LET'S BUILD A COMPILER!

By

Jack W. Crenshaw, Ph.D.

2 April 1989

Part VIII: A LITTLE PHILOSOPHY

****
*
* COPYRIGHT NOTICE
*
* Copyright (C) 1989 Jack W. Crenshaw. All rights reserved.
*
****

INTRODUCTION

This is going to be a different kind of session than the others in our series on parsing and compiler construction. For this session, there won't be any experiments to do or code to write. This once, I'd like to just talk with you for a while. Mercifully, it will be a short session, and then we can take up where we left off, hopefully with renewed vigor.

When I was in college, I found that I could always follow a prof's lecture a lot better if I knew where he was going with it. I'll bet you were the same.

So I thought maybe it's about time I told you where we're going with this series: what's coming up in future installments, and in general what all this is about. I'll also share some general thoughts concerning the usefulness of what we've been doing.

THE ROAD HOME

So far, we've covered the parsing and translation of arithmetic expressions, Boolean expressions, and combinations connected by relational operators. We've also done the same for control constructs. In all of this we've leaned heavily on the use of top-down, recursive descent parsing, BNF definitions of the syntax, and direct generation of assembly-language code. We also learned the value of such tricks as single-character tokens to help us see the forest through the trees. In the last installment we dealt with lexical scanning, and I showed you simple but powerful ways to remove the single-character barriers.

Throughout the whole study, I've emphasized the KISS philosophy ... Keep It Simple, Sidney ... and I hope by now you've realized just how simple this stuff can really be. While there are for sure areas of compiler theory that are truly intimidating, the ultimate message of this series is that in practice you can just politely sidestep many of these areas. If the language definition cooperates or, as in this series, if you can define the language as you go, it's possible to write down the language definition in BNF with reasonable ease. And, as we've seen, you can crank out parse procedures from the BNF just about as fast as you can type.

As our compiler has taken form, it's gotten more parts, but each part is quite small and simple, and very much like all the others.

At this point, we have many of the makings of a real, practical compiler. As a matter of fact, we already have all we need to build a toy compiler for a language as powerful as, say, Tiny BASIC. In the next couple of installments, we'll go ahead and define that language.

To round out the series, we still have a few items to cover. These include:

- Procedure calls, with and without parameters
- Local and global variables
- Basic types, such as character and integer types
- Arrays
- Strings
- User-defined types and structures
- Tree-structured parsers and intermediate languages
- Optimization

These will all be covered in future installments. When we're finished, you'll have all the tools you need to design and build your own languages, and the compilers to translate them.

I can't design those languages for you, but I can make some comments and recommendations. I've already sprinkled some throughout past installments. You've seen, for example, the control constructs I prefer.

These constructs are going to be part of the languages I build. I have three languages in mind at this point, two of which you will see in installments to come:

TINY - A minimal, but usable language on the order of Tiny BASIC or Tiny C. It won't be very practical, but it will have enough power to let you write and run real programs that do something worthwhile.

KISS - The language I'm building for my own use. KISS is intended to be a systems programming language. It won't have strong typing or fancy data structures, but it will support most of the things I want to do with a higher-order language (HOL), except perhaps writing compilers.

I've also been toying for years with the idea of a HOL-like assembler, with structured control constructs and HOL-like assignment statements. That, in fact, was the impetus behind my original foray into the jungles of compiler theory. This one may never be built, simply because I've learned that it's actually easier to implement a language like KISS, that only uses a subset of the CPU instructions. As you know, assembly language can be bizarre and irregular in the extreme, and a language that maps one-for-one onto it can be a real challenge. Still, I've always felt that the syntax used in conventional assemblers is dumb ... why is

```
MOVE.L A,B
```

better, or easier to translate, than

```
B=A ?
```

I think it would be an interesting exercise to develop a "compiler" that would give the programmer complete access to and control over the full complement of the CPU instruction set, and would allow you to generate programs as efficient as assembly language, without the pain of learning a set of mnemonics. Can it be done? I don't know. The real question may be, "Will the resulting language be any easier to write than assembly"? If not, there's no point in it. I think that it can be done, but I'm not completely sure yet how the syntax should look.

Perhaps you have some comments or suggestions on this one. I'd love to hear them.

You probably won't be surprised to learn that I've already worked ahead in most of the areas that we will cover. I have some good news: Things never get much harder than they've been so far. It's possible to build a complete, working compiler for a real language, using nothing but the same kinds of techniques you've learned so far. And THAT brings up some interesting questions.

WHY IS IT SO SIMPLE?

Before embarking on this series, I always thought that compilers were just naturally complex computer programs ... the ultimate challenge. Yet the things we have done here have usually turned out to be quite simple, sometimes even trivial.

For awhile, I thought is was simply because I hadn't yet gotten into the meat of the subject. I had only covered the simple parts. I will freely admit to you that, even when I began the series, I wasn't sure how far we would be able to go before things got too complex to deal with in the ways we have so far. But at this point I've already been down the road far enough to see the end of it. Guess what?

Then, I thought maybe it was because we were not generating very good object code. Those of you who have been following the series and trying sample compiles know that, while the code works and is rather foolproof, its efficiency is pretty awful. I figured that if we were concentrating on turning out tight code, we would soon find all that missing complexity.

To some extent, that one is true. In particular, my first few efforts at trying to improve efficiency introduced complexity at an alarming rate. But since then I've been tinkering around with some simple optimizations and I've found some that result in very respectable code quality, WITHOUT adding a lot of complexity.

Finally, I thought that perhaps the saving grace was the "toy compiler" nature of the study. I have made no pretense that we were ever going to be able to build a compiler to compete with Borland and Microsoft. And yet, again, as I get deeper into this thing the differences are starting to fade away.

Just to make sure you get the message here, let me state it flat out:

> USING THE TECHNIQUES WE'VE USED HERE, IT IS POSSIBLE TO BUILD A PRODUCTION-QUALITY, WORKING COMPILER WITHOUT ADDING A LOT OF COMPLEXITY TO WHAT WE'VE ALREADY DONE.

Since the series began I've received some comments from you. Most of them echo my own thoughts: "This is easy! Why do the textbooks make it seem so hard?" Good question.

Recently, I've gone back and looked at some of those texts again, and even bought and read some new ones. Each time, I come away with the same feeling: These guys have made it seem too hard.

What's going on here? Why does the whole thing seem difficult in the texts, but easy to us? Are we that much smarter than Aho, Ullman, Brinch Hansen, and all the rest?

Hardly. But we are doing some things differently, and more and more I'm starting to appreciate the value of our approach, and the way that it simplifies things. Aside from the obvious shortcuts that I outlined in Part I, like single-character tokens and console I/O, we have made some implicit assumptions and done some things differently from those who have designed compilers in the past. As it turns out, our approach makes life a lot easier.

So why didn't all those other guys use it?

You have to remember the context of some of the earlier compiler development. These people were working with very small computers of limited capacity. Memory was very limited, the CPU instruction set was minimal, and programs ran in batch mode rather than interactively. As it turns out, these caused some key design decisions that have really complicated the designs. Until recently, I hadn't realized how much of classical compiler design was driven by the available hardware.

Even in cases where these limitations no longer apply, people have tended to structure their programs in the same way, since that is the way they were taught to do it.

In our case, we have started with a blank sheet of paper. There is a danger there, of course, that you will end up falling into

traps that other people have long since learned to avoid. But it also has allowed us to take different approaches that, partly by design and partly by pure dumb luck, have allowed us to gain simplicity.

Here are the areas that I think have led to complexity in the past:

- Limited RAM Forcing Multiple Passes

  I just read "Brinch Hansen on Pascal Compilers" (an excellent book, BTW). He developed a Pascal compiler for a PC, but he started the effort in 1981 with a 64K system, and so almost every design decision he made was aimed at making the compiler fit into RAM. To do this, his compiler has three passes, one of which is the lexical scanner. There is no way he could, for example, use the distributed scanner I introduced in the last installment, because the program structure wouldn't allow it. He also required not one but two intermediate languages, to provide the communication between phases.

  All the early compiler writers had to deal with this issue: Break the compiler up into enough parts so that it will fit in memory. When you have multiple passes, you need to add data structures to support the information that each pass leaves behind for the next. That adds complexity, and ends up driving the design. Lee's book, "The Anatomy of a Compiler," mentions a FORTRAN compiler developed for an IBM 1401. It had no fewer than 63 separate passes! Needless to say, in a compiler like this the separation into phases would dominate the design.

  Even in situations where RAM is plentiful, people have tended to use the same techniques because that is what they're familiar with. It wasn't until Turbo Pascal came along that we found how simple a compiler could be if you started with different assumptions.

- Batch Processing

  In the early days, batch processing was the only choice ... there was no interactive computing. Even today, compilers run in essentially batch mode.

  In a mainframe compiler as well as many micro compilers, considerable effort is expended on error recovery ... it can consume as much as 30-40% of the compiler and completely drive the design. The idea is to avoid halting on the first error, but rather to keep going at all costs, so that you can tell the programmer about as many errors in the whole program as possible.

  All of that harks back to the days of the early mainframes, where turnaround time was measured in hours or days, and it was important to squeeze every last ounce of information out of each run.

  In this series, I've been very careful to avoid the issue of error recovery, and instead our compiler simply halts with an error message on the first error. I will frankly admit that it was mostly because I wanted to take the easy way out and keep things simple. But this approach, pioneered by Borland in Turbo Pascal, also has a lot going for it anyway. Aside from keeping the compiler simple, it also fits very well with the idea of an interactive system. When

compilation is fast, and especially when you have an editor such as Borland's that will take you right to the point of the error, then it makes a lot of sense to stop there, and just restart the compilation after the error is fixed.

o Large Programs

Early compilers were designed to handle large programs ... essentially infinite ones. In those days there was little choice; the idea of subroutine libraries and separate compilation were still in the future. Again, this assumption led to multi-pass designs and intermediate files to hold the results of partial processing.

Brinch Hansen's stated goal was that the compiler should be able to compile itself. Again, because of his limited RAM, this drove him to a multi-pass design. He needed as little resident compiler code as possible, so that the necessary tables and other data structures would fit into RAM.

I haven't stated this one yet, because there hasn't been a need ... we've always just read and written the data as streams, anyway. But for the record, my plan has always been that, in a production compiler, the source and object data should all coexist in RAM with the compiler, a la the early Turbo Pascals. That's why I've been careful to keep routines like GetChar and Emit as separate routines, in spite of their small size. It will be easy to change them to read to and write from memory.

o Emphasis on Efficiency

John Backus has stated that, when he and his colleagues developed the original FORTRAN compiler, they KNEW that they had to make it produce tight code. In those days, there was a strong sentiment against HOLs and in favor of assembly language, and efficiency was the reason. If FORTRAN didn't produce very good code by assembly standards, the users would simply refuse to use it. For the record, that FORTRAN compiler turned out to be one of the most efficient ever built, in terms of code quality. But it WAS complex!

Today, we have CPU power and RAM size to spare, so code efficiency is not so much of an issue. By studiously ignoring this issue, we have indeed been able to Keep It Simple. Ironically, though, as I have said, I have found some optimizations that we can add to the basic compiler structure, without having to add a lot of complexity. So in this case we get to have our cake and eat it too: we will end up with reasonable code quality, anyway.

o Limited Instruction Sets

The early computers had primitive instruction sets. Things that we take for granted, such as stack operations and indirect addressing, came only with great difficulty.

Example: In most compiler designs, there is a data structure called the literal pool. The compiler typically identifies all literals used in the program, and collects them into a single data structure. All references to the literals are done indirectly to this pool. At the end of the compilation, the compiler issues commands to set aside storage and initialize the literal pool.

We haven't had to address that issue at all. When we want to load a literal, we just do it, in line, as in

```
MOVE #3,D0
```

There is something to be said for the use of a literal pool, particularly on a machine like the 8086 where data and code can be separated. Still, the whole thing adds a fairly large amount of complexity with little in return.

Of course, without the stack we would be lost. In a micro, both subroutine calls and temporary storage depend heavily on the stack, and we have used it even more than necessary to ease expression parsing.

- Desire for Generality

  Much of the content of the typical compiler text is taken up with issues we haven't addressed here at all ... things like automated translation of grammars, or generation of LALR parse tables. This is not simply because the authors want to impress you. There are good, practical reasons why the subjects are there.

  We have been concentrating on the use of a recursive-descent parser to parse a deterministic grammar, i.e., a grammar that is not ambiguous and, therefore, can be parsed with one level of lookahead. I haven't made much of this limitation, but the fact is that this represents a small subset of possible grammars. In fact, there is an infinite number of grammars that we can't parse using our techniques. The LR technique is a more powerful one, and can deal with grammars that we can't.

  In compiler theory, it's important to know how to deal with these other grammars, and how to transform them into grammars that are easier to deal with. For example, many (but not all) ambiguous grammars can be transformed into unambiguous ones. The way to do this is not always obvious, though, and so many people have devoted years to develop ways to transform them automatically.

  In practice, these issues turn out to be considerably less important. Modern languages tend to be designed to be easy to parse, anyway. That was a key motivation in the design of Pascal. Sure, there are pathological grammars that you would be hard pressed to write unambiguous BNF for, but in the real world the best answer is probably to avoid those grammars!

  In our case, of course, we have sneakily let the language evolve as we go, so we haven't painted ourselves into any corners here. You may not always have that luxury. Still, with a little care you should be able to keep the parser simple without having to resort to automatic translation of the grammar.

We have taken a vastly different approach in this series. We started with a clean sheet of paper, and developed techniques that work in the context that we are in; that is, a single-user PC with rather ample CPU power and RAM space. We have limited ourselves to reasonable grammars that are easy to parse, we have used the instruction set of the CPU to advantage, and we have not concerned ourselves with efficiency. THAT's why it's been easy.

Does this mean that we are forever doomed to be able to build only toy compilers? No, I don't think so. As I've said, we can add certain optimizations without changing the compiler structure. If we want to process large files, we can always add file buffering to do that. These things do not affect the overall program design.

And I think that's a key factor. By starting with small and limited cases, we have been able to concentrate on a structure for the compiler that is natural for the job. Since the structure naturally fits the job, it is almost bound to be simple and transparent. Adding capability doesn't have to change that basic structure. We can simply expand things like the file structure or add an optimization layer. I guess my feeling is that, back when resources were tight, the structures people ended up with were artificially warped to make them work under those conditions, and weren't optimum structures for the problem at hand.

CONCLUSION

Anyway, that's my arm-waving guess as to how we've been able to keep things simple. We started with something simple and let it evolve naturally, without trying to force it into some traditional mold.

We're going to press on with this. I've given you a list of the areas we'll be covering in future installments. With those installments, you should be able to build complete, working compilers for just about any occasion, and build them simply. If you REALLY want to build production-quality compilers, you'll be able to do that, too.

For those of you who are chafing at the bit for more parser code, I apologize for this digression. I just thought you'd like to have things put into perspective a bit. Next time, we'll get back to the mainstream of the tutorial.

So far, we've only looked at pieces of compilers, and while we have many of the makings of a complete language, we haven't talked about how to put it all together. That will be the subject of our next two installments. Then we'll press on into the new subjects I listed at the beginning of this installment.

See you then.

****
*
* COPYRIGHT NOTICE
*
* Copyright (C) 1989 Jack W. Crenshaw. All rights reserved.
*
****