LET'S BUILD A COMPILER!

By

Jack W. Crenshaw, Ph.D.

24 July 1988

Part I: INTRODUCTION

INTRODUCTION

This series of articles is a tutorial on the theory and practice of developing language parsers and compilers. Before we are finished, we will have covered every aspect of compiler construction, designed a new programming language, and built a working compiler.

Though I am not a computer scientist by education (my Ph.D. is in a different field, Physics), I have been interested in compilers for many years. I have bought and tried to digest the contents of virtually every book on the subject ever written. I don't mind telling you that it was slow going. Compiler texts are written for Computer Science majors, and are tough sledding for the rest of us. But over the years a bit of it began to seep in. What really caused it to jell was when I began to branch off on my own and begin to try things on my own computer. Now I plan to share with you what I have learned. At the end of this series you will by no means be a computer scientist, nor will you know all the esoterics of compiler theory. I intend to completely ignore the more theoretical aspects of the subject. What you *WILL* know is all the practical aspects that one needs to know to build a working system.

This is a "learn-by-doing" series. In the course of the series I will be performing experiments on a computer. You will be expected to follow along, repeating the experiments that I do, and performing some on your own. I will be using Turbo Pascal 4.0 on a PC clone. I will periodically insert examples written in TP. These will be executable code, which you will be expected to copy into your own computer and run. If you don't have a copy of Turbo, you will be severely limited in how well you will be able to

follow what's going on. If you don't have a copy, I urge you to get one. After all, it's an excellent product, good for many other uses!

Some articles on compilers show you examples, or show you (as in the case of Small-C) a finished product, which you can then copy and use without a whole lot of understanding of how it works. I hope to do much more than that. I hope to teach you HOW the things get done, so that you can go off on your own and not only reproduce what I have done, but improve on it.

This is admittedly an ambitious undertaking, and it won't be done in one page. I expect to do it in the course of a number of articles. Each article will cover a single aspect of compiler theory, and will pretty much stand alone. If all you're interested in at a given time is one aspect, then you need to look only at that one article. Each article will be uploaded as it is complete, so you will have to wait for the last one before you can consider yourself finished. Please be patient.

The average text on compiler theory covers a lot of ground that we won't be covering here. The typical sequence is:

- An introductory chapter describing what a compiler is.
- A chapter or two on syntax equations, using Backus-Naur Form (BNF).
- A chapter or two on lexical scanning, with emphasis on deterministic and non-deterministic finite automata.
- Several chapters on parsing theory, beginning with top-down recursive descent, and ending with LALR parsers.
- A chapter on intermediate languages, with emphasis on P-code and similar reverse polish representations.
- Many chapters on alternative ways to handle subroutines and parameter passing, type declarations, and such.
- A chapter toward the end on code generation, usually for some imaginary CPU with a simple instruction set. Most readers (and in fact, most college classes) never make it this far.
- A final chapter or two on optimization. This chapter often goes unread, too.

I'll be taking a much different approach in this series. To begin with, I won't dwell long on options. I'll be giving you *A* way that works. If you want to explore options, well and good … I encourage you to do so … but I'll be sticking to what I know. I also will skip over most of the theory that puts people to sleep. Don't get me wrong: I don't belittle the theory, and it's vitally important when it comes to dealing with the more tricky parts of a given language. But I believe in putting first things first. Here we'll be dealing with the 95% of compiler techniques that don't need a lot of theory to handle.

I also will discuss only one approach to parsing: top-down, recursive descent parsing, which is the *ONLY* technique that's at all amenable to hand-crafting a compiler. The other approaches are only useful if you have a tool like YACC, and also don't care how much memory space the final product uses.

I also take a page from the work of Ron Cain, the author of the original Small C. Whereas almost all other compiler authors have historically used an intermediate language like P-code and divided the compiler into two parts (a front end that produces P-code, and a back end that processes P-code to produce executable object code), Ron showed us that it is a straightforward matter to make a compiler directly produce executable object code, in the form of assembler language statements. The code will *NOT* be the world's tightest code … producing optimized code is a much more difficult job. But it will work, and work reasonably well. Just so that I don't leave you with the impression that our end product will be worthless, I *DO* intend to show you how to "soup up" the compiler with some optimization.

Finally, I'll be using some tricks that I've found to be most helpful in letting me understand what's going on without wading through a lot of boiler plate. Chief among these is the use of single-character tokens, with no embedded spaces, for the early design work. I figure that if I can get a parser to recognize and deal with I-T-L, I can get it to do the same with IF-THEN- ELSE. And I can. In the second "lesson," I'll show you just how easy it is to extend a simple parser to handle tokens of arbitrary length. As another trick, I completely ignore file I/O, figuring that if I can read source from the keyboard and output object to the screen, I can also do it from/to disk files. Experience has proven that once a translator is working correctly, it's a straightforward matter to redirect the I/O to files. The last trick is that I make no attempt to do error correction/recovery. The programs we'll be building will RECOGNIZE errors, and will not CRASH, but they will simply stop on the first error … just like good ol' Turbo does. There will be other tricks that you'll see as you go. Most of them can't be found in any compiler textbook, but they work.

A word about style and efficiency. As you will see, I tend to write programs in *VERY* small, easily understood pieces. None of the procedures we'll be working with will be more than about 15-20 lines long. I'm a fervent devotee of the KISS (Keep It Simple, Sidney) school of software development. I try to never do something tricky or complex, when something simple will do. Inefficient? Perhaps, but you'll like the results. As Brian Kernighan has said, FIRST make it run, THEN make it run fast. If, later on, you want to go back and tighten up the code in one of our products, you'll be able to do so, since the code will be quite understandable. If you do so, however, I urge you to wait until the program is doing everything you want it to.

I also have a tendency to delay building a module until I discover that I need it. Trying to anticipate every possible future contingency can drive you crazy, and you'll generally guess wrong anyway. In this modern day of screen editors and fast compilers, I don't hesitate to change a module when I feel I need a more powerful one. Until then, I'll write only what I need.

One final caveat: One of the principles we'll be sticking to here is that we don't fool around with P-code or imaginary CPUs, but that we will start out on day one producing working, executable object code, at

least in the form of assembler language source. However, you may not like my choice of assembler language … it's 68000 code, which is what works on my system (under SK*DOS). I think you'll find, though, that the translation to any other CPU such as the 80x86 will be quite obvious, though, so I don't see a problem here. In fact, I hope someone out there who knows the '86 language better than I do will offer us the equivalent object code fragments as we need them.

THE CRADLE

Every program needs some boiler plate … I/O routines, error message routines, etc. The programs we develop here will be no exceptions. I've tried to hold this stuff to an absolute minimum, however, so that we can concentrate on the important stuff without losing it among the trees. The code given below represents about the minimum that we need to get anything done. It consists of some I/O routines, an error-handling routine and a skeleton, null main program. I call it our cradle. As we develop other routines, we'll add them to the cradle, and add the calls to them as we need to. Make a copy of the cradle and save it, because we'll be using it more than once.

There are many different ways to organize the scanning activities of a parser. In Unix systems, authors tend to use getc and ungetc. I've had very good luck with the approach shown here, which is to use a single, global, lookahead character. Part of the initialization procedure (the only part, so far!) serves to "prime the pump" by reading the first character from the input stream. No other special techniques are required with Turbo 4.0 … each successive call to GetChar will read the next character in the stream.

```
#include <cstdlib>
#include <string>
#include <iostream>

// Variable Declarations
char Look;

// Report an Error
void Error(std::string s) { std::cout << "Error: " + s; }

// Report Error and Halt
void Abort(std::string s) {
    Error(s);
    std::abort();
}

// Report What Was Expected
void Expected(std::string s) { Abort(s + " Expected"); }

// Match a Specific Input Character
```

```
void Match(char x) {
    if (Look == x) {
        getchar();
    } else {
    Expected(std::to_string(x));
    }
}

// Recognize an Alpha Character

bool IsAlpha(char c) { return std::isalpha(c); }

// Recognize a Decimal Digit

bool IsDigit(char c) { return std::isdigit(c); }

// Get an Identifier

std::string GetName() {
    char c = Look;
    if (!IsAlpha(Look)) {
    Expected("Name");
    }
    getchar();
    return std::to_string(c);
}

// Get a Number

std::string GetNum() {
    char c = Look;
    if(!IsDigit(Look))
    {
        Expected("Integer");
        return "";
    }
    getchar();
    return std::to_string(c);
}

// Output a String with Tab
```

```
void Emit(std::string s)
{
    std::cout << s;
}

// Output a String with Tab and CRLF
void EmitLn(std::string s) {
    Emit(s);
    std::cout << std::endl;
}

// Initialize

void Init()
{
    getchar();
}

int main() {
    Init();
    return 0;
}
```

That's it for this introduction. Copy the code above into TP and compile it. Make sure that it compiles and runs correctly. Then proceed to the first lesson, which is on expression parsing.