Eli: A Complete, Flexible Compiler Construction System

Robert W. Gray
Vincent P. Heuring
Steven P. Levi
Anthony M. Sloane
William M. Waite

Over the past five years, our group has developed the Eli[1] system to reduce the cost of producing compilers. Eli has been used to construct complete compilers for standard programming languages extensions to standard programming languages, and special-purpose languages. For the remainder of this article, we will use the term compiler when referring to language processors.

One of the most important ways to enhance productivity in software engineering is to provide more appropriate descriptions of problems and their solutions. The Eli system reduces the cost of producing a compiler in exactly this manner. Compiler construction is one of the most widely studied applications in computer science. As such, many common subproblems of this application have been identified and standard solutions devised. With the decomposition of the compilation problem into a number of smaller subproblems, it becomes possible to solve each of the small problems by using declarative specifications instead of writing algorithmic code. The use of specifications permits the description of the nature of the problem rather than the nature of the solution; the nature of the solution is embodied in a tool that translates the specifications into code.

Unfortunately, very few compiler construction tools are in common use. Two of the most popular are LEX [11], a scanner generator that accepts regular expressions and produces a table-driven recognizer, and Yacc [8], an LALR(1) parser generator. Both of these tools are used principally for phototyping and for the generation of special-purpose processors, but are viewed by many as being too slow [11] or not providing adequate error recovery [8] to be used in a production environment. There are a number of additionl barriers to increased tool use:

• A steep learning curve is associated with the large number of different

[1]Named after Eli Whitney, who was the first U.S. manufacturer to make extensive use of interchangeable parts.

```
xSource:                  xBlock.
xBlock:                   'declare' xDecl (';' xDecl)* 'begin' xStatement_list 'end'.
xDecl:                    xVariable_declaration / xIdentity_declaration.
xVariable_declaration:    xIdDef ':' xType.
xIndentity_declaration:   xIdDef 'is' xExpression ':' xType.
xType:                    'integer' / 'real'.
xStatements:              xStatement / xStatements ';' xStatement.
xStatement:               xIteration / xExpression.
xIteration:               'which' xCondition 'do' xStatements 'end'.
xCondition:               xExpression.
xExpression:              xAssignment / xDisjunction.
xAssignment:              xLHS xBecomes xExpression.
xBecomes:                 ':='.
xLHS:                     xIdUse.
xDisjunction:             xConjunction / xDisjunction xOr xConjunction.
xOr:                      'or'.
xConjunction:             xComparison / xConjunction xAnd xComparison.
xAnd:                     'and'.
xComparison:              xRelation / xRelation xEq xRelation.
xEq:                      '='.
xRelation:                xSum / xSum xRelop xSum.
xRelop:                   '<' / '>'.
xSum:                     xTerm / xSum xAddop xTerm.
xAddop:                   '+' / '-'.
xTerm:                    xFactor / xTerm xMulop xFactor.
xMulop:                   '*' / 'div' / 'mod'.
xFactor:                  xPrimary / xUnop xFactor.
xUnop:                    'not'/ '+'/ '-'.
xPrimary:                 xDenotation / xIdUse/ '('xExpression')' / xBlock / xClause.
xDenotation:              xInteger / xReal.
xIdDef:                   xIdentifier.
xIdUse:                   xIdentifier.
xClause:                  'if' xExpression 'then' xStatements 'end'/
                          'if' xExpression 'then' xStatements 'else' xStatements 'end'.
```

Figure 1. Concrete syntax for Minilax

conventions and options needed to control each tool.
• Most tools were developed independently of one another, and do not work smoothly together.
• Many tools produce code whose performance is far below that of a hand-coded solution to the same problem.

The Eli system was created to overcome these barriers. A simple compiler model provides a specific set of subproblems to be solved by the designer. Solutions are described in the form of specifications that may be created specially, or extracted from a library, and the actual compiler construction is managed by an expert system that hides the details of tool use. Eli smooths the interaction between independently developed tools in three ways: it removes redundancy from the specifications, it hides all of the processing that might be needed to prepare the output of one tool for input to another, and it greatly simplifies user requests for products. Finally, Eli is an *open sys-*

Table 1.
Compilation Subproblems

Structuring	Lexical analysis	Scanning Conversion
	Syntactic analysis	Parsing Tree construction
Translation	Semantic analysis	Name analysis Type analysis
	Transformation	Data mapping Action mapping
Encoding	Code generation	Execution-order determination Register allocation Instruction selection
	Assembly	Instruction encoding Internal address resolution External address resolution

tem as it is easy to add tools or to replace existing tools with new versions that create higher-performance compilers.

The Eli System

Eli is a collection of off-the-shelf tools controlled by an expert system whose problem domain is the management of complex user requests [20]. This structure allows us to attack all three of the barriers to tool use we have mentioned.

Since the expert system manages the construction process, we can interpose arbitrarily complex processing to make simple user input acceptable to off-the-shelf tools. The number of specifications can be reduced, and special-purpose languages can be used to simplify those specifications. Values for many of the tool parameters can be deduced from the specifications themselves.

The Eli system carries out arbitrary processing to match the output of one tool to the input of another, to prepare consistent input for several tools from a single specification, and to combine outputs from several tools. Thus, tools developed by different people with different conventions can be combined into an integrated system, even when only executable versions of those processors are available.

To add a new tool to Eli, or to replace an existing tool with a better one, only the expert system's knowledge base must be changed. Eli users are not concerned with the knowledge base; they are only interested in the products and parameters that Eli provides. Knowledge base changes may add new products and parameters or make existing ones disappear, but most users can continue business as usual.

Interacting with Eli

The steep learning curve associated with compiler construction tools is the most significant barrier to their widespread use. To overcome this barrier Eli embodies a decomposition of the total problem into a set of well-defined subproblems, provides libraries and special-purpose languages for solving those subproblems, and automatically combines the solutions into a working compiler.

A rather stable overall design for a compiler has evolved from experience over the last 20 years. As summarized in Table 1, this design decomposes the problem of compiling a program into subproblems. The structuring task analyzes the phrase structure of the source program, determining the semantically significant components. Conceptually, the structuring task builds a tree according to the abstract syntax of the source language. The translation task uses contextual information to verify that the source program is consistent and derives an equivalent tree, whose structure is determined by the abstract syntax of the target language. Finally, the encoding task allocates target machine resources to the target program and maps the target tree into the appropriate machine instruction sequence.

Eli generates a compiler from specifications of the structures of the four objects postulated by this model (source program text, source program tree, target program tree and machine instruction set) and the relationships between them. Effectively, the designer defines a particular instance of the general compilation problem by providing these specifications. Some of the tools check the specifications for consistency, some extract information relevant to the specific subproblems listed in the third column of Table 1, and others generate code to solve those subproblems. Finally, the generated modules are combined with standard modules from Eli's library to obtain a complete compiler that solves the problem specified by the designer.

To make the discussion concrete, this article bases its examples on a simple language called Minilax [19]. Minilax is a conventional block-structured, expression-oriented language with variables of type *integer* and *real*. Assignment is an operator, and there are Boolean expressions (but no Boolean variables), if-clauses and while-loops. The context-free grammar for Minilax is shown in Figure 1.

Controlling the Tools. A complete description of a Minilax compiler consists of several files, each containing a specification or part of a specification. Some of these files may be taken from a library, others may be shared among several different projects. A user must be able to submit this collection of files to Eli, and to select properties of the compiler being described. It may only be necessary to test some aspect of the compiler. On the other hand, it might be necessary to obtain an executable version of the compiler, or a directory of source files from which an executable version could be built without the use of Eli.

Most requests will involve a bewildering array of tools and intermediate products. Some of these products may already have been constructed to satisfy previous requests. Eli removes the burden of managing this complexity from the user by placing it on the shoulders of an expert system called Odin [2] whose area of expertise is the management of complex user requests [5]. Eli's component tools and their relationships are described by a derivation graph that resides in Odin's knowledge base. Odin also manages a cache of derived objects. When a user makes a request of Eli, Odin's inference engine determines the sequence of operations needed to satisfy that request, reusing cached objects in the derivation wherever possible.

Eli's primary input is a text file whose name is the name of the compiler being generated, followed by the extension '.specs'. (Each file name has an extension that names the *type* of that file. File types are used by the expert system to determine how the file should be processed. Thus the extension '.oil' indicates that the compiler for the

```
#include "structure.specs"   Specifications for the structuring task
#include "translate.specs"   Attribute grammar for the translation task
environment.lib              Use the standard name analysis module
Opident.oil                  Specification for the type analysis module
Properties.ddl               Specification for the definition table module
#include "vax.specs"         Specification for the encoding task
```

(a) The content of file Minilax.specs, a specification file

Check whether the concrete syntax satisfies the parser generator constraints:

```
    Minilax.specs :parsable
```

Apply the compiler being developed to program `test.mla` and display its output:

```
    Minilax.specs +arg=(test.mla) :stdout
```

As above, but only display error reports:

```
    Minilax.specs +arg=(test.mla) :stdout :err
```

Put an executable version of the Minilax compiler into file `Minilax.exe`:

```
    Minilax.specs :exe > Minilax.exe
```

Put complete source text for the Minilax complier into directory `src`:

```
    Minilax.specs :source > src
```

Obtain a version of the Minlax compiler with embedded profiling code:

```
    Minilax.specs +prof :exe > Minilax_prof.exe
```

(b) Some typical Eli requests involving the specification file of (a)

Figure 2. Using Eli

```
% Operations

OPER bEql, bOr, bAnd (boolean,boolean): boolean;
OPER bNot (boolean): boolean;
OPER iAsn (refinteger,integer): refinteger;
OPER rAsn (refreal,real): refreal;
OPER iLss, iEql, iGtr(integer,integer): boolean;
OPER rLss, rEql, rGtr(real,real): boolean;
OPER iAdd, iSub, iMpy, iDiv, iMod(integer,integer): integer;
OPER rAdd, rSub, rMpy, rDiv(real,real): real;
OPER iNeg, iNop(integer): integer;
OPER rNeg, rNop(real): real;

% Identification

INDICATION Plus:     iAdd, rAdd;
INDICATION UPlus:    iNop, rNop;
INDICATION Minus:    iSub, rSub;
INDICATION UMinus:   iNeg, rNeg;
INDICATION Star:     iMpy, rMpy;
INDICATION Div:      iDiv, rDiv;
INDICATION Mod:      iMod;
INDICATION Assign:   iAsn, rAsn;
INDICATION Or:       bOr;
INDICATION And:      bAnd;
INDICATION Equal:    bEql, iEql, rEql;
INDICATION Less:     iLss, rLss;
INDICATION Greater:  iGtr, rGtr;
INDICATION Not:      bNot;

% Coercions

COERCION float(integer): real;
COERCION ifetch(refinteger): integer;
COERCION rfetch(refreal): real;
CORECION idiscard(integer): void;
COERCION rdiscard(real): void;
COERCION bdiscard(boolean): void;
```

Figure 3. Specifying the type lattice and operator identification; Opident.Oil

Operator Identification Language should be applied to that file. Eli passes the specification file through the C preprocessor, and then interprets each line as the name of a specification file. The use of the C preprocessor allows the user to group specification file names logically, control the selection of certain specifications by directives, and include specifications from libraries.

Figure 2 shows the top-level specification file for the Minilax compiler, and some Eli requests that might be made during compiler development. The specification files **structure.specs**, **translate.specs** and **vax.specs** list the specifications for the three major compilation subtasks given in Table 1. We will consider **structure.specs** in the section on tool interaction, **translate.specs** will be given in Figure 4a, and **vax.specs** are extracted from Eli's library. A standard module for carrying out the name analysis task, shown in Table 1 is also available in Eli's library, and this module's interface is made available via **environment.lib**. This interface is not a set of specifications, and therefore its name does not have the extension **.specs**. **Opident.oil** and **Properties.ddl** are all specifications, and are discussed in the section on subproblem specifications.

Figure 2b shows examples of some user requests of the Eli system. Each request line in the figure is read from left to right. A colon (:) can be read as "derive to"; a plus sign (+) introduces a keyword parameter, which may or may not have an associated value. Notice that it is possible to derive an object from a derived object — ":err" is a general derivation that obtains error reports produced by some other derivation. A greater than symbol (>) is a redirection mechanism. It is used to place the object resulting from the derivation into a file or a directory.

Notice that individual tools are never invoked directly when using Eli, and their particular interfacing requirements are invisible. The user is concerned only with composing the appropriate request. Based on that request, and the state of the cache, Eli determines what needs to be done, that is, what tools to invoke, and what intermediate results to produce. In order to make requests of Eli, a person must learn only a few derivation and parameter names. By hiding all the conventions and most of the options needed to control each tool, Eli sharply reduces the number of items that must be learned.

The following will illustrate how subproblem solutions are specified to Eli and how the resulting modules are combined to produce a complete compiler.

Subproblem Specifications. One of the difficulties in learning to use conventional compiler construction tools is learning how to specify the solutions to the subproblems (as listed in Table 1) in the notation of those tools. Eli attacks this problem by providing specialized languages that are tailored to specific subproblems. If one understands the subproblem, the specification language is quite obvious.

One of these special-purpose languages, OIL, is used to solve the operator identification problem common to compiler construction. Our purpose in showing this example is to illustrate both how specialized notations match the designer's understanding of the subproblem, and how the designer can describe an instance of the problem rather than a solution method.

The operator identification problem is easily stated: most typed languages permit the *overloading* of operators, so that a given operator may have operands of several types. It is the responsibility of the compiler to detect valid operator/operand combinations and to generate the correct code to implement the operation. The compiler may also have to generate code to coerce operands of one type to another. We use the term *operator indication* to make explicit that the same operator may be used to indicate several kinds of operations. In Minilax the operator indication "+" may indicate integer addition or real addition. The language also permits operations such as,

IntegerValue + RealValue

for which the compiler needs to generate a coercion operation to coerce the integer value to a real value prior to performing real addition.

The first section of Figure 3 lists the operations that are available in Minilax, specifying the signature for each. Thus, Minilax provides the usual arithmetic operations on both integer and real values, and the compiler writer has chosen to name these operations *iAdd* (integer addition), *rAdd* (real addition), and so forth. The "+" is an overloaded operator that could mean either integer or real addition, as indicated in the section entitled "Identification." If an integer value is given in a context where a real value is required, then the compiler is allowed to use the *float* operation to convert that value from integer to real because an appropriate coercion has been specified, as shown in the section entitled "Coercions." Minilax is a typed language. Types in a language are distinct entities that are related to other types by a lattice [7]. Figure 3 describes this type lattice, and the operator identification based on it. (The relationship on which the lattice is based is the partial order *is coercible to.*)

Figure 3 is the input specification to OIL. OIL allows the language designer to specify the operator indications, operators, operator identifications, and coercions permitted by the language. OIL then generates functions that can be called within the attribute grammar specification to perform operator identification.

Subproblem Interaction. The particular module that solves the operator identification subproblem can be derived from Figure 3. This module makes operations available, but those operations must be invoked with arguments that are determined from the specific program being compiled. The order in which the operations are invoked is also dependent on the source program. In this section we show how Eli uses an attribute grammar [17] written in LIDO to knit together those operation invocations into a consistent, cohesive, solution to the translation subproblem of Table 1.

The file `translate.specs`, shown in Figure 4a is a list of the files constituting the attribute grammar that describes the information flow of the translation task. Each file contains a piece of the attribute grammar, defining the computations associated with a few related nonterminal symbols of the grammar. These computations are basically invocations of the procedures defined by modules generated from specifications such as those in Figure 3 or extracted from the Eli library. A complete description of LIDO, the language in which Figure 4b is written, is beyond the scope of this paper [10]. Briefly, a rule describes actions related to a single node of a tree. That node is the one described by the production lying between the keywords '***RULE***' and '***STATIC***'. The first rule in the figure, under the comment "% Minilax Operators" is a rule describing actions related to the binary plus operator.

Attribute computations associated with the given rule are contained between the keywords "***STATIC***" and "***END***". The first line under ***STATIC*** sets the indication attribute of the nonterminal symbol *xBinop* to *Plus*. As Figure 3 shows, this operator indication could be identified as either the integer addition operator *iAdd* or the real addition operator *rAdd*. The actual operator identified becomes the value of the *operator* attribute of the "*xBinop*" node. Assert invokes the

OIL library function OilIsValidOp to verify that the identified operator does not have a distinguished "invalid" value. If OilIsValidOp returns *false,* then the specified string is output with the line number and character position of the corresponding '+'.

Rule *rDyadic,* under the comment "% Minilax Expressions" employs three OIL library functions to complete the operator identification process. (The multiple *xExpr's* occurring in the rule are numbered left to right in the attribute computations.) The function *OilIdResultTS2* computes the set of possible result types given the operator indication *xBinop.indication* and the set of possible result types *xExpr*[2].*pt* and *xExpr*[3].*pt* that are computed in descendant expression nodes. *OilIdOpTS2* performs the actual operator identification operation given the final type of *xExpr*[1] computed in an ancestor node, the operator indication, and the possible types of the operands. *OilGetArgType* computes the final types of the operands given the operator *xBinop.operator.*

Figure 4. Information flow over the tree

Figure 5. Structuring specifications

`Minilax.lido`	Visibility and program structure
`Declaration.lido`	Declarations
`Type.lido`	Primitive types
`Expression.lido`	Expressions
`Operator.lido`	Operator indications
`Identifier.lido`	Defining and applied occurrences of identifiers
`Leaf.lido`	Basic symbols

(a) The content of file `translate.specs`

% Minilax operators
```
RULE rPlusOp: xBinop ::= '+'
STATIC
   xBinop.indication:=Plus;
CONDITION Assert(OilIsValidOp(xBinop.operator),
   'Invalid operand(s) for this operator');
END;
...
```
% Minilax Expressions
```
RULE rDyadic: xExper ::= xExpr xBinop xExpr
STATIC
   xExpr[1].pt:=OilIdResultTS2(xBinop.indication,xExpr[2].pt,xExpr[3].pt);
   xBinop.operator:=
      OilIdOpTS2(xExpr[1].ft,xBinop.indication,xExpr[2].pt,xExpr[3].pt);
   xExpr[2].ft:=OilGetArgType(xBinop.operator,1);
   xExpr[3].ft:=OilGetArgType(xBinop.operator,2);
END;
...
```
(b) Partial contents of files `Operator.lido` and `Expression.lido`

`minilax.gla`	Non-literal basic symbols and comments
`minilax.con`	Concrete syntax
`minilax.sym`	Relationship between phrases and tree nodes
`minilax.cull`	Phrases not reflected in the tree

(a) The content of file `structure.specs`

```
xIdentifier:     ADA_IDENTIFIER
xInteger:        $[0-9]+[mkint]
xReal:           PASCAL_REAL
                 PASCAL_COMMENT
```
(b) The content of file `minilax.gla`

Smoothing Tool Interaction

Eli smooths the interaction between tools in two ways: It removes redundancy from the specifications, and it hides all of the processing that might be needed to prepare the output of one tool for input to another. Both simplifications are important in reducing the load on the user. It is the Odin expert system that enables Eli to combine arbitrary tools written by different people into a smoothly functioning unit. A node in Odin's derivation graph represents a manufacturing step: the process of applying a particular tool to a particular set of inputs and creating a set of outputs. The interface to each tool is a Unix™ shell script. This script is used to run the off-the-shelf tool and to make that tool conform to Eli's environment. Thus, tools developed by different people with different conventions can be combined into an integrated system, even when only executable versions of those processors are available. Since Odin, and not the user, decides when a particular manufacturing step is needed, knowledge of the individual tools can be concealed. Hiding the tools from the user not only simplifies the task of generating a compiler, but also makes system evolution much less painful. In this section we will use the tools associated with the structuring task to illustrate both simplifications.

There are four subtasks of the structuring task listed in the third column of Table 1: scanning, conversion, parsing and tree construction. Conversion procedures (which convert identifiers into unique internal representations and obtain the values of constants) are usually extracted from a library. Eli generates scanning and parsing procedures from specifications provided by the user. The relationship between these specifications shows how Eli smooths tool interaction by eliminating redundancy.

Removing Redundancy from Specifications. The scanner generator and the parser generator each require a list of the basic symbols of the source language. A distinct integer code is associated with each basic symbol, and this code is assigned by the scanner and used to tell the parser which basic symbol was recognized. The actual values of the codes are not important, so long as each basic symbol code is unique and the codes associated with a given basic symbol by the scanner and the parser are the same. If the user were preparing input specifications for both of these tools individually, the list of basic symbols and encodings would have to be produced and then supplied to each. Clearly there is no need for an Eli user to supply encodings; a tool can generate them and add them to the specifications for the scanner and parser generators. In fact, the Eli user does not even directly supply the list of basic symbols. Eli extracts the set of literal terminals, such as **begin,** from the context-free grammar that describes the phrase structure so the user only needs to describe the nonliteral terminals, such as integers and identifiers, and the structure of comments.

Figure 5a lists the files that specify the Minilax structuring task. From these files, Eli can generate the scanning and parsing procedures, determine which conversion procedures to invoke from the scanner, and determine where the parser should call tree construction procedures.

Figure 5b describes Minilax nonliteral basic symbols and comments. The identifier that precedes the colon is the name used for a nonliteral basic symbol in the accompanying context-free grammar. Since comments do not appear in the grammar, no identifiers are given for them. This figure specifies that identifiers are like those of Ada, while real denotations and comments are like those of Pascal. Integers are specified by a regular expression, and an integer conversion processor, named mkint, which is part of the Eli library. Notice how, with the exception of the `xInteger` specification, the user has drawn on existing library specifications to avoid detailed descriptions of these nonliteral basic symbols and comments. (The `xInteger` specification is given in this way solely to illustrate the underlying specification mechanisms. Normally it would be written as `xInteger: PASCAL_INTEGER`.) The library specifications not only describe the scanning characteristics, they also name the appropriate conversion processors. Thus, Figure 5b defines the complete lexical analysis task for Minilax's nonliteral basic symbols and comments. The act of mentioning a nonliteral terminal symbol, such as `xIdentifier`, in the context-free grammar and in the `.gla` specification permits it to be used in other parts of the specification. This can be contrasted with Yacc, where some amount of effort is required to make nonliteral basic symbols available to the processor.

Since the literal basic symbols are enclosed in apostrophes in the concrete syntax of Figure 1, it is a simple matter for an appropriate tool to extract them. In fact, the extraction tool is a bit more sophisticated. It not only extracts the literal basic symbols, but also the nonliteral basic symbol names—identifiers that no not appear on the left-hand side of any grammar rule—in order to verify the consistency of the two specifications.

Using Output of One Tool as Input to Another. If the parser generator were being used independently, the compiler designer would need to insert calls on tree-building functions into the context-free grammar of Figure 1. This is not required of the Eli user because the output of another tool, the attribute grammar processor, is used to obtain the necessary information. The process, though intricate, illustrates how Eli smooths the interaction between two tools by transforming the output of one into input for the other.

The attribute grammar describes the structure of the abstract syntax tree that represents the source program. By examining the attribution rules, the attribute grammar analyzer can output a context-free grammar with embedded calls to procedures that build the tree nodes. A separate tool uses the files `minilax.sym` and `minilax.cull`, shown in Figure 5a to understand the relationship between this grammar, which represents the abstract syntax, and the grammar of Figure 1, which represents the concrete syntax. [1]. That tool then rewrites Figure 1, embedding the tree construction calls in the proper positions. Finally, the modified version of Figure 1 is supplied as input to the parser generator.

Eli's use of the relationship between the concrete and abstract syntaxes is another example of specifying an instance of a problem rather than a solution method. If the designer is going to use an abstract syntax to describe the source program tree, the designer must understand the relationship between that abstract syntax and the concrete syntax that describes the phrase structure of the source program text. Once this relationship is specified, there is no need for any further knowledge. In contrast with Yacc, the designer does not need to know how trees are built (if they are built), or what the names of the procedures that build them are. There is no need to design structures to represent the tree nodes. No decisions about where to place calls in the concrete syntax need be made. All of this tedious, error-prone decision making is carried out by an Eli component whose very existence is unknown to the designer.

Evolution

The third barrier to tool use noted in the introduction was the poor performance of tool-generated compilers. There are two general approaches to attacking this problem: changing the tool and changing the specification technique. An individual tool can be selected and the module it generates compared with the module in hand-coded compilers [18]. The tool is modified on the basis of this comparison to bring the code it generates more into line with that produced by hand. Such modifications may change the relationship of the tool's output to the remainder of the compiler, and may necessitate changes in the overall compiler generation strategy.

One possible reason for changing a specification technique is that some weakness in the specification language deprives Eli of the information it needs to generate an efficient compiler. Another, more subtle reason is that the technique requires the designer to describe a particular solution method rather than an instance of the problem. If the particular solution method is inherently inefficient, then the generated compiler will exhibit this inefficiency. Changing a specification technique involves addition of a new tool or tools, and almost certainly requires that the overall compiler generation strategy be altered.

We have seen that a complete compiler description is made up of many specifications, processed by a variety of tools. A change in one specification technique or tool should not affect other specifications or tools. The structure of Eli makes this kind of graceful evolution possible. For example, our original GLA lexical analyzer generator provided very high-level specification capability, based on our assumption that users would appreciate the ability to specify complex lexical structures without using regular expression [6]. This version of GLA was installed in the Eli system, and used for the first several years of operation. When users of the system complained that the specification language was too inflexible, the GLA package was rewritten [3].

We changed the specification technique to use a library of specifications of common programming language lexical items, but to permit a user to specify lexical items by regular expressions or by explicit scanning code when the need arose. This necessitated changes in the interface between GLA and the rest of the system, which were easily accommodated by modifying the Odin derivation graph. Thus the new GLA provided users additional flexibility in describing the lexical properties and internal representation of basic symbols, without requiring them to understand the changes to the tool interface.

Experience with ELI

We have used Eli extensively as a tool in its own development, in support of course work, and to develop language processors. This experience includes generation of processors for normal programming languages, for extensions to standard languages, and for special-purpose languages. In this section we present several case studies. Our intent is to demonstrate the broad application of Eli, give an idea of the increase in productivity over hand coding, and indicate the quality of the generated processors.

Programming Languages

Compilers for existing programming languages have been developed to provide data for reliable performance evaluations. Four kinds of comparisons could be made with an existing hand-coded compiler for the same language:

- The size of the specifications can be compared to the size of the hand-coded source text.
- The specification development effort can be compared to the development effort for hand-coding.
- The object code sizes can be compared.
- The time to process a specific set of test programs can be compared.

When carrying out such comparisons, it is important to guarantee that the specification and the hand-coded compiler have exactly the same functionality, and that they use roughly the same algorithms. In this section we consider compiler specifications for two programming languages: ALGOL 60 and ICON. Each of these languages has a compiler that is extensively described in literature, so that it is relatively easy to check for equivalent functionality.

ALGOL 60. A compiler was constructed by one graduate student as a class project in six weeks based on Randell and Russell's Whetstone ALGOL [13]. He had previously taken the graduate compiler construction course (which was not based on tools) and had industrial compiler experience. He had studied the ALGOL 60 report in a graduate programming languages course, but had no other experience with ALGOL. Since his compiler did not follow the design given in Randell and Russell's book, because that design does not follow the Eli model, the six weeks included a complete redesign. An experienced software engineer, who had worked with the Whetstone compiler on the ICL KDF9, estimated that he could carry out a new implementation of Randell and Russell's compiler in twelve weeks.

ICON. Similarly, for our semester-long graduate compiler tools course, a translator was constructed for Icon [4]. This translator produces as output intermediate code which is then interpreted. Of particular concern in this project was a comparison between the automatically produced compilers of Eli and hand-written ones. For this reason, the Eli-produced translator was designed to duplicate the behavior of part of the standard Icon implementation [5].

It is estimated that the total development time was about one-third of that which could be expected if the translator was built from scratch in C. The standard version contains (Icon-dependent) modules, for example, lexical analysis and tree construction. Since Eli already provides many of these tasks in language-independent modules, specifications relating to lexical and syntactic analysis are about one-third of the size of the corresponding code in the standard version. Code size of the portions relating to semantic analysis and code generation is about the same in the two versions. The Eli specifications are at a higher level, however, which greatly reduced the development time.

Run-time comparison, using a large set of test programs, showed that the Eli-generated version is only about 50% slower than the hand-coded version. Future efforts will involve a more efficient attribute evaluator, which is expected to reduce this difference significantly with no change to the specifications.

Programming Language Extensions

Eli is ideal for implementing processors for extensions to programming languages because it is possible to begin with specifications that already generate a correct compiler for the base language. These specifications are much easier to modify than the code for an existing compiler would be, because they describe instances of the compiler subproblems rather than ways in which those subproblems were solved. Therefore, the designer of the extension needs only to change the description so that it fits the new instance of the problem engendered by the extension.

Dino is a superset of C designed for writing parallel programs for distributed memory machines [14, 15]. A Dino compiler that translates Dino code into C code targeted for the Intel IPSC family of hypercubes has been developed using the Eli system. In order to save development time, an existing C specification, which was previously written using Eli, was augmented with the Dino constructs. The resulting specification was roughly three times as large as the original C specification. The main benefit that Eli has supplied to the project was the support of a relatively high-level perspective on the construction of the compiler. This has provided the users of Eli two advantages over modifying the Portable C Compiler for instance. First, the time required to understand the Eli C specification was much less than that required to understand the Portable C Compiler specification. This is quite important, since the implementation of the Dino modifications is very dependent on the compiler being modified. Second, it was much easier to make changes in the structure of the language, since the constructs of the Eli system hide a great deal of the tedium that the user of the Portable C Compiler is forced to handle, greatly aiding the language design process.

Since the Dino compiler is based on a fairly old C specification, it uses none of the more recent Eli library facilities such as the DDL compiler or OIL. Almost all of the compiler tasks are described by the attribute grammar. Compiler support code consists of an early version of the Eli environment module, a module for doing machine-dependent arithmetic, and an abstract data type used for processing C typedef constructs.

Special-Purpose Languages

The compiler for OIL shown in Figure 3 was one of the Eli components implemented via Eli. OIL is typical of the little languages that can provide high leverage for certain problems. The implementation was carried out in about eight weeks by a student with little compiler construction experience.

OIL is based on the method for overload resolution developed for Ada compilers [9, 12]. The OIL compiler produces a module that provides functions for operator identification and for determining coercion sequences. It also creates an initialized database for use by those functions, and an interface specification that can be included by any module invoking the functions.

The specifications used to define the OIL compiler follow the same pattern as those for Minilax. They comprise descriptions of the nonliteral basic symbols, the phrase structure of an OIL specification, and the structure and decorations of the source program tree. Values for some of the decorations are computed by the library environment module, while others are computed by modules written in C specifically for this compiler.

Reducing Specification and Code Sizes

By analyzing the structure of compilers such as those we have described, common problems can be identified, their solutions studied and efforts can be made to produce standard modules which solve the problems in a general way. Areas where this has happened in the past include the definition table (DDL) and operator identification (OIL).

The production of external modules for use in individual compilers provides an opportunity for similarities to be identified and general additions to the Eli library to be designed. For example, we have now developed a general facility for output—each of the compilers described here contained their own such modules. Rather than have each compiler writer code his or her own file management and output functions by hand, a library module can now be invoked.

Future Work

Hand-coded compilers often avoid constructing one or both of the trees we have discussed, or they write them in a linearized form to secondary storage. Optimizing compilers usually build explicit versions of both data structures, but may identify common subexpression nodes to convert the target tree into a directed acyclic graph.

Strategies that either avoid tree construction entirely or use linearized versions in secondary storage restrict the amount of information available to make certain decisions. For example, suppose neither tree is built explicitly. This means that only information available "earlier" in the program can be used in making any decision. The structure of a standard Pascal program was defined specifically to avoid requiring a Pascal compiler to build either tree explicitly: All identifiers must be declared before they are used, with the exception of the domain type of a pointer. (Since information about the domain type of a pointer is not needed to process type declarations, and the declaration must appear before the end of the type section in which the pointer is declared, the compiler's constraint is met in this case as well.)

None of the specification techniques used by Eli *require* that explicit trees be built. The attribute grammar describes the information flow in terms of a tree, but it can be shown that if the flow has certain patterns, then all attribute computation can be done during parsing. Other flow patterns can be realized by compilers that write linearized versions of the trees to secondary storage. Our current tools are able to give the user feedback about the data flow patterns in a given specification, guiding a modification to satisfy any constraint that user wishes to place on the data flow. We are beginning to think about how to generate a procedure to carry out parse-time attribution in those cases where it is possible, and how to integrate that procedure with the remainder of the generated compiler.

Eli's open nature makes experiments with a wide variety of tools possible. For example, attribute-grammar-like techniques for describing data flow in directed acyclic graphs have been reported in the literature [16]. Tools for processing such specifications could be added to Eli, allowing experiments to test their practicality for real compiler generation problems.

Conclusions

Eli is a complete, flexible compiler construction system. Based on existing tools, it is an open system that is able to evolve as new tools and techniques become available. Eli does not rely on any one specification language to describe all of the subproblems of compiler construction; rather it provides a coherent framework in which specifications written in a number of languages are combined to describe a complete product. A simple user interface provides a uniform method for requesting derivations from specifications. Such requests only involve product names. Eli determines the particular set of tool invocations on the basis of the state of its cache, as reflected in a knowledge base.

We have used Eli to create compilers for small, special-purpose languages, standard programming languages and extensions to existing languages. It has improved our productivity and has enabled inexperienced users to undertake and complete significant compiler projects. We believe that Eli effectively addresses the problems that have prevented widespread use of compiler construction tools. Our current research program is aimed at further simplifying the use of Eli itself and improving the performance of the generated compilers.

Acknowledgments
Portions of this research were funded by the U.S. Army Research Office under contract number DAAL 03-86-k-0100.

References

1. Bahrami, A. CAGT—An automated approach to abstract and parsing grammars. MS thesis, Department of Electrical and Computer Engineering, University of Colorado, Boulder, Colo., 1986.
2. Clemm, G.M. and Osterweil, L.J. A mechanism for environment integration. *ACM Trans. Prog. Languages and Syst. 12* (Jan. 1990), 1–25.
3. Gray, R.W. A generator for Lexical analyzers that programmers can use. In *Proceedings of USENIX Conference* (June 1988).
4. Griswold, R.E. and Griswold, M.T. *The Icon Programming Language.* Prentice Hall, Englewood Cliffs, N.J., 1983.
5. Griswold, R.E. and Griswold, M.T. *The Implementation of the Icon Programming Language.* Princeton University Press, Princeton, N.J., 1986.
6. Heuring, V.P. The automatic generation of fast lexical analyzers. *Softw.—Prac. Exp. 16* (Sept. 1986), 801–808.
7. Hext, J.B. Compile-time type matching. *The Comput. J. 9* (Feb. 1967), 365–369.
8. Johnson, S.C. Yacc—Yet another compiler-compiler. Computer Science Tech. Rep. 32, Bell Telephone Laboratories, Murray Hill, N.J., July 1975.
9. Kastens, U. Code generation based on operator identification. Bericht der Reihe Informatik Nr. 49, Universitat-GH Paderborn, Paderborn, FRG, Jan. 1988.
10. Kastens, U. LIDO—A specification language for attribute grammars. Betriebsdatenerfassung, Fachbereich Mathematik-Informatik, Universitat-GH Paderborn, Paderborn, FRG, Oct. 1989.
11. Lesk, M.E. LEX—A Lexical analyzer generator. Computing Science Tech. Rep. 39, Bell Telephone Laboratories, Murray Hill, N.J., 1975.
12. Persch, G., Winterstein, G., Dausmann, M., and Drossopoulou, S. Overloading in preliminary Ada. *SIGPLAN Not. 15* (Nov. 1980), 47–56.
13. Randell, B. and Russell, L.J. *ALGOL 60 Implementation,* Academic Press, London, 1964.
14. Rosing, M. and Schnabel, R.B. An overview of Dino—A new language for numerical computation on distributed memory multiprocessors. CU-CS-385-88, Department of Computer Science, University of Colorado, Boulder, Colo., Mar. 1988.
15. Rosing, M., Schnabel, R.B., and Weaver, R.P. Dino: Summary and Examples. In *Proceedings of the 1988 Conference on Hypercube Concurrent Computers and Applications,* ACM Press, 1988.
16. Sonnenschein, M. Graph translation schemes to generate compiler parts. *ACM Trans. Prog. Languages and Syst. 9* (Oct. 1987), 473–490.
17. Waite, W.M. Use of attribute grammars in compiler construction. In *Attribute Grammars and their Applications,* vol. 41, P. Deransart and M. Jourdan, Eds., Springer Verlag, Berlin, 1990.
18. Waite, W.M. and Carter, L.R. The cost of a generated parser. *Softw.—Prac. Exp. 15* (Mar. 1985), 221–239.
19. Waite, W.M., Grosch, J., and Schroer, F.W. Three compiler specifications. GMD-Studien Nr. 166, Gesellschaft fur Mathematik und Datenverarbeitung, Karlsruhe, FRG, Aug. 1989.
20. Waite, W.M., Heuring, V.P., and Kastens, U. Configuration control in compiler construction. In *Proceedings of the International Workshop on Software Version and Configuration Control,* Teubner (Stuttgart, FRG, 1988).

CR Categories and Subject Descriptors: D.2.6 [**Software Engineering**]: Programming Environments; D.3.2 [**Programming Languages**]: Language Classifications—*specialized application languages*; D.3.4. [**Programming Languages**]: Processors

General Terms: Languages
Additional Key Words and Phrases: Compiler generators, language design, text processors

About the Authors:
ROBERT W. GRAY received a Ph.D. in computer science from the University of Colorado in 1990. His research interests include performance evaluation, compiler construction tools and graphics. **Author's Present Address:** US WEST Advanced Technologies, 4001 Discovery Drive, Boulder, CO 80303

STEVEN P. LEVI received his Ph.D. in computer science at the University of Colorado, Boulder. His research interests include configuration management, software engineering environments, programming language design and implementation, and process programming. **Author's Present Address:** IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

***VINCENT P. HEURING** is an assistant professor of electrical and computer engineering at the University of Colorado, Boulder. His major research interests are programming languages, computer architecture, and optical computing.

ANTHONY M. SLOANE is a Ph.D. student in the Department of Computer Science at the University of Colorado, Boulder. His research interests include programming language design and implementation using tools, software development environments and human-computer interaction.

WILLIAM M. WAITE is a professor of electrical and computer engineering at the University of Colorado, Boulder. His major research interests are the design and translation of programming languages.

Authors' Present Address: Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80309-0425. email: compiler@boulder.colorado.edu

*All correspondence should be addressed to Vincent P. Heuring