier to understand, and, as a tactical solution to quick parsing, it is nonpareil. So, top-down is the method used for our compiler's parsing procedures.

¹. Niklaus Wirth was an early proponent of safe, as opposed to merely efficient, computing.