Making Compiler Design Relevant for Students who will (Most Likely) Never Design a Compiler*

Saumya Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85721
debray@cs.arizona.edu

Abstract

Compiler Design courses are a common component of most modern Computer Science undergraduate curricula. At the same time, however, compiler design has become a highly specialized topic, and it is not clear that a significant number of Computer Science students will find themselves designing compilers professionally. This paper argues that the principles, techniques, and tools discussed in compiler design courses are nevertheless applicable to a wide variety of situations that would generally not be considered to be compiler design. Generalizing the content of compiler design courses to emphasize this broad applicability can make them more relevant to students.

1 Introduction

Compiler design courses are a common component of Computer Science undergraduate curricula at most universities. Students typically study a variety of topics about compiler design theory, such as LR(1) parsing or attribute grammars, and implement a compiler for some (smallish) subset of a language such as C or Java. It seems unlikely, however, that typical computer science students will apply, in their day-to-day professional lives, the arcana of LR(1) parse table construction or graph-coloring-based register allocators. The vast majority of these students are unlikely to ever design a compiler, in the conventional sense of that term: i.e., something that generates machine code from a high-level program. My impression is that students are aware of this, consider compiler design to be less "relevant" to their technical education than, say, courses on operating systems or networking, and thereby put correspondingly less time and effort into studying compiler design.

*This work was supported in part by the National Science Foundation under grants EIA-0080123 and CCR-0113633.

It turns out, however, that many of the techniques and algorithms used by compilers are actually much more broadly applicable than just for translating high-level programming languages to asembly or machine code. Emphasizing this aspect of compiler design—and illustrating it with a wide variety of examples during the course—can bring home to students that the material taught in a compiler design course in fact has a great deal of relevance to a variety of computational problems well outside what one typically thinks of as compilation problems. The idea is to consider compilers as just one instance of translators, broadly, from (almost) any arbitrary source language to (almost) any arbitrary target language, rather than in the more narrowly defined traditional view of compilers where the input is a program in a high-level computer programming language and the output is low-level assembly or machine code.

There are many examples of such translators—discussed later in this paper—that fall outside the traditional model of compilers; a lot of them don't involve programming languages at all. In each of these cases, however, the translation process has roughly the same structure: an input string is decomposed into tokens; the token sequence is grouped into "phrases" whose structure is specified by (something akin to) a context-free grammar; and these phrases are finally mapped to the output sequence in a manner determined by their structure and the context in which they occur. Many of the issues that arise, including the ways in which the input can be organized into tokens and phrases and the ways in which such phrases can be represented and manipulated, are very similar across all of these examples. Focusing on these commonalities makes it possible to present many traditional compiler techniques, e.g., buffer management for lexical analysis, parsing techniques for context-free languages, and attribute evaluation and propagation in parse trees, in a much more general setting that emphasizes their relevance to a significantly wider range of applications. It also shows how compiler development tools such as *lex* and *yacc* can be applied for many translation problems that students do not typically see as compilation problems.

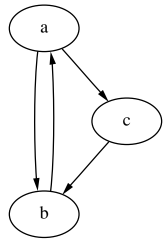
In addition to illustrating conceptual similarities between superficially very different translation problems, a discussion of the commonalities and differences between various such translation problems can help clarify the kinds of situations where one can reasonably expect (or not expect) to con-

```
graph          →  Type id '{' stmt_list '}'
Type           →  digraph | graph
stmt_list      →  stmt stmt_list | ε
stmt           →  node_stmt | edge_stmt
               |  subgraph | id = id
node_stmt      →  id opt_attribs
opt_attribs    →  '[' attrib_list ']' | ε
edge_stmt      →  edge_id edge_rhs_list opt_attribs
edge_id        →  id | subgraph
edge_rhs_list  →  edge_rhs edge_rhs_list | ε
edge_rhs       →  edge_op edge_id
   ...
```

(a)

```
digraph G {
   a -> b;
   b -> a;
   a -> c;
   c -> b;
}
```

(b)



(c)

Figure 1: (a) A (partial) grammar for graph specifications for *dot*; (b) a sample graph specification; (c) the graph drawing produced by *dot* for the specification in (b)

struct such translators. For example, we can identify specific technical reasons, e.g., metaphor and ambiguity, that make it impossible to construct an automatic translator that is able to translate arbitrary pieces of English text into, say, French; but this allows us to conclude that in specific situations where features such as ambiguity and metaphor can be eliminated, e.g., in technical manuals, it may be possible to construct automatic translators for natural languages using techniques derived from compiler design.

The remainder of this paper considers how different components of a traditional compiler can be generalized along these lines. We have used this approach in the undergraduate Compiler Design course at the University of Arizona for several years.

2 Some Example Translation Problems

The previous section suggested a generalized view of translators. In this section we discuss two specific examples of such systems in more detail in order to make the analogies to compilers nore explicit.

2.1 Dot: A Graph-Drawing Tool

Dot is a tool that reads in a textual specification for a (directed or undirected) graph and produces a drawing of that graph, e.g., in the form of a PostScript file [1]. The input to *dot* is a text string—it turns out that these strings can be described by a context-free grammar, i.e., form a context-free language—while the output is a string that is a pictorial representation of the graph. This is illustrated in Figure 1: Fig. 1(a) shows part of the grammar for *dot* inputs, Fig. 1(b) shows a sample input for *dot*, and Fig. 1(c) shows the drawing for the graph of Fig. 1(b) produced by *dot* (the actual output produced by *dot* is a PostScript file which, when viewed, yields the picture shown).

The actions carried out by *dot* when processing an input file such as that shown in Figure 1(b) are as follows:

1. Read in the graph specification using context-free parsing techniques.
2. Check for semantic consistency between components of the specification. For example, *edge_op* must be '`->`' for a graph of type `digraph` and '`--`' for one of type `graph`.
3. Construct an internal representation of the graph specified.
4. Determine the "output" representation of this graph, i.e., where different nodes and edges will be placed and how they will look.
5. Modify the output representation to improve its appearance, e.g., by reducing the number of unnecessary edge crossings (where one edge crosses over another) where possible.
6. Generate the final PostScript for the graph.

It is not difficult to see that there is a close correspondence between this sequence of actions and those of a compiler: step (1) above corresponds to the lexical analysis and parsing phase of the compiler; step (2) to type checking; step (3) to syntax tree construction; step (4) to code generation; step (5) to code optimization; and step (6) to code generation. A similar comment applies to other drawing tools such as *jgraph* [4] and *gnuplot*.

2.2 Translating LaTeX to HTML

LaTeX [2] is a typesetting system that is widely used for document formatting (at least in academia); HTML is a markup language used for specifying the appearance of web pages on the Internet. While superficially similar in that they both describe the appearance of documents, the two languages have very different syntax—fragments of context-free grammars for the respective syntaxes are shown in Figure 2—and are considerably different in their features and strengths.

Nevertheless, authors who prepare documents using LaTeX may then want to create web pages from them by translating them to HTML. To this end, several tools are available for translating Latex documents to HTML (e.g., see [3, 5]).

Such translators typically proceed as follows:

1. Read in the LaTeX document using context-free parsing techniques.

document → *hdr preamble body*
hdr → *docCls DocOpts* { *class* }
docCls → `\documentclass`
DocOpts → '`[`' *doc_opt_list* '`]`' | ε
preamble → ···
body → *begn_doc secn_list end_doc*
begn_doc → `\begin{document}`
end_doc → `\end{document}`
secn_list → *secn secn_list* | ε
secn → *secn_hdr para_list*
···

(a) LaTeX

document → `<html>` *head body* `</html>`
head → `<head>` *title* `</head>` | ε
title → `<title>` *word_list* `</title>`
body → `<body>` *body_opts* > *objlist* `</body>` | ε
body_opts → *body_opt_list* | ε
body_opt_list → *body_opt* '`,`' *body_opt_list* | *body_opt*
body_opt → `bgcolor` = *color* | ···
objlist → *obj objlist* | ε
obj → *para* | *table* | *list* | *image* | ···
···

(b) HTML

Figure 2: Context-free grammar fragments for LaTeX and HTML documents

2. Construct internal representations of portions of the document, as necessary.

3. Process the LaTeX constructs and output the corresponding HTML.

The sequence of steps here is somewhat different from that of a graph-drawing tool or a compiler, primarily because the source and target languages are semantically much closer in this case, simplifying the translation process considerably. Nevertheless there are a number of similarities, primarily in the initial lexical analysis and parsing phase (step (1) above) and the final HTML generation (step (3) above), which is carried out by what is in effect a recursive tree walk. However, the translation is not entirely trivial, since we have to deal with the problem of handling LaTeX features, such as mathematical symbols, pictures, etc., that are not supported by HTML. This is typically done by resorting to GIF or JPEG images of the corresponding constructs. This requires the construction of an appropriate internal representation for the LaTeX construct and then transforming this to an image (step (2) above); the corresponding compiler analog is that of code generation for language features—such as inheritance and virtual function calls in an object-oriented language—that are not directly supported by the target architecture.

In the undergraduate compiler design course at the University of Arizona, the 0^{th} programming assignment has the students use *lex* and *yacc* to implement, in roughly $1\frac{1}{2}$ weeks, a translator from (a subset of) LaTeX to (a subset of) HTML. At that point, most students know very little about LaTeX, many don't know a lot about HTML, and none of them know anything about *lex* and *yacc*. The goals of the assignment are twofold: first, to get the students acquainted with *lex* and *yacc*, in preparation for a more traditional project implementing a compiler for a subset of C; and second, to illustrate the applicability of these tools to other translation problems. We use discussion sessions and on-line tutorials to give them just enough acquaintance with LaTeX and HTML so that the students know what they are doing. We revisit the problem in classroom discussions at the end of the term, when they are much better versed with these tools (*lex* and *yacc*); students often seem quite surprised and pleased to realize that they are now equipped to implement a nontrivial and practically useful piece of software, for a significant fragment of LaTeX, reasonably quickly and without a great deal of effort.

3 Phases of a Compiler

The execution of a compiler conceptually consists of four phases: (*i*) lexical analysis and parsing; (*ii*) semantic analysis; (*iii*) code generation; and (*iv*) code optimization. This section discusses each such phase with regard to how its ideas, concepts, and techniques can be useful in translation problems outside the realm of traditional compilation.

3.1 Lexical Analysis and Parsing

Lexical analysis refers to the process of examining the input to be translated and dividing it into groups of adjacent characters, called "tokens," that form the units for the remainder of the translation process. Conceptually, this is analogous to examining a stream of English text such as this document, character by character, and grouping the characters into units such as words, numbers, and punctuation. This is typically done using regular expressions to specify the structure of tokens, and using the corresponding finite state machines to carry out pattern matching against the sequence of input characters being examined. Since this is the only phase of a compiler where the input is examined a character at a time, lexical analysis tends to be amongst the most expensive components of the compilation process; compilers employ sophisticated buffer management techniques to reduce the cost of lexical analysis as far as possible. Further, given the well-understood nature of regular expressions and finite automata, tools, such as *lex* and *flex*, have been developed that can automatically generate lexical analyzers given a set of regular expressions that specify the structure of the tokens to be recognized. These tools incorporate the buffer management techniques mentioned above, making the generation of lexical analyzers a relatively straightforward and painless process.

Parsing, or syntax analysis, is the process of imposing structure on the sequence of tokens obtained from lexical analysis. It is conceptually akin to taking a sequence of words and punctuation obtained from the tokenization of a document and constructing sentences from it, together with information about the structures of those sentences, e.g., the subject, object, modifiers, etc. The syntactic structure of programming languages is typically specified using context-free grammars, with the parsing process then being carried out using pushdown automata obtained from those grammars. The result of parsing is a representation of the syntactic structure of the input program, typically in the form of a structure called the parse tree. Again, the theory of context-free parsing is well understood, and tools, such as *yacc* and *bison*, can take (suitable) grammar specifications and generate parsers from them.

For many translation problems—particularly those where the input consists of ASCII text—the tokens can be specified as regular expressions. This makes it possible to directly apply tools and techniques developed for lexical analysis to handle tokenization for such problems. Similarly, the syntactic structure of such token sequences can very often be expressed in the form of context-free grammars, making it possible to use off-the-shelf parser generators such as *yacc* or *bison* to construct parsers for them.

A specific example of such a non-compiler problem that can be handled using lexical and syntax analysis techniques and tools borrowed from Compiler Design is that of database query translation, from a domain-specific natural-language-based query language convenient for humans to a language such as SQL supported by commercial database systems. My personal acquaintance with such a problem is in the context of a local company that makes software for hospitals and medical applications. I was told me of a product they were working on to allow doctors to quickly look up patient records, medication histories, etc., from a centralized database. Recognizing the unlikelihood of having doctors learn SQL, they designed a simple natural-language-like domain-specific query language for this application, struggling long and hard to build an *ad hoc* front end for this language, where a *lex*-and-*yacc* front-end would have been much quicker to build, and perhaps sturdier. (I wish I could say that I prevailed upon them to use The Right Tools for their project; unfortunately, the manager involved had neither the time nor the inclination to look into *lex* and *yacc*.)

The observation that the front-end issues for many translation problems closely resemble those of a compiler's is not particularly deep. The conclusion that follows, that techniques and tools developed for compiler front ends may be applicable to other translation problems as well, also does not come as a great surprise. However, students often seem to compartmentalize their knowledge, and thereby find it difficult to apply lessons from compiler design courses to other translation problems unless the underlying similarities between the problems are pointed out explicitly and repeatedly. Once they see the similarities, however, they find that using tools and techniques from compiler design can be very helpful. As an example, a few years ago, while teaching an undergrauate course on Formal Languages and Automata Theory, I asked one of my teaching assistants to write a software package to allow our students to specify various sorts of automata in the form of a text file and then simulate their behavior on input strings. Despite being a bright student and talented programmer, he struggled unsuccessfully with the construction of a front end for over a week, after which I suggested that he use *lex* and *yacc* to construct the front end. Once he was able to abstract away from specifications for automata and view this as just another translation problem, he was able to program up the front end in under a day.

3.2 Semantic Analysis

Semantic analysis refers to the computation and propagation of information that is not part of the context-free syntax of the language. In a compiler, this might refer to the type or scope of a variable. A common way of handling such information is using "attribute grammars," which associate properties ("attributes") with grammar symbols and specify rules, called semantic rules, for computing their values. These rules in effect specify the flow of information between different points in the parse tree for a program.

Not surprisingly, information has to be propagated along the parse tree for many other translation problems as well. An example that we discuss in class involves displaying HTML documents in a browser. The input in this case is an HTML document, with tags such as ··· and <i>···</i> that affect the way specific characters are displayed, as well as the amount of space taken by a group of characters (a boldface character is typically wider than one that is not). The output is the sequence of characters being displayed in the browser window. Among the problems to be addressed is the determination of when the line being displayed is "long enough," making it necessary to emit a line break character. This makes it necessary to figure out how to compute and propagate semantic information about the font in use at any particular point in the text as well as the line length in the display window up to that point.

While this problem is straightforward when restricted to simple text with a few different fonts, it becomes considerably more complex when other kinds of objects, e.g., images and tables, are allowed. Thinking about it in terms of attributes and semantic rules provides a systematic approach to addressing the problem, and makes it easier to figure out what information needs to be propagated, how, and between which points. While at first glance the problem seems very far removed from programming language compilation, ideas and techniques from compiler design carry over quite directly to produce a clean solution to a technically nontrivial problem.

3.3 Code Generation

Code generation in a compiler is the process of traversing the tree representation of a program to generate assembly or machine code for the target machine. More generally, however, we can think of this as an instance of the process of translating from a representation of a source language entity to that of a corresponding target language entity. This view accommodates many other translation problems, and allows us to think of them within a coherent framework.

Typically, code generation involves a post-order traversal of the tree representation of the input program. This means that the children of a node a in the tree—which represent the operands of the operation at node a—are processed first, i.e., have code generated to compute their values. After this, node a is processed to generate code for its operation; this can be code that uses the values computed by the child nodes to compute some other value (e.g., if a is an arithmetic operation), or it can be "glue" code that incorporates additional instructions to manage the correct control flow with the code for a's children (e.g., if a represents an *if-then-else* or a loop). The essential intuition here is that the node a specifies how (the values computed by the code generated for) its children are to be *used*.

This intuition can easily be transferred to other translation problems. When translating a technical manual from English to German, say, this involves traversing the tree representation of the original English sentences. The actions at a particular node of the tree, then, might involve determining the order in which the translated fragments from the child nodes are assembled, e.g., with verbs moved towards the end of the sentence. When translating a natural-language query from a user into an SQL query for a back-end database, this might involve mapping user-level constructs (e.g., "which account has the highest balance?") to the appropriate SQL constructs ("`select ...`"). The correspondence is not difficult to see once it is pointed out. However, by generalizing the actions of the back end of a compiler, from the narrow domain of emitting assembly code for a microprocessor to the broader domain of producing a target language entity, we can understand the essential similarities between the back-end actions for a variety of translation problems.

3.4 Optimization

Compiler courses traditionally treat optimization in terms of code transformations that make the program run faster. A more general view is that optimization aims to reduce the "cost" of the generated code for some cost measure of interest. Traditionally, the cost measure most often used has been execution time; however, even within mainstream compiler research, other measures of cost have recently been gaining credence: these include code size (for limited-memory processors, e.g., in embedded and mobile systems) and energy usage (e.g., for battery-operated portable computers).

When we generalize to other translation problems, it may still make sense to consider the "cost" of a representation. As an example, the graph drawing tool *dot* [1] takes a textual specification of a graph as input and produces a pictorial representation of the graph, e.g., as a JPEG or PostScript file, as output. Since a picture with many edges crossing one another is harder to understand than one with fewer edge crossings, *dot* tries to "optimize" the pictorial representation it produces by changing the layouts of vertices and edges so as to reduce the number of edge crossings. Conceptually, this is exactly analogous to the optimization phase of a compiler. Other such examples of "optimization" include eliminating double negatives, or transforming passive voice sentences to active voice in machine translation of natural languages.

If the only effect of drawing these parallels was to point out analogies between components of different translation problems, they would have limited utility. It turns out that we can use these analogies to illustrate deeper aspects of the translation process than is usually covered in a typical compiler course. For example, dataflow analyses are often discussed as a collection of algorithms—e.g., for liveness, or reaching definitions—without the observation that the *raison d'être* for these analyses is to infer invariants about the behavior of a program; such invariants can then be used to support optimizations or other transformations in a way that guarantees "semantic equivalence" between the original and transformed representations. In other words, these analyses arise out of questions of the form *"what properties have to hold such that we can carry out some specific sort of transformation that we believe may be profitable?"* Transferred to other translation problems, we can ask similar questions about invariants necessary to guarantee semantic equivalence—or, even more generally, preserve some property of interest—when carrying out "optimizations" such as restructuring a natural language sentence, or changing the layout of a graph.

4 Conclusions

Compiler design courses typically focus narrowly on the translation of high-level programming languages into low-level assembly or machine code. Given that the majority of computer science students are unlikely yo be involved in compiler design as a day-to-day professional activity, this limits the relevance of such courses to the students' eventual careers. However, it is possible to generalize the traditional view and consider the problem of translating from a source language to a target language, where both the source and target languages are defined broadly, e.g., need not even be programming languages. Such a generalized view includes many translation problems, e.g., document formatting or graph drawing, that are not traditionally viewed as "compiler problems." Viewing such translation problems in this way allows us to identify and understand essential underlying commonalities of the translation process.

This has several benefits, among them that the use of tools such as *lex* and *yacc* to generate the front end of a translator reduces development time, and that by relying on well-understood techniques and avoiding *ad hoc* approaches to the lexical analysis and parsing problems, reliability is enhanced. Overall, therefore, students benefit from having a deeper understanding of a variety of translation problems; being able to apply techniques and tools developed for compilers to other translation problems; and thereby being able to produce better code more quickly.

References

[1] E. Koutsofios and S. C. North, "Drawing graphs with *dot*", AT&T Bell Laboratories, Murray Hill, NJ, 1993.

[2] L. Lamport, *LaTeX: A Document Preparation System, User's guide and Reference Manual*. Addison-Wesley, 1994.

[3] L. Maranget, "HeVeA User Documentation version 1.06-7", INRIA, France, May 2001. `http://para.inria.fr/~maranget/hevea/doc/index.html`

[4] J. S. Plank, "Jgraph – A Filter for Plotting Graphs in PostScript", Conference Proceedings, Usenix Winter 1993 Technical Conference, January 1993, pp. 63–68.

[5] R. W. Quong, "Ltoh: a customizable LaTeX to HTML converter", April 2000. `http://www.best.com/~quong/ltoh`.