An Evaluation of an Automatically Generated Compiler

ANTHONY M. SLOANE
James Cook University

Compilers or language translators can be generated using a variety of formal specification techniques. Whether generation is worthwhile depends on the effort required to specify the translation task and the quality of the generated compiler. A systematic comparison was conducted between a hand-coded translator for the Icon programming language and one generated by the Eli compiler construction system. A direct comparison could be made since the generated program performs the same translation as the hand-coded program. The results of the comparison show that efficient compilers can be generated from specifications that are much smaller and more problem oriented than the equivalent source code. We also found that further work must be done to reduce the dynamic memory usage of the generated compilers.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Measurement techniques; D.2.m [**Software Engineering**]: Miscellaneous—*reusable software*; D.3.4 [**Programming Languages**]: Processors—*translator writing systems and compiler generators*

General Terms: Experimentation, Languages, Performance

Additional Key Words and Phrases: Compiler generation

1. INTRODUCTION

Considerable progress has been made toward the goal of automatically generating programming language compilers. Using specialized notations it is possible to specify compilation subtasks such as lexical analysis [Bumbulis and Cowan 1993; Gray 1988; Heering et al. 1992; Heuring 1986; Lesk 1975], parsing [Dencker et al. 1984; Gray 1987; Johnson 1975; Röhrich 1980], and semantic analysis [Deransart et al. 1988; Lee 1989; Schmidt 1986; Tofte 1990]. Other compiler tasks such as code generation for complex architectures have partially yielded to attack and are the subject of continuing research [Emmelmann 1989; Fraser et al. 1992]. Libraries of existing infrastructure can be used to combine solutions for subtasks into a complete compiler implementation [Gray et al. 1992].

Proponents of automatic compiler generation argue that it is a viable technique for producing compilers that would normally be coded by hand (e.g., see Lee [1989] and Gray et al. [1992]). This claim must be validated by comparing the generation of compilers with the production of compilers without the use of generators. Evaluation should be carried out often enough to enable cross-pollenation to occur and isolation to be avoided. The potential benefits are large because improvements resulting from evaluation will be available to all users of a generation system. In contrast, improved techniques for hand-coding compilers are often not widely pub-

An earlier version of this article appeared in the Proceedings of the 18th Australasian Computer Science Conference, Adelaide, Australia, February 1995.
Author's address: Department of Computer Science, James Cook University, Townsville, QLD, 4811, Australia; email: Anthony.Sloane@jcu.edu.au.

licised and may be "reinvented" by many developers.

Previous work evaluating the performance of generated compilers has largely concentrated on subtasks of the compilation problem. Lexical-analyzer generators have been compared with each other [Bumbulis and Cowan 1993; Gray 1988; Heuring 1986] and with the "minimal" lexical analyzer: a program that examines each input character once without any further processing [Gray 1989]. Similarly, the performance of parser generators has been examined (e.g., see Waite and Carter [1985] and Grosch [1990]). Tofte [1990] reports timing results from a comparison of two generated code generators for the same simple language.

Comparatively little work has evaluated complete generated compilers. One study compared different ways of specifying the same compiler from a notational point of view [Waite et al. 1989]. Based on the same input language, detailed experiments were performed to identify problems with generation tools [Gray 1989]. Some comparison of generated scanners with hand-coded scanners was performed but was not extended to other compilation phases. Lee [1989, p.47] summarizes results obtained measuring the performance of compilers generated from denotational specifications. The generated compilers measured were between 25% and 50% slower than hand-written ones. Also, the code produced by the generated compilers was in an intermediate form that imposed significant time penalties compared to compiled code. Lee also reports performance results for compilers generated by his MESS system. MESS-generated compilers generate good object code compared to compilers for related languages.

Two main deficiencies in previous work are addressed by this article: (1) a lack of evaluation of complete generated compilers and (2) limited comparisons of generated compilers with hand-written ones. While independent evaluation of generated compiler components serves a purpose for tool developers, users rarely use these components in isolation. Since complete compilers are being generated, complete compilers should be evaluated. Similarly, it is useful for tool developers to know how their tools shape up against other tools. The ultimate test of a compiler generation system, however, is against the most widely used compiler construction technique: hand coding.

This article reports the results of a comparison between a compiler generated by the Eli compiler construction system[1] [Gray et al. 1992] and an equivalent existing compiler written largely by hand. To enable direct comparison, the generated compiler was constructed to duplicate the functionality of the existing compiler. (Hence, the quality of the generated code is *not* part of the comparison.) We compare (1) the construction methods along the dimensions of the amount of information the compiler writer must provide and (2) how long it actually takes to construct the compiler given that information. We also compare the quality of the two compilers by measuring their performance compiling a large suite of programs. Both time and space consumption are considered.

Section 2 describes the functionality of the compilers and the test program suite used, while the following section summarizes our results. We found that compiler generation techniques offer notational advantages because specifications are shorter than the equivalent code and are more closely related to the problem domain.

[1]Eli distribution information can be obtained from `http://www.cs.colorado.edu/~eliuser`.

The generated compiler is usually faster than the hand-written compiler but is much larger and consumes a lot more dynamic memory. Section 4 contains a more-detailed analysis of some of the extreme measurement results. In particular, we consider the reasons why the generated compiler performs well and discuss areas for improvement. Finally, we analyze the reasons for the excessive memory consumption by the generated compiler.

2. TWO COMPILERS AND THEIR INPUT

Building a full-scale compiler for a general-purpose programming language is a significant effort. For practical reasons it was necessary to find a language that did not require an inordinate amount of work to compile. The translation task had to be complex enough to enable realistic evaluation, but not too complex so as to overstep the capabilities of current compiler generation tools. For example, a comparison with an optimizing compiler for a pipelined architecture would certainly be interesting. However, the state of the art in code generator generators is not at the stage where the backends of compilers of this kind can be fully (or even mostly) generated. Consequently, the "generated" compiler would necessarily include a significant amount of nongenerated code, limiting the utility of the comparison.

The Icon programming language [Griswold and Griswold 1983] was chosen because translation of Icon is of intermediate difficulty and because a well-documented implementation of the language is freely available [Griswold and Griswold 1986]. Icon is a general-purpose procedural language with particularly good support for string processing and high-level data structures such as tables and lists. More-unusual features included in Icon are goal-directed evaluation, generators, and co-expressions.

2.1 Icont

The baseline for the experiment was provided by the program *Icont*, part of the Icon implementation from the University of Arizona (version 8.10).[2] Icont translates Icon programs into a target form called Ucode [Griswold and Griswold 1986]. Ucode is a stack-based, postfix language designed explicitly for implementing Icon programs but similar in style to conventional assembly languages.

The distributed version of Icont was modified in the following ways:

(1) Invocation of the Icon preprocessor was disabled. Since we were not interested in measuring the implementation of the preprocessor its functionality was not duplicated.

(2) Icont as distributed contains code to link the Ucode for multiple input files and to invoke the generated program using a separate interpreter. This code was removed because it was unrelated to the translation task.

(3) Icont outputs a variety of messages indicating its progress during translation (mainly procedure names as they are processed). The code to generate these messages was removed to avoid biasing measurements with unnecessary output.

An informal inspection of the Icont source code indicates that it was produced by a programmer or programmers with extensive C programming experience.

[2] Icont is available via ftp from cs.arizona.edu as part of the Arizona Icon distribution.

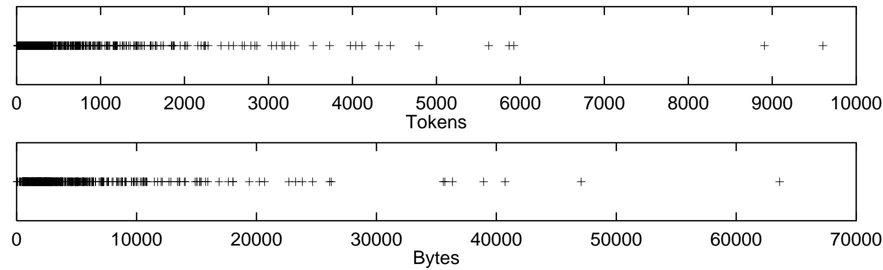

Fig. 1. Size distribution for the Icon program test suite. Each plus sign represents a single test program.

2.2 EliIcont

Icont was compared to an Icon translator (hereafter referred to as *EliIcont*) constructed using version 3.8.1 of the Eli compiler construction system [Gray et al. 1992].[3] EliIcont was written by the author, an experienced Eli user.

EliIcont was specifically designed to produce the same Ucode output as Icont, except for label numbering. To reproduce the label-numbering algorithm of Icont would have introduced code that would clearly not be written by anyone constructing the translator for any purpose other than comparison. The labels generated by EliIcont are placed in the same places as those generated by Icont, and there is a one-to-one mapping between them. In all other respects the Ucode generated by EliIcont is identical to that produced by Icont.

Other less-important aspects of Icont were duplicated in EliIcont as much as was practical. For example, relevant command-line options are supported, and equivalent user diagnostics are produced.

Once the functionality of Icont was duplicated by EliIcont and once the specifications were reasonably readable, the specifications were "frozen." No attempt was made to improve EliIcont's performance beyond that provided by the frozen version, since it would have been hard to know when to stop. In summary, EliIcont is representative of a class of specifications written in a "natural" style by experienced Eli users without any particular focus on performance. It is reasonable to assume that Icont has been optimized somewhat since it was first written, so the results presented in later sections are in some sense biased against EliIcont.

2.3 The Test Suite

The two programs were compared using a suite of 471 source files obtained from the University of Arizona Icon distribution. Every Icon program file from the distribution was included as distributed, except that those requiring preprocessing were preprocessed first. Their sizes in tokens and bytes are summarized in Figure 1.

The majority of the files are real applications or parts thereof, including graphics programs, text-processing applications, and various games. Thus the results reported later in this article apply to a wide variety of typical Icon programs. The suite also provides a particularly good basis for a comparison of translator function-

[3]The specifications for EliIcont can be obtained from the author.

Table I. Specification Sizes Measured in Bytes Allocated to Compiler Subtasks

	Total		Lexical Anal.		Parsing		Code Gen.		Misc.	
	Decl	Oper	Decl	Oper	Decl	Oper	Decl	Oper	Decl	Oper
Icont	124855		15873		27726		26174		55082	
	10786	114069	2900	12973	7455	20271	431	25743	0	55082
EliIcont	58410		7361		5236		45638		175	
	45091	13319	401	6960	5236	0	39279	6359	175	0

Comments were first removed, and each run of consecutive white space was compressed to a single space.

ality because it includes inputs designed to test the Arizona implementation fully. This fact gives us confidence that EliIcont does in fact duplicate the functionality of Icont.

3. RESULTS

This article concentrates on two attributes of compiler construction techniques which can be used for evaluation: effort and quality. This section summarizes measurements of these attributes of Icont and EliIcont. Section 4 analyzes the performance of the translators in more detail.

All measurements reported in this article were performed on a DEC 3000/500 Alpha AXP workstation running DEC OSF/1 V3.0. All files were stored on a local disk.

3.1 Specification Size

One way to assess the effort required to use a program construction method is to analyze the information that must be provided by the user. There are two categories of information that are used to construct our two compilers: operational specifications and declarative specifications. The former consists of code written in a general-purpose high-level language (in our case, C) that describes *how* to solve a problem. Declarative specifications are written in problem-specific notations and processed by tools to produce operational code. The identifying feature of a declarative specification is that it describes the problem itself rather than a particular solution. For example, a context-free grammar is a declarative specification of a parsing problem, whereas code implementing a recursive-descent parser is an operational specification.

Although it is not possible to compare the specifications for Icont and EliIcont directly, it is useful to get some idea of the relative effort needed to create them. Bearing in mind that the specifications reflect the different programming styles of their authors, Table I summarizes the specification sizes. (The figures for Icont are inflated slightly because they include some configuration and other code that are shared with other parts of the Arizona Icon distribution.) Since formatting styles vary, the figures are presented as byte counts after all comments were removed and white space compressed. These measurements are supplied only to give a feeling for the comparative sizes of the two compilers and are not generally applicable.

The two programs have the same basic structure. They both parse the input text and create an abstract tree representing the entire program. The abstract tree is then traversed to emit code. The "Parsing" column of Table I includes

tree construction. "Code Generation" includes semantic analysis because the two subtasks are intertwined in both compilers.

EliIcont is constructed mostly from declarative specifications with a small amount of supporting code. The declarative specifications are:

(1) A small set of regular expressions for lexical analysis.

(2) A concrete grammar for parsing.

(3) A five-line specification to control tree construction.

(4) Just over 1500 lines of attribute grammar code to perform semantic analysis and code generation.

(5) A short specification of output structure.

The supporting code is split about two-thirds for lexical analysis and one-third for output routines. The lexical-analysis code deals with complexities of Icon such as the insertion of semicolons at the ends of lines under various circumstances, handling multiline strings with character escapes, and processing `#line` directives (similar to those supported by C).[4]

Icont is built mostly from operational specifications. Declarative specifications are used for the parsing grammar and tables of operators and keywords. A hand-written scanner provides tokens for a YACC-generated parser. Parsing actions invoke tree construction routines, and a tree walker emits code.

Overall, EliIcont requires specifications less than half the size of those required by Icont. Code generation is more verbose in the attribute grammar formalism, but otherwise the phases of EliIcont require much smaller specifications. The major differences are:

—Icont's lexical-analysis phase is almost completely operational. The only declarative part is a table of operators. In contrast, the "easy" bits of EliIcont's lexical analyzer are specified with one line each. Only the complex parts (see above) require operational code.

—Icont uses YACC for parsing but requires extra code to construct the abstract tree. Almost all of the tree construction necessary in EliIcont is provided automatically by Eli based on an analysis of the parsing and abstract grammars.

—The Icont tree traversal for code generation is directly coded, so it is hard to determine what (if any) side-effects there may be if the tree structure is changed (Figure 2). Also, storage for auxiliary data must be explicitly managed (e.g., the mark count). Since EliIcont uses an attribute grammar formalism, tree traversals and attribute storage are computed automatically from dependences that reflect the structure of the output code (Figure 3).

The figures illustrate that the two code generation specifications are quite similar. The EliIcont approach is more verbose due to the extra structural elements that must be included (such as the production text and symbol names). However, the extra structure is likely worth it since the specification should be more maintainable since the traversals and storage are not explicit.

[4]This ratio of declarative-to-operational specifications for lexical analysis is not typical for Eli-generated compilers. It is usually the case that little or no operational specification is needed.

```
lab = alclab(2);
emitl("mark", lab);
loopsp->markcount++;
traverse(Tree0(t));
loopsp->markcount--;
emit("unmark");
traverse(Tree1(t));
emitl("goto", lab+1);
emitlab(lab);
traverse(Tree2(t));
emitlab(lab+1);
```

Fig. 2. Icont code generation for an *if-then-else* expression with three subexpressions. Code is emitted during an explicit traversal of the tree. Maintenance of auxiliary data such as mark counts must be coordinated with the traversal.

```
RULE: Ifexpr ::= 'if' Expr 'then' Expr 'else' Expr
COMPUTE
    Expr[1].marks = IncHeadintList (Ifexpr.marks);
    TRANSFER marks WITH Expr[2], Expr[3];
    .lab1 = GenLab ();
    .lab2 = GenLab ();
    Ifexpr.code = PTGseq8 (PTGinsnl ("mark", .lab1), Expr[1].code,
                           PTGIunmark (), Expr[2].code,
                           PTGinsnl ("goto", .lab2), PTGlab (.lab1),
                           Expr[3].code, PTGlab (.lab2));
END;
```

Fig. 3. EliIcont code generation for an *if-then-else* expression. PTG functions build output tree constructs which are emitted later. In contrast to the Icont version (Figure 2), no explicit tree traversal is needed, and auxiliary data are passed as attributes instead of needing to be coordinated with the traversal.

—Eli provides extensive infrastructure support, so almost no specification is needed other than for the translation tasks themselves.[5] The code for Icont contains a large amount of infrastructure support including memory allocation, string manipulation, command-line processing, and a driver to control the compilation phases.

In summary, the support that Eli provides enables compiler writers to concentrate more easily on the translation problem rather than peripheral details. Eli's philosophy [Gray et al. 1992] is to make the common things easy to do and to provide mechanisms by which more-difficult tasks can be accommodated. This approach particularly pays off for EliIcont for lexical analysis, tree construction, automatic generation of tree traversal algorithms, and compiler infrastructure.

Eli can be asked to provide complete source code so that a compiler implementation can be transported to sites where Eli is not present. The source code generated by Eli for EliIcont is much larger than the specifications provided by the user. For EliIcont, the total size is about 24,000 noncommented nonblank lines in 87 files. Of this, about 80% is generated code; 17% is from Eli libraries; and the rest is the operational portion of the specifications. The generated code is mostly lexical analysis (12%), parsing and tree construction (28%), and code generation (59%).

[5] The only miscellaneous specification is a three-line description of the compiler's command line.

3.2 Generation Time

Another measure of the utility of a compiler generation method is the amount of time it takes to process the specifications to produce the compiler. Dynamically linked, unoptimized executables for Icont and EliIcont can be built in 26 seconds and 218 seconds of real time, respectively.[6] The overhead of Eli is significant but not prohibitive. If the generated C source for EliIcont is extracted from Eli and compiled independently the build time reduces to 53 seconds.

3.3 Compiler Execution Time

To measure the relative speed of the two translators they were both instrumented using ATOM [?] which uses binary modification techniques to enable a large variety of performance analysis tools to be constructed [Eustace and Srivastava 1994]. ATOM was also used to collect the memory utilization measurements reported later in the article.

An ATOM-based reimplementation of the gprof tool [Graham et al. 1982] was used to collect call-graph-based execution profiles. The original gprof uses program counter sampling to estimate execution time. In contrast, the ATOM version uses the Alpha cycle counter [Sites 1992] to enable exact execution times to be collected. Also, gprof apportions execution time to the profile collection routine, so profiles are not exact representations of nonprofiled execution. The ATOM version does not have this drawback.

All execution times reported in this article are in terms of Alpha machine cycles. A benefit of the exact measurements obtainable with ATOM is that processing of small inputs can be used without the measurements being affected by measuring overhead or the granularity of the timing mechanism. Only user-level cycle counts were collected to remove perturbations due to factors outside the scope of this article (such as operating system overhead). Thus the figures reported are independent of system load.

Both Icont and EliIcont were compiled with the OSF C compiler using the highest level of optimization (-O3 option). Optimizations performed at this level include global register allocation and extensive procedure inlining. The executables were statically linked to avoid runtime overheads associated with dynamic linking and to enable them to be processed by ATOM.

Figure 4 summarizes the execution times of the two programs running on the complete test suite. Each vertical line represents a subset of the test programs. The percentage axis indicates the cycles taken by EliIcont to process those inputs relative to the cycles taken by Icont on the same inputs. EliIcont processes 406 of the 471 programs (86%) faster than Icont. Averaged over the whole test suite EliIcont is approximately 5.4% faster.

3.4 Memory Usage

The memory usage of a program can play a large role in determining its overall performance. Memory access patterns can produce subtle performance problems due to architectural features such as cache organization. Unfortunately, the performance characteristics of memory hierarchies vary considerably from machine to

[6] Measured by the Unix time(1) command on a lightly loaded system and averaged over ten runs.

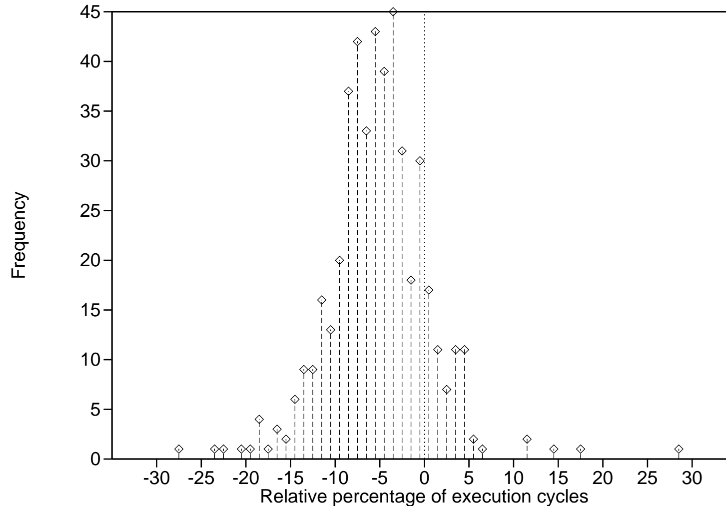

Fig. 4. The execution time of EliIcont relative to the execution time of Icont. Zero on the x-axis represents the runtime of Icont. Spikes to the left of zero indicate programs for which EliIcont was faster than Icont.

Table II. Sizes of Memory Used by Icont and EliIcont (in Kilobytes)

	Icont	EliIcont		Icont	EliIcont
Text	123	270	Initialized Data	41	57
Heap	222	1351	Uninitialized Data	25	19

Heap consumption is the maximum amount allocated at any one time during translation of the largest available application program.

machine, making general conclusions hard to reach. For this reason, this section only considers first order effects by looking at the overall amount of memory used.

Table II summarizes the memory consumption of the two compilers. Heap consumption was measured while translating the largest application program (an X11 text widget). Two major points of difference can be seen from Table II. First, as noted in Section 3.1, the Eli-generated compiler contains a lot more code. Second, EliIcont uses a lot more dynamic memory. The reasons for this difference will be discussed in Section 4.3.

4. PERFORMANCE ANALYSIS

The measurements reported in Section 3 show that there is an overall performance advantage in generating a compiler for translating Icon compared to the available Icon translator. However, there are some programs for which the generated compiler comes off second best. Moreover, the memory consumption of the generated compiler is higher than it needs to be.

This section considers the extreme points of Figure 4 in an attempt to understand the reasons for the performance differences. A program at the left end of the figure represents a significant advantage for the generated translator. These

programs provide the best context in which we can determine the reasons why compiler generation is better. Conversely, the programs at the right end of the figure represent programs for which the generated compiler performs badly. Study of these programs may suggest refinements of compiler generators that will further improve the performance of generated compilers. We also analyze the reasons for the high memory consumption of the generated compiler.

4.1 Influence On Current Practice

One of the benefits of compiler generation systems is that they allow users to take advantage of good compiler construction practices without needing to be experts. A good case in point is provided by the program for which EliIcont performs the best relative to Icont. When processing this program, Icont's lexical analyzer consumes about 46% (293 kilocycles) of the execution time, compared with about 15% (71 kilocycles) for EliIcont's lexical analyzer on the same program. This difference is more than the overall difference in execution times for the two compilers on this input.

Closer analysis of Icont reveals that most of the lexical-analysis time (33% of the total execution time) is spent in a routine to get the next character from the input. Previous research has shown that this is an area which deserves to be handled carefully [Waite 1986]. As a result, Eli includes a support module that provides efficient access to input characters without any special effort on the part of the compiler writer.

4.2 Efficiency Lessons

On the other side of the coin are the 65 programs for which EliIcont performs poorly compared to Icont. EliIcont suffers from an initialization inefficiency due to an Eli library module that maintains scoping information in an environment structure. Currently the whole of this structure is initialized with room for 8192 identifiers regardless of how many are actually needed. Given that the *largest* input has only 3790 identifiers much of the space and time spent during initialization is wasted. Reducing the initialization overhead by half would allow EliIcont to compile an additional 36 of the test inputs faster than Icont. Dynamic initialization may improve on this result, but the runtime penalties of such an approach need to be investigated.

Of the eight programs for which EliIcont is more than 5% slower than Icont, six are affected by environment initialization. For the other two programs (11% and 6.5% slower) it is hard to find an obvious reason for the difference. One possible factor is a difference between the abstract tree traversal overheads of the two compilers. As we will see in the next section, EliIcont allocates more abstract tree nodes than Icont, so it is more sensitive to extra fixed overhead in traversal.

4.3 Dynamic Memory Consumption

Section 3.4 drew attention to the large amount of dynamic memory used by EliIcont relative to Icont. While main memory is a relatively cheap resource compared to

CPU speed, a factor of six is a high price to pay for using a generated compiler.[7]

Closer examination of the memory dynamically allocated by EliIcont shows that the majority of it constitutes nodes for either the abstract tree or the output tree. The abstract tree is the basis for attribution (semantic analysis and code generation) and stores much of the data used during this process. EliIcont uses an Eli tool that allows output to be built in a piecemeal fashion according to an output grammar. Once complete, the output tree is traversed in a depth-first manner to actually produce the output.

Running on the text widget described in Section 3.4, EliIcont dynamically allocates about 1360 kilobytes of memory. Of this amount, about 522 kilobytes (38%) is used for 22,203 abstract tree nodes and just over 553 kilobytes (41%) for 25,486 output tree nodes. Abstract tree nodes consume 24 bytes on average and output tree nodes 22 bytes. For the same program, Icont allocates about 479 kilobytes for 13,770 abstract tree nodes with an average node size of 35 bytes.

Much previous research has been devoted to space-saving optimizations for attribute grammar systems (e.g., Deransart et al. [1988], Hall [1987], and Kastens [1987; 1989]). Work is currently underway to implement new optimizations in Eli and measure their effectiveness. Initial results indicate that significant space savings can be achieved.

The approach of building a complete output tree is questionable for this application since Icont demonstrates that output during abstract tree traversal is possible. For more-complex output languages than Ucode the memory overhead of constructing an output tree may be warranted if specifications are simplified as a result. Preliminary work on optimizing output tree space shows that up to 10% can easily be saved if identical leaf nodes are shared. More than 50% of the remaining space is currently devoted to sequencing information, so current research is aimed at reducing this overhead.

5. CONCLUSION

Comparisons of automatically generated compilers with equivalent hand-written compilers are important because they are the best to enable the utility of generation systems to be ascertained. The results from these kinds of comparisons are useful for two reasons. First, compiler writers who do not use generators may be convinced of the utility of their use. This article shows that a generated compiler can be faster than a hand-written one and can be produced from smaller, more problem-oriented specifications. Eli embodies detailed domain-specific knowledge that allows the compiler writer to concentrate on aspects specific to the translation, while at the same time producing a competitive compiler.

Second, developers of generation systems can use evidence from detailed comparisons to determine areas in which generated compilers lag behind hand-written compilers. For example, it is clear that the dynamic memory consumption of Eli-generated compilers should be addressed. Because any forthcoming improvements will be embodied in Eli, they will be available to all Eli users. This contrasts with the situation for hand-written compilers where improvements to one compiler are

[7]It should be noted that the dynamic memory consumption reported for both programs is larger than it would be on many machines due to the use of 64-bit pointers on the DEC Alpha.

usually not easily incorporated into other compilers.

Future work should include more studies of the type described in this article. Similar studies should be conducted for other compiler generation systems to determine whether the results presented here are Eli specific or are characteristic of a variety of generation approaches. The scope of evaluation should also be increased. While the Icon translator served as a good starting point for comparisons, it is necessary to gauge the effectiveness of generated compilers over a range of languages and in competition with a variety of hand-written compilers. In particular, it would be useful to compare a generated compiler with a hand-written compiler that did not explicitly build a complete abstract tree. Further work should also consider issues arising in generating compilers that perform code generation for real machines including optimization for modern architectures.

ACKNOWLEDGMENTS

Many thanks to the Icon team at the University of Arizona for making their Icon implementation freely available. Thanks to Alan Eustace and Amitabh Srivastava of DEC Western Research Laboratory for ATOM. William Waite, Karl Prott, Sam Kamin, and the anonymous reviewers made suggestions that improved the article.

REFERENCES

Bumbulis, P. and Cowan, D. D. 1993. RE2C: A more versatile scanner generator. *ACM Lett. Program. Lang. Syst. 2,* 1–4 (Mar.–Dec.), 70–84.

Dencker, P., Dürre, K., and Heuft, J. 1984. Optimization of parser tables for portable compilers. *ACM Trans. Program. Lang. Syst. 6,* 4 (Oct.), 546–572.

Deransart, P., Jourdan, M., and Lorho, B. 1988. *Attribute Grammars: Definitions, Systems and Bibliography.* Lecture Notes in Computer Science, vol. 323. Springer-Verlag, Berlin, Germany.

Emmelmann, H. 1989. *BEG—A Back End Generator.* GMD Forschungsstelle an der Universitaet Karlsruhe, Karlsruhe, Germany.

Eustace, A. and Srivastava, A. 1994. ATOM: A flexible interface for building high performance program analysis tools. Tech. Rep. TN-44, Digital Western Research Laboratory, Palo Alto, Calif.

Fraser, C. W., Hanson, D. R., and Proebsting, T. A. 1992. Engineering a simple, efficient code generator generator. *ACM Lett. Program. Lang. Syst. 1,* 3 (Sept.), 213–226.

Graham, S. L., Kessler, P. B., and McKusick, M. K. 1982. `gprof`: A call graph execution profiler. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction. SIGPLAN Not. 17,* 6 (June), 120–126.

Gray, R. W. 1987. Generating fast, error recovering parsers. M.S. thesis, Dept. of Computer Science, Univ. of Colorado, Boulder, Colo.

Gray, R. W. 1988. A generator for lexical analysis that programmers can use. In *Proceedings of the Summer USENIX Conference.* USENIX Assoc., Berkeley, Calif., 147–160.

Gray, R. W. 1989. Declarative specifications for automatically constructed compilers. Ph.D. thesis, Dept. of Computer Science, Univ. of Colorado, Boulder, Colo.

Gray, R. W., Heuring, V. P., Levi, S. P., Sloane, A. M., and Waite, W. M. 1992. Eli: A complete, flexible compiler construction system. *Commun. ACM 35,* 2 (Feb.), 121–131.

Griswold, R. E. and Griswold, M. T. 1983. *The Icon Programming Language.* Prentice-Hall, Englewood Cliffs, N.J.

Griswold, R. E. and Griswold, M. T. 1986. *The Implementation of the Icon Programming Language.* Princeton University Press, Princeton, N.J.

Grosch, J. 1990. Lalr—A generator for efficient parsers. *Softw. Pract. Exper. 20,* 11 (Nov.), 1115–1135.

Hall, M. L. 1987. The optimization of automatically generated compilers. Ph.D. thesis, Dept. of Computer Science, Univ. of Colorado, Boulder, Colo.

Heering, J., Klint, P., and Rekers, J. 1992. Incremental generation of lexical scanners. *ACM Trans. Program. Lang. Syst. 14,* 4 (Oct.), 490–520.

Heuring, V. P. 1986. The automatic generation of fast lexical analysers. *Softw. Pract. Exper. 16,* 9 (Sept.), 801–808.

Johnson, S. C. 1975. YACC – Yet another compiler-compiler. Computer Science Tech. Rep. 32, Bell Telephone Laboratories, Murray Hill, N.J.

Kastens, U. 1987. Lifetime analysis for attributes. *Acta Informatica 24*, 633–651.

Kastens, U. 1989. LIGA: A language independent generator for attribute evaluators. Bericht der Reihe Informatik Nr. 63, Universität-GH Paderborn, Paderborn, Germany.

Lee, P. 1989. *Realistic Compiler Generation.* The MIT Press, Boston, Mass.

Lesk, M. E. 1975. LEX – A lexical analyzer generator. Computing Science Tech. Rep. 39, Bell Telephone Laboratories, Murray Hill, N.J.

Röhrich, J. 1980. Methods for the automatic construction of error correcting parsers. *Acta Informatica 13,* 2 (Feb.), 115–139.

Schmidt, D. A. 1986. *Denotational Semantics.* Allyn and Bacon, Newton, Mass.

Sites, R. L. 1992. *Alpha Architecture Reference Manual.* Digital Press, Burlington, Mass.

Tofte, M. 1990. *Compiler Generators.* EATCS Monographs on Theoretical Computer Science, vol. 19. Springer-Verlag, Berlin, Germany.

Waite, W. M. 1986. The cost of lexical analysis. *Softw. Pract. Exper. 16,* 5 (May), 473–488.

Waite, W. M. and Carter, L. R. 1985. The cost of a generated parser. *Softw. Pract. Exper. 15,* 3 (Mar.), 221–239.

Waite, W. M., Grosch, J., and Schröer, F. W. 1989. Three compiler specifications. GMD-Studien Nr. 166, Gesellschaft für Mathematik und Datenverarbeitung, Karlsruhe, Germany.

Received March 1995; revised June 1995; accepted July 1995