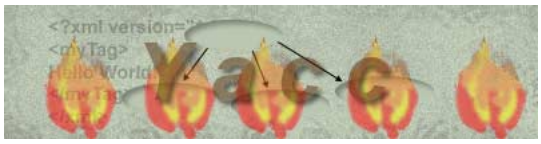
# Why Bison is Becoming Extinct

by **John Aycock**




## Introduction

At some point in your career, you're going to implement a computer language. You probably won't be implementing Java or C++. You may not even recognize it as a language. Truth be told, there are an awful lot of domain-specific languages, or "little languages" [7] in common use:

- configuration files,
- HTML/XML documents,
- shell scripts,
- network protocols,
- mail headers,
- command-line arguments.

The list goes on. A number of programs allow you to write scripts to control their operation; in fact, just the other day I downloaded a neural network simulator which provided a little programming language to steer the simulation.

How will you implement your language? There's the *ad hoc* approach, of course, but it's not well suited to languages whose design is complex or frequently changing. You also end up writing code to perform tasks which can be effectively automated.

You might also consider using existing languages like Tcl [**18**] and Python [**6**]. These languages are designed to either be embedded in an existing application, or easily extended. This is a good solution when it can be used, saving a lot of time and effort. However, there may be concerns about tying your language to one which is itself changing, or the syntax and semantics of your language may not match those of such a "host" language.

A third approach is to use compiler tools to implement your language. Most were designed for the implementation of large programming languages, but the same principles and techniques apply equally well to little languages. This article is the story of one such tool -- a parser generator tool -- and more importantly, what sort of tool is going to replace it, and why.

## The Role of the Parser

Parsing is typically the second phase of compilation. First, a scanner performs lexical analysis on the input, breaking it up into tokens; this is roughly equivalent to dividing a sentence into words. Then the parser, purveyor of the dreaded syntax error, looks at the stream of tokens to verify its syntactic correctness.

The parser checks syntax using a set of grammar rules that have been supplied by the language implementer. (To be precise, I'm referring to "context-free" grammar rules here.) If the rules can be applied in a way so as to derive the input, then the input is syntactically correct. It still may have no coherent meaning, but at least the syntax is valid.

**Figure 1** shows a grammar for simple arithmetic expressions. A good way to think about grammars is as a set of substitution rules: the symbol on the left-hand side of the arrow is a variable, which can be replaced with whatever there is on the right-hand side [**21**]. **Figure 2** gives a sequence of substitutions that derive the input `n + n`. In a sense, a parser must work backwards, because its input is the last step of this substitution sequence (in this case, `n + n`) and the parser must divine the sequence of substitutions that led up to it.

start → expression
expression → expression + term
expression → term
term → term * factor
term → factor
factor → n

**Figure 1:** Expression grammar

start ⇒ expression
⇒ expression + term
⇒ term + term
⇒ factor + term
⇒ n + term
⇒ n + factor
⇒ n + n

**Figure 2:** A derivation

Parsing is a *very* well-studied topic, and there are many excellent books on the subject ranging from the compiler-oriented [**1**, **9**] to the theoretical [**2**] to the encyclopedic [**12**]. Why has so much work been done on it? It turns out that there isn't just one parsing algorithm, but dozens. If you think of the set of context-free grammars as a pie, then each parsing algorithm is limited to working on grammars that belong to a particular slice of that pie. A lot of compiler research in the 1970s, give or take a decade, was devoted to slicing that pie up in various creative ways, combining one piece of pie with the next piece, stapling this piece to that piece, and so on. You get the idea.

The point of this culinary exercise was to devise a parsing algorithm which was the perfect trade-off between expressive power, speed of execution, and space consumption. Keep in mind that, at the time, the cost of computing resources was high relative to programmer cost. It was perfectly acceptable to expect you, as a programmer, to spend time transforming your grammar into a form palatable for an efficient parsing algorithm. It was in this environment that Yacc was born.

## Enter Yacc

Yacc [**14**,**16**] stands for "Yet Another Compiler-Compiler" and is what would now be called a parser generator. It takes a grammar as input, and produces C code which implements a parser for that grammar. As a language implementer using Yacc, you do two things: you provide the input grammar, and you take Yacc's C code output and compile and link it in with your code. *Voila*! Instant parser!

Yacc emerged from Bell Labs in the mid-1970s, and is indisputably the most successful parser generator tool available. Numerous variants have been produced, like Berkeley Yacc and the Free Software Foundation's Bison, and Yacc-like tools can be found for most languages (e.g., the **CUP** parser generator for Java). I'll use the term "Yacc" to mean any of this family of tools.

Internally, Yacc uses an LALR(1) parsing algorithm, along with some tweaks which let it accept a slightly larger class of grammars. If you don't know what this means, you're not alone. The problem is, you *must* know about this algorithm in order to successfully use Yacc. Consider, for example, you want to implement a simple assembly language, where each line has at most one optional label, one opcode, and one optional operand. An assembly program would look something like this:

```
loop:  lda   foo
       shift bar
       halt
```

An obvious approach would be to start at a high level and decompose the design, using a grammar like the one in

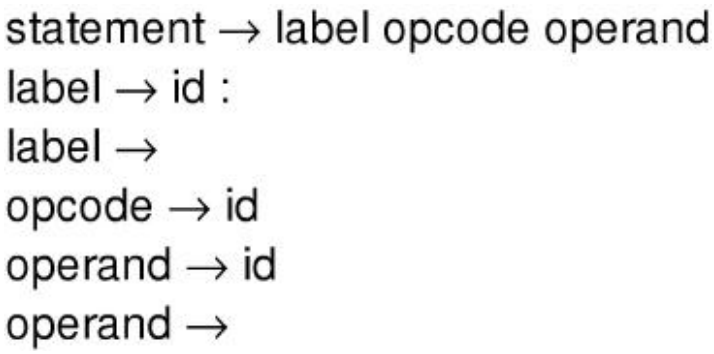



**Figure 3:** Assembly language grammar

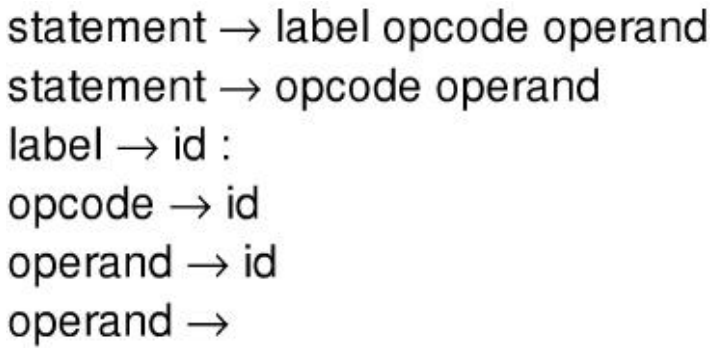



**Figure 4:** Yet another assembly language grammar

This grammar says that a line has a label, an opcode, and an operand; that a label is optional and can be absent is represented by the label rule with an empty right-hand side. The same device is used for operands. This grammar correctly describes the example assembly language, but for your efforts Yacc would reward you with the error message

```
example.y contains 1 shift/reduce conflict
```

What does this mean? The answer relates back to that whole LALR(1) thing from before -- essentially, the algorithm is unable to choose between several conflicting actions. To make the long story short, if you rewrite your grammar to the slightly-altered version shown in **Figure 4**, Yacc will accept it without complaint. But this isn't quite the final goal: Yacc has produced a parser which does nothing besides complaining bitterly when a syntax error is found. To make it really useful, you need to add semantic actions.

Semantic actions are bits of code which you can associate with grammar rules. When the parser determines that a grammar rule has been used, it will execute the associated semantic action. In a compiler, you might use these actions to build an internal representation of the input, perform semantic checking, emit code, or anything else you can imagine.

Internally, Yacc implements semantic actions by modifying the grammar. Your grammar. The one you painstakingly transformed so that Yacc wouldn't complain about it in the first place. Add a semantic action in the wrong place and you're right back to

```
example.y contains 1 shift/reduce conflict
```

This example brings out some major flaws in Yacc as a tool:

- A knowledge of Yacc's internal algorithm is necessary for use. This locks out an entire class of casual users.
- Modifying the grammar to make it acceptable to Yacc introduces the potential for error. Are you really sure that your modified grammar describes the same language as before?
- Maintenance issues are also introduced by grammar modification, because the modified grammar resembles the original grammar less and less.
- Productivity is adversely affected as a result of spending time struggling with a tool.

**Figure 5** gives an overview of the Yacc development cycle. To be fair, Yacc does provide a debugging mechanism. The LALR(1) algorithm is based on finite state automata, and Yacc is more than happy to give you an ASCII rendition of the automaton in order to further your debugging efforts.

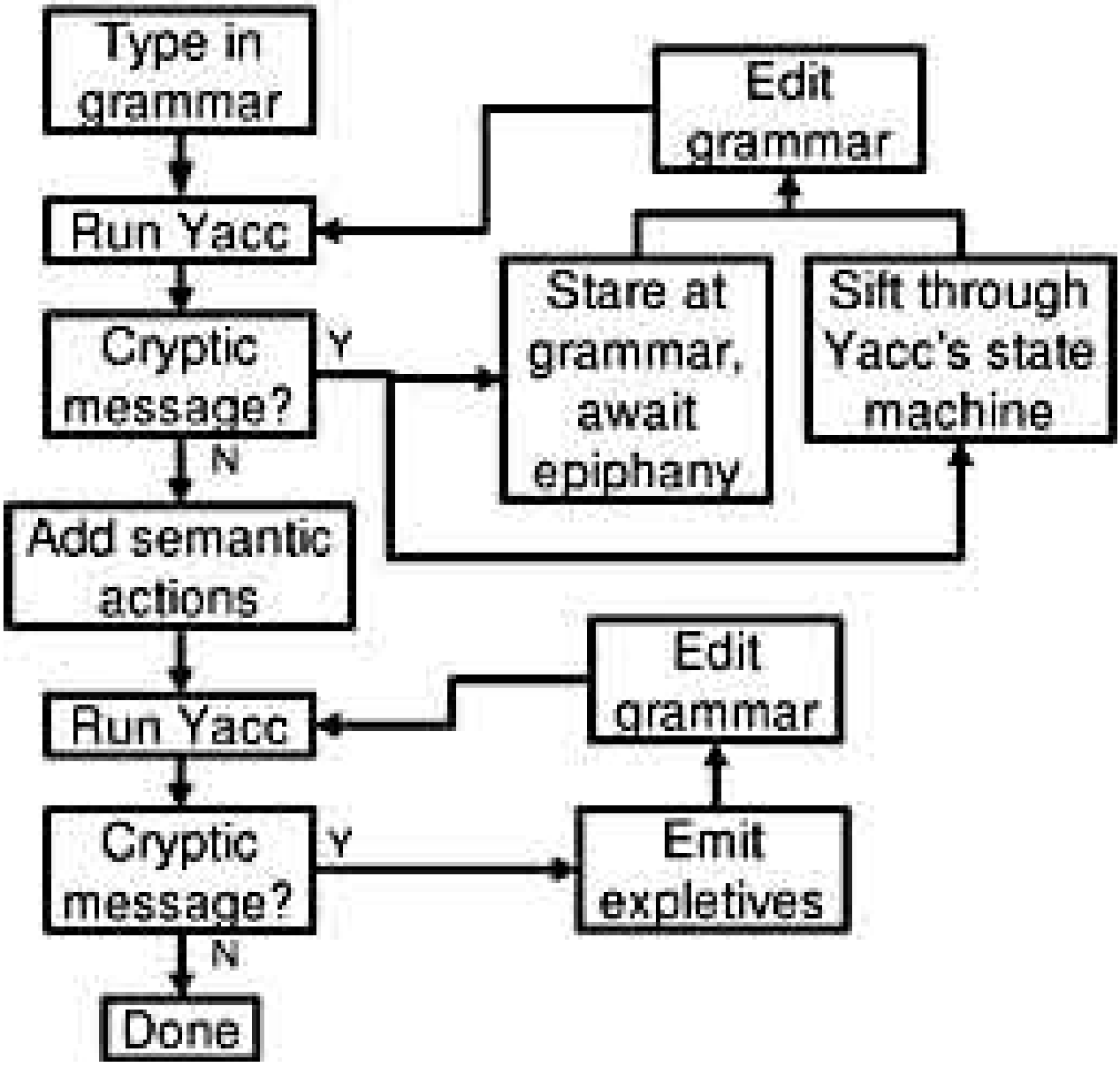



**Figure 5: Yacc development cycle**

At this point, you've expended an awful lot of time and required an awful lot of knowledge about how Yacc works in order to get it to accept your grammar. Computing conditions have changed since the 1970s -- is Yacc really still an effective tool?

## Beyond Yacc

To help answer this question, let's imagine a tool that places no restrictions on the grammar we give it. What things can be done with such a tool?

For starters, a lot of common programming languages are specified using ambiguous grammars, which are beyond the capabilities of the LALR(1) algorithm used by Yacc. This includes languages like C, C++, and Perl. As I mentioned in the introduction, you may not be implementing a full compiler for the language. But maybe you want to write a tool which extracts variable declarations, or pretty-prints

source code, or measures software quality metrics. All these jobs can be done using parsing tools.

A lot of domain-specific languages have *ad hoc* designs which have accumulated features over the years, and have no nice, simple Yacc-compatible grammar which describes them. This is another instance where a more powerful parsing tool can help. (Ironically, Yacc's own input files are best parsed with a more powerful parser than Yacc [**14**].)

Parsing can also be seen as a form of pattern matching. This has been exploited for code generation in compilers: a (highly ambiguous) grammar specifies what patterns to look for in the input program, and semantic actions emit code when executed [**10**]. I've used a similar idea to decompile Python byte-codes back into source code. I wouldn't have even attempted it with Yacc.

Finally, ambiguous grammars can be used in software re-engineering [**20**]; being able to easily parse legacy programs is very useful in this field. The problem re-engineers face is, the correct answer to "can you parse this COBOL program?" is "yes, but which dialect of COBOL are you talking about?" With an ambiguous grammar you can either specify multiple, possibly conflicting dialects of a language, or have a core grammar specification with which you compose dialect-specific grammar rules as needed.

Oh, and either assembly language grammar works without modification.

## The Case for General Parsing Algorithms

How can general grammars be parsed? In some cases, the above grammars can be mangled into a form that Yacc will accept, using various tricks. But then you're really no further ahead than before.

As it happens, you can have your cake and eat it, too. Or pie, in this case. There are a few parsing algorithms which are able to handle all context-free grammars, even ambiguous ones. Two practical ones are Earley parsing [**8**] and generalized LR (or GLR) parsing [**19**]. (The latter is sometimes called "Tomita" parsing, but it is based on a much earlier idea by Lang [**15**].)

Curiously enough, the Earley and GLR parsing algorithms come from that same time period as Yacc and company: the 1970s. The major reason they didn't take over the world back then was not a question of functionality, but of performance. It will come as little surprise that a general algorithm is slower than a much more specialized one. The worst-case time complexity of Earley's algorithm is $O(n^3)$, where $n$ is the length of the input [**8**], and the worst case for GLR parsing is even more horrifying.

Such bounds as mentioned above are, however, for the absolute worst cases. Feed a general parsing algorithm the meanest, ugliest ambiguous grammar you can think of, and it will, no doubt, take a while to mull it over. The usual case, though, is quite different. For most practical grammars, like those of programming languages, the general parsing algorithms mentioned here give linear performance, just like Yacc.

Complexity analysis abstracts away a lot of detail, however. In practice, general parsing algorithms have far more bookkeeping overhead than specialized ones, and run that much more slowly [12]. There are two reasons to still consider using a general parsing algorithm, though:

1. Current research is making these algorithms faster [3,4, 5,11, 17]. My colleague and I have recently made an Earley parser run in time comparable to Bison.
2. This isn't the 1970s! Your time and productivity cost far more than any computer. What are you saving those CPU cycles for?

## Your Next Parsing Tool

There are currently a handful of freely-available tools which employ general parsing algorithms:

- **ACCENT**, an Earley parser generator tool along the lines of Yacc.
- **SPARK**, the Scanning, Parsing, And Rewriting Kit. Provides support for compilation of little, domain-specific languages in Python, although it has been used for larger projects. Uses Earley's algorithm for parsing.
- **ASF+SDF**, a suite of tools for language processing. Uses GLR parsing.

The fact that each of the above must provide some mechanism for managing ambiguity in grammars makes them noticeably different from the Yacc group. In other words, when there are multiple ways to interpret the input, there must be some way of telling the tool which one you want. Unfortunately, it's not possible to tell if an arbitrary grammar is ambiguous [13], so you'll have to rely on testing to flush out problems. I haven't found this to be a big problem in practice.

Another thing you'll find different, especially if you're used to more traditional parsing tools like Yacc, is how easy tools based on general parsing algorithms are to use. You put in your grammar and **it just works**. No fiddling around with the grammar. Just plunk it in and you're moving on to the next task. More like how a good tool is supposed to work.

## Conclusion

The moral of the story is loud and clear. Tools designed for one era of computing may not be the best choice for another. The selection of a tool is a choice that must be periodically evaluated. In this particular instance, it makes sense to begin moving away from older tools such as Yacc, with their specialized parsing algorithm. Parser tools using general parsing algorithms allow easier development and simpler maintenance, eliminate a class of errors, and empower a class of users.

Two hundred years ago, **bison**, a.k.a. buffalo, roamed North America in massive herds. They were

hunted to near extinction by arriving settlers. In contrast, the coming extinction of Bison, a.k.a. Yacc, is no cause for remorse; it is simply a matter of choosing more powerful, effective tools.

## Acknowledgments

Nigel Horspool and Shannon Jaeger made helpful comments on a draft of this article, as did Amie Souter and the anonymous reviewers.

## References

**1**

Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

**2**

Aho, A.V., and Ullman, J.D. *The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing*. Prentice-Hall, 1972.

**3**

Aycock, J., and Horspool, N. Faster generalized LR parsing. In *Proceedings of the 8th International Conference on Compiler Construction (LNCS #1575)*. Springer-Verlag, 1999, pp. 32-46.

**4**

Aycock, J., and Horspool, N. Directly-executable Earley parsing. In *Proceedings of the 10th International Conference on Compiler Construction (LNCS #2027)*. Springer-Verlag, 2001, pp. 229-243.

**5**

Aycock, J., Horspool, N., Janousek, J., and Melichar, B. Even faster generalized LR parsing. To appear in *Acta Informatica*.

**6**

Beazley, D. *Python Essential Reference*. New Riders, 1999.

**7**

Bentley, J. Little languages. In *More Programming Pearls*. Addison-Wesley, 1988, pp. 83-100.

**8**

Earley, J. An efficient context-free parsing algorithm. *Communications of the ACM* 13, 2 (Feb. 1970), 94-102.

**9**

Fischer, C.N., and LeBlanc Jr., R.J. *Crafting a Compiler*. Benjamin/Cummings, 1988.

**10**

Glanville, R.S., and Graham, S.L. A new method for compiler code generation. In *5th Annual ACM Symposium on Principles of Programming Languages*. ACM/SIGPLAN, 1978, pp. 231-240.

**11**

Graham, S.L., Harrison, A.M., and Ruzzo, W.L. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems* 2, 3 (1980), 415-462.

**12**

Grune, D., and Jacobs, C.J.H. *Parsing Techniques: A Practical Guide*. Ellis Horwood, 1990.

**13**

Hopcroft, J.E., and Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

**14**

Johnson, S.C. YACC -- yet another compiler compiler. *UNIX Programmer's Manual, 7th Edition*, 2B, 1978.

**15**

Lang, B. Deterministic techniques for efficient non-deterministic parsers. In *Automata, Languages, and Programming (LNCS #14)*. Springer-Verlag, 1974, pp. 255-269.

**16**

Levine, J.R., Mason, T., and Brown, D. *Lex & Yacc*, second edition. O'Reilly & Associates, 1992.

**17**

McLean, P., and Horspool, R.N. A faster Earley parser. In *Proceedings of the International Conference on Compiler Construction (CC '96)*. Spring-Verlag, 1996, pp. 281-293.

**18**

Ousterhout, J.K. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

**19**

Tomita, M. *Efficient Parsing for Natural Languages*. Kluwer Academic, 1986.

**20**

van den Brand, M., Sellink, A., and Verhoef, C. Current parsing technologies in software renovation considered harmful. In *International Workshop on Program Comprehension*, 1998, pp. 108-117.

**21**

Weingarten, F.W. *Translation of Computer Languages*. Holden-Day, 1973.

---

# Biography

**John Aycock** (**aycock@csc.uvic.ca**) is a Ph.D. student at the Department of Computer Science, University of Victoria. He has been a Yacc user for slightly over ten years. His research interests include compilers and programming languages, and his thesis topic is, not surprisingly, general parsing algorithms.