Possible course and project coverage

I’ve found that if one tries to cover all the parsing methods in sequence, then the projects tend to fall behind because the necessary lectures haven’t been given. My solution is to alternate between the two kinds of lectures. I’ve tried to develop examples that serve as glue. One such example is the grammar that structures left and right values. While this is a good introduction to type checking, the grammar also illustrates the limitations of SLR parsing.

I begin with a few warm up assignments

1. The Chinese menu problem (10 days).
2. A finite-state machine problem, such as a reserved keyword or table generator (12 days).
3. Prefix expression evaluation, by recursive descent and a simple \texttt{YACC} grammar (2 weeks).

And then start the sequence of assignments that leads to a finished compiler.

1. Symbol tables, starting with the \texttt{C} grammar (2 weeks).
2. Abstract syntax trees (10 days).
3. Semantic analysis: left and right values, type checking (10 days).
4. Preliminary code generation: simple expressions (10 days).
5. Final code generation (2–3 weeks).
Conclusions

- Much of the work in crafting a compiler has been automated, by parser generators, tokenizing tools, attribute grammars, and automatic code generators.
- Partly due to these tools, creating the runtime library now occupies a large portion of compiler construction time.
- To understand and better appreciate the automatic tools, I believe in giving students practice in creating some components “by hand”.
- It’s important to get all the way through code generation in a one-semester course.
- I prefer to leave program optimization for a second course.
- In a projects course of this nature, one is often loathe to assign (or grade) homework. I have found the following strategy works well: Give out a list of 6 problems that the students should be able to work. One week later, give one of the six problems as a quiz, where the problem is determined by the roll of a die.

---

a Thanks to Ken Goldman and Sally Goldman for this idea.
In summary

Sung to the tune:
“I am the very model
of a modern Major General”

We start with some descriptions of our languages fanatical
That specify the syntax and the attributes grammatical
Through Yacc and Lexx our BNF is processed quite dramatical
By front ends that we generate these parsers automatical.

They shift, reduce, and scrutinize our errors problematical
And sometimes honest programs get transformed into the radical
But what the heck we know our derivations are canonical
And you’ll admit our diagnostics are the most laconical.

And you’ll admit our diagnostics are the most laconical,
Yes you’ll admit our diagnostics are the most laconical,
Because we know our diagnostics are the most laconi conical.

Code motion, hoisting, commoning and all the transforms you’d expect
Your program’s faster even if the output isn’t quite correct.
But most of us believe our transformations are canonical
And you’ll admit our diagnostics are the most laconical.
Some textbooks

Aho, Sethi, Ullman: *Compilers: principles, techniques, and tools*, Addison-Wesley, 1988 (affectionately called “The Dragon Book”). A long-time favorite in classes and as a reference book. Good coverage of popular parsing methods, though Earley’s method is not presented. Good description of runtime storage organization. No real connections are given to projects, and no tools are provided (but references to extant tools are given). No real insight given on how to disambiguate a grammar.

Fischer and LeBlanc: *Crafting a Compiler with C*, Benjamin/Cummings, 1991. There are actually two versions of this book, one with C and one based on ADA. This is an excellent textbook (my favorite), with excellent coverage of parsing, semantic analysis, and code generation. The text meshes nicely with tools provided by the authors.


Waite and Goos: *Compiler Construction*, Springer-Verlag, 1984. A now-dated but good text, fairly broad and not overly deep in any one area.

My favorite in teaching has been Fischer and LeBlanc. That book is currently undergoing revision, which will merge the language-specific versions and include much new material, for example on program optimization. I am joining them as a coauthor in this revision.
Reference books


References


