Using Ada 95 in a Compiler Course

S. Tucker Taft
AverCom Corporation,
a Titan company
12 Oak Park Drive
Bedford, MA 01730
+1-781-266-5500

STT@AverCom.Net

1. ABSTRACT

In this extended abstract, we describe the use of Ada 95 in a Compiler Construction Course.

1.1 Keywords

Ada 95, compiler construction, course

2. INTRODUCTION

The Compiler Construction course is often one of the most challenging elements of a Computer Science curriculum. Expecting a student to build a complete, functioning compiler in a single semester is asking a lot, and is often more than can be accomplished. Given the positive experience associated with using Ada for other challenging computer science courses [refs TBD], it seemed interesting to try to develop a Compiler Construction course based on Ada. The first task is to actually build a compiler in Ada, and then choose how to break it apart into pieces that either the student needs to build or the professor will provide in advance.

This spring, when beginning to teach my third Compiler course in the past few years, I decided to try to "follow along" with the assignments I gave to the students, who were generally building their compilers in C, C++, Java, or ML, but build my compiler in Ada. It has certainly kept me busy, but the compiler is now nearing completion, and it has been an interesting experience. This paper will explore the process of building a compiler in Ada 95 oriented toward teaching Compiler Construction, and provide some specifics on which parts might make sense for students to write, and which parts the professor might provide. It is my intent to make this compiler freely available for others interested in the teaching using Ada 95.

The recently published series of three books [ref] by Andrew W. Appel have provided the structure for the compiler courses I have taught. The books are essentially identical to one another, except that the programming examples are either in C, ML, or Java. The titles are "Modern Compiler Implementation in {C, Java, ML}." Appel provides some pieces in these three languages for students to use. However, I found various aspects of Appel's compiler design less than satisfactory, and in particular, he makes little or no use of object orientation, even though Java is one of the languages supported. Clearly when building a compiler in Ada 95, it would be a shame not to take advantage of the tagged type capabilities.

Compilers are loaded with places where Ada's type extension is a perfect fit to the data representation problem. There are various kinds of tokens in the lexical analyzer (aka "lexer"), there are various kinds of nodes in the abstract syntax tree (AST), there are various kinds of constructs in the intermediate representation (IR), and various kinds of instructions in the generated machine code. Each of these correspond to a place where a hierarchy of tagged types is the natural choice for implementation. But making this most basic choice is just the beginning of a myriad of decisions that need to be made in structuring a compiler that should be robust and reasonably efficient yet still easy to understand and extend.

One set of critical decisions is determining where the language and target dependencies should appear. Some compilers are designed to maximize the amount of code that is independent of the language being compiled, and even more so independent of the target instruction set architecture. The ultimate "dream" is the compiler-compiler, where the compiler structure and algorithms are independent of the language and target, and a "simple" parameterization process is all that is required to produce a compiler for any given language and target. Alas this has largely remained just a "dream," despite many efforts in the past. As usual, the "devil is in the details," and either efficiency or completeness have typically suffered to such an extent that most compilers are still being built largely by hand. Probably the only truly successful compiler building tool produced over the past 25 years is YACC and its derivatives. Even lexer generators are often bypassed in favor of a hand-written lexer, given the surprising amount of CPU time devoted to lexical analysis in a typical compiler, and parser generators like YACC sometimes fall

SIGAda 2001 09/01 Bloomington, MN, USA

by the wayside when good syntactic error recovery is desired.

One very common approach is to split a compiler into two "ends," a front end which is language-specific, but largely target independent, and a back end which is target-specific, but largely language independent. The two ends communicate through an "intermediate representation" (IR) which, ideally, is both language and target independent. Sometimes a middle "end" is thrown in, either to translate from a high-level IR to a lower level IR, or to perform IR-to-IR optimizations. This is basically the approach that Appel follows, the one that I have encouraged for my students, and the one I have taken for my "own" Ada compiler project.

In the front end is the lexical analyzer (lexer), the syntactic analyzer (parser), static semantic analyzer (semantics), and the IR generator (dynamic semantics insertion). An optional flow analyzer "end" transforms the IR back to itself, hopefully smaller and/or more time efficient (while still producing the right answer!). In the back end is the instruction selection phase, the register allocator, optionally some amount of peep-hole or instruction-scheduling optimization, and the final assembly-code generator.

In the "old" days, often more than half of a compiler construction course was spent on lexing and parsing, but these days, there is much more time spent on semantics and back end phases. This happens to correspond better to where time is spent on modern "industrial-strength" compilers. In addition, the availability of tools like "lex" and "yacc" have made lexing and parsing more an exercise in running a couple of tools than in designing data structures and algorithms. On the other hand, one of the main challenges for students (or any compiler-writer) in a compiler course is coming up with the "right" data structures and abstractions that will allow them to create the various phases of the compiler and have them actually all work together as a productive whole. It is here that the instruction and pre-defined interfaces provided by the instructor come in most handy. The actual algorithms involved are relatively straightforward and interesting for students to write, but if the underlying abstractions are wrong, the compiler writing efforts can produce a chaotic morass of snarled code.

[The remainder of this paper will provide details on the various abstractions designed to support the compiler, essentially in the form of one or more Ada package specs, and how they can be used to help a student structure their compiler while still giving them plenty of opportunity for learning about compilers and software engineering. We will compare our approach with the approach used by various compiler textbooks, including Appel's, as well with various other textbooks that attempt to lead a student through a complex software development project without giving them the entire answer. We will attempt to indicate where the various features of Ada support the educational process, and provide natural places to separate the pre-written code provided by the instructor from the code to be written by the student.]