ICS 2002: Workshop on Databases and Software Engineering

An Effective Approach to Compiler Construction Based on Attribute Grammars and Reusable Components

Stephen S. Yau† Pei-Chi Wu‡ (contact author) Ji-Tzay Yang* Feng-Jian Wang*

†Department of Computer and Information Sciences
Arizona State University, U.S.A.
yau@asu.edu

‡Department of Computer Science and Information Engineering
National Penghu Institute of Technology, Taiwan, R.O.C.
Address: 300 Liu-Ho Road, Makung, Penghu 880, Taiwan, R.O.C.
TEL: +886-6-9264115 ext. 3506
FAX: +886-6-9277361
pcwu@npit.edu.tw

*Department of Computer Science and Information Engineering
National Chiao Tung University, Taiwan, R.O.C.
{jjyang, fjwang}@csie.nctu.edu.tw

ABSTRACT

There are two general approaches to software reuse: compositional and generative. Compositional techniques for compiler construction have less been explored, because most compiler parts were thought to be language dependent and hard to reuse. After years of development, however, applying attribute grammars (AGs) to construction of production-quality compilers has not been successful. One major reason is that traditional AG specification methods are too primitive for real specification tasks. This paper presents an approach that integrates both generative and compositional techniques for compiler construction. The approach uses AGs as the compiler backbone and employs reusable components in AG specifications. By extending AGs to incorporate modularity, remote access, collective computing, and object-oriented views on tree nodes, compiler specifications can be more modular and concise. By employing reusable components to handle circular dependency and non-tree structures, specification difficulties due to the theoretical limitations of AGs can be easily removed. The effectiveness and efficiency of this approach are addressed.

Key Words: semantic analysis, modularity, object-orientation, software components, compiler generators.

1. INTRODUCTION

Compilers are getting bigger and more complex. One force driving compiler front-end evolution is the evolution of programming languages. For example, the C language has evolved into the C++ language [6] with object-oriented features (classes, templates, etc.). Ada has also been revised to support object-orientation, mega-programming, and real-time programming [31]. Compiler designers need a software architecture that can be reused as programming languages evolve. In general, there are two approaches to software reuse [27]. Generative reuse, which generates software from a given specification, is based on a specific language and a program generator. Compositional reuse composes software from existing building blocks.

Compositional techniques in compiler construction have less been explored, because most parts of a compiler were thought to be language dependent and hard to reuse. Generative techniques work well in some compiling tasks. For example, GNU CC uses a parser generated by a parser generator and a generator of code-generators for several target machines. However, few production-quality compilers adopt generative techniques for semantic analysis due to the lack of widely-accepted specification techniques. Most semantic analyses in production-quality compilers work in one of the following ways: recursive descent parsing/analysis, action routines, and attribute grammars (AGs). The first way, e.g., lcc [9], is fully hand-coded, the second, e.g., GNU CC, is partly hand-coded, and only the last, e.g., Linguist-86 [7], is fully generative.

AGs seem to be better than the other hand-coded methods due to their theoretical simplicity. After years of development, however, applying AGs to construction of production-quality compilers has not been successful [34]. One major reason is that traditional AG specification methods are too primitive for real specification tasks. For example, Waite [34] suggests object-orientation as a new direction for AG specifications. Kastens [21] emphasizes the modularity and remote access issues of AGs. Another reason is that AGs cannot easily handle circular dependency or non-tree structures, which, unfortunately, appear in many important tasks like symbol processing and data-flow analysis. For these tasks, current generative techniques cannot generate codes that compete in efficiency with those coded according to specific algorithms. Consequently, compositional reuse techniques may be more appropriate for constructing components for these compiling tasks.

A two-fold problem faces compiler designers: 1) the lack of a specification tool (and a clear methodology) for writing AG specifications and 2) the lack of reusable components for compiler construction, e.g., components for name analysis [19]. In this paper we present an effective approach to compiler construction based on AGs and reusable components. This approach adopts AGs as the compiler backbone and uses reusable components in AG specifications. By extending AGs to incorporate modularity, remote access, collective computing, and object-oriented views on tree nodes, compiler specifications can be more modular and concise. By employing reusable components to handle circular dependency and non-tree structures, the specification difficulties due to theoretical limitations of AGs can be easily removed. Compiler construction can be greatly simplified using our approach.

2. RELATED APPROACHES

2.1 Recursive-descent parsing, action routines, and attribute grammars

Recursive descent parsing is a top-down parsing technique, where semantic actions can be inserted within recursive procedures. This technique can directly coordinate control sequences of semantic actions (e.g., error recovery, symbol table management, etc.) in the body of a recursive procedure. It can easily be incorporated into programming languages that provide recursive procedures. However, it has many disadvantages as well. First, the codes for syntax analysis and semantic actions are highly coupled. Semantic action codes need to be rewritten for even slight changes in syntax. Second, all the semantic actions are hand coded and prone to error.

The first problem in recursive descent analysis can be solved by applying *action routines*, which incorporate formal syntax rules with semantic action symbols. Action routines can be adapted to bottom-up or top-down parsing techniques. The control of semantic actions is guided by parsing actions. In some cases, this simplifies both parsing and the control sequence of semantic actions. Many action routines follow a Start-Process-Finish sequence and keep a semantic record or a stack during processing. However, semantic stack operations are still have to be hand coded.

Attribute grammars (AGs) are well known as a formal technique for compiler construction. Most AG generators use an abstract syntax tree as an intermediate representation. Each symbol on the tree is associated with some (semantic) attributes. The semantic specifications in AGs are

production-oriented: defining attribute computations (attribution rules) associated with production rules. The value of an attribute cannot be accessed outside the context of the production rule. Thus a production rule can be considered as a specification unit, where an inherited attribute is an input and a synthesized attribute is an output of the context of the production rule.

AGs outperform action routines because computation of attributes has no side effects and is independent of parsing actions. Still, AGs have their own problems. Except for some AG extensions that are considered in the next section, fundamentally, AGs are purely generative: A powerful generator is needed to apply the technique. To develop compilers designers need to learn a specification language that may be totally different from programming languages. Thus, it is not surprising that using AGs in semantic analysis of production-quality compilers is still not widely accepted, although AGs are theoretically better than the other two methods.

2.2 AG Specification Extensions

In this section, we list systems related to AG extensions and make a comparison. Table 1 shows the current extensions of AGs, addressed in previous research into rectifying some AG shortcomings. These extensions are categorized into: modularity, object-orientation, remote access, and collective computing. The following are the typical specification extensions in previous research. *Separate specification* allows users to define the attributes and attribution rules of a symbol in more than one file (from different semantic aspects). *Inheritance* construct lets users define the attribution rules of a symbol by composing from those of other symbols. An *upward remote attribute* denoted *X.a* accesses the nearest ancestor of symbol *X* and attribute *a*. *Constituents* [18, 21] represent a set of attributes in the descendants of a symbol. A *chaining* [18, 21] is a left-to-right attribution. A *bucket brigade* [17] is a bi-directional chaining. A *target language's expression* [11, 13] is an expression evaluated into a reference to a tree node in accessing a remote attribute. *List attribution* assigns the inherited attribute of each son.

Table 1. Summary of existing systems on extensions of AGs.

AG system	*Modularity*	*Object-orientation*	*Remote access*	*Collective computing*
ALADIN/GAG [18]	separate spec.	--	upward attribute constituents	chaining constituent(s) list attribution
Lido/Eli [21]	separate spec.	multiple	upward attribute constituent(s)	chaining constituent(s)
Linguist [7]	--	--	--	--
Regular Right-Part AGs [17]	--	--	--	list attribution bucket brigade
SSL [28]	separate spec.	--	upward attribute	--
TOOLS [23, 24]	--	single	pass variable	family
OLGA [16]	separate spec. piped AGs	--	upward attribute	list attribution collect sons' attrs
Door AGs [11]	--	single	target's expr.	list attribution door
Ag [13, 14]	separate spec.	multiple	target 's expr.	thread (chaining)
Scan Grammar [29]	--	--	--	scan
Extended AGs [36]	--	--	--	--
Modular AGs [5]	textual matching on productions	--	--	pattern matching on productions
Composable AGs [8]	component grammars	--	--	--

separate spec. = separate specification; single/multiple = single/multiple inheritance;
upward attribute = upward remote attribute; target's expr. = target language's expression.

Besides these common extensions, Extended AGs [36] use a concise AG notation to express semantic predicates. A Modular AG [5] consists of a number of *patterns* associated with a set of *templates*. A template specifies the attribution rules to be generated for each matching production rule. A Composable AG [8] is built from some *component AGs*, each of which models a particular sub-domain. The interconnections of component AGs are through input/output attributes specified by a glue grammar. De Moor et al. [4] classify three types of AG components: families, rules and aspects. Multiple AG inheritance [25] allows an AG to inherit the specifications from ancestors: adding or overriding specifications from ancestors. Hedin [12] also addresses an extension to AGs, permitting attributes to be references to arbitrary nodes in a syntax tree.

2.3 Other Extensions

Demers *et al*. [3] propose a method in which message propagation is decided by a *successor function*, a mapping between tree nodes. OOAG [30] is an approach that integrates AGs and object-oriented programming. The static semantics is specified in AGs; the dynamic semantics is defined using message passing, which may cause side effects. Attribute Coupled Grammars [10] solve the modularity problem by providing more higher level AG compositions: A *compilation* is decomposed into a number of AGs,

each of which reads an attributed tree and generates an attributed tree. This is an AG extension for multi-pass compiling. Higher-Order AGs [32, 33] start from another direction: promoting abstract syntax trees to "first class citizens." The model allows an attribute to be an abstract syntax tree called *non-terminal attribute* (NTA), which can be attributed again. Attribute Coupled Grammars can be treated as a special case of Higher-Order AGs where only the root contains an NTA. Circular AGs (e.g., [15]) allow circular-attribute dependency. The power of fixed-point attribute evaluation can elegantly solve a collection of attributes involved in a dependency circle. In Conditional AGs [2], attribution rules may have guards. Rules are active only when their guards are satisfied. They can express well-behaved computations that involve circular attribute dependency. Kikuchi and Katayama [22] propose generalized AGs based on using type-0 grammars in place of context-free grammars. They regard generalized AGs as constraint satisfaction systems.

3. AN INTEGRATED APPROACH TO COMPILER CONSTRUCTION

As addressed in Section 2, many AG-based specification techniques have their own extensions, and each tries to rectify some AG shortcomings. It is difficult to combine all these extensions in one language. Instead of focusing on AG extensions, our approach integrates specification constructs and reusable components for compiler construction. With our approach, a compiler specification is done based on a unified AG and a framework of reusable components. This section also addresses the integration of our AG extensions and reusable components.

3.1 Extending the AG Concepts

Traditional AGs describe attribution rules by using local dependency, and attribute computing is applicative. This mathematical model is rather elegant but too primitive for specification use. Here we introduce the following key concepts into AGs to extend their capability to specify:

1) an attribute to be accessed being either local or remote;
2) an attribute being either applicative or state transitional.

A *remote (attribute) access*, a reference to an attribute on a remote tree node, is used as a tool to overcome part of the "modularity" problem. For example, an attribute instance may need to be propagated through a sequence of nodes in corresponding to a number of production rules. Such a propagation sequence can be avoided by introducing a remote access to the propagated attribute. In

other words, specification by remote access seems to be simpler than using lengthy attribute propagation. In [37, Ch. 4], we define *R-AGs*, AGs with a flexible remote access construct that expresses remote attributes. Because an R-AG can be translated into an AG, the resulting specification is still an AG. In addition, a remote access construct can operate collectively, since a remote access may access a collection of attributes. Thus, the design of collective computing construct can also be based on remote access.

Traditional AGs emphasized theoretical elegance (applicative computing), so all attributes were applicative, i.e., their values (or states) remain unchanged. However, for many compiling tasks, it is convenient to let an attribute change its internal data. For instance, a symbol table attribute needs to change its internal value when an operation such as insertion is performed on it. An attribute with this property is called *state transitional*, and the operation on a state-transitional attribute is called an *action*. Describing the explicit dependency among the operations on a state-transitional attribute is a useful constraint to guarantee the correct use of these attributes.

Some AG systems (e.g., Lido) adopt these concepts by providing ad-hoc constructs. For example, many systems support explicit (action) dependencies by treating them as special attribution rules. For example, the *depends_on* operator in an expression that does not define any attribute value or defines a *void* attribute.

3.2 A Unified and Simple AG Specification Extension

As discussed in Section 2, each AG-based system has its own extension constructs. These constructs include separate specification, inheritance, upward remote attributes, chaining, list attribution, and more flexible remote access via target language expressions. One key difficulty in designing a new specification language is that the more new constructs a language contains, the more difficult the language is to use. To solve this problem, our language is designed with a simple unified set of language constructs for AG extensions according to the concepts discussed in Section 3.1. In addition, the language satisfies not only conventional concerns like remote access and collective computing, but also two general software engineering issues: modularity and object-orientation on tree nodes. These four issues try to bridge the gap between a simple theoretical model (i.e., AG) and its practical use (i.e., AG specification).

To address modularity and object-orientation issues, our extension specifies a two-level abstraction: module-level and class-level. A compiler specification can be decomposed into a number of modules, each one a specification unit, containing a number of class definitions and interfaces to scanner, parser, and other specification units. A class is the only semantic specification construct. An *abstract class* defines common properties for a number of (sub)classes but cannot be directly used to create a tree node object. The class in our language is more powerful than a production rule associated with attribution rules in AGs, thus the production rule construct is replaced and supported with class and related constructs. The module and class constructs can also support separate specification. Consider a symbol *X* that involves name analysis, type checking, and code generation. By separate specification, symbol *X* may be defined in several specification files. With our unified specification constructs, class *X* can inherit the multiple classes *X_name*, *X_typechk*, and *X_codegen*, which define different semantic aspects in three corresponding modules. Table 2 summarizes how we address the modularity and object-orientation issues in the unified specification.

Table 2. Using the unified specification extension for various modularity and object-orientation features.

Features	Unified specification
specification unit	module
interface to other spec. files	import/export lists
interface to scanner	token definitions
interface to parser	syntax rules
attribution rules on each production	class definitions
composition of attribution rules from several spec. units	using inheritance to compose a class with multiple bases from some modules
production rule	Class
inheritance	class's base classes
abstract syntax tree	class's components
attribute	class's attribute declaration
attribution rule	class's attribute definition
remote access	class's remote site
non-terminal attribute	object ID of tree node
semantic function	class's member function

To address remote access and collective computing issues, we allow unified constructs to specify various features according to the key concepts defined in Section 3.1. As shown in Table 3, upward remote attribute access and constituents can be described as *site expressions* [37, Ch. 4], a generalized mechanism for remote access. A chaining can be described using constituents to access the tree nodes involved in the chaining and computing the information according to the order of the attributes of tree

nodes. A list attribution can be described in another direction: getting the attribute originally assigned to each son, i.e., by an upward remote attribute access from each son. A family can be developed using a symbol table, which is state transitional and needs some explicit dependency to guide correct use of the table.

Table 3. Using the unified specification extension for various remote access and collective computing features.

Feature	Unified specification
upward attribute	site expressions
constituent(s)	site expressions
target's expression	site expressions / tree node references
chaining	constituent(s), according to the order of attributes
bucket brigade	constituent(s), according to the [reverse] order of attributes
list attribution	using upward attribute in sons
collect attributes of sons	chaining
pass variable	chaining
scan	chaining with an associative operator
family	using symbol table (state-transitional attribute)
door	state-transitional attribute

3.3 Writing a Semantic Specification with Reusable Components

Many reusable components are state transitional. The use of these components as attributes in AGs integrates generative and compositional techniques in compiler construction. The following shows the use of both AG specification (AGs with extensions addressed in Section 3.2) and reusable components.

The use of AG specification:

- supports architectural specification/design of compilers. AGs provide various decomposition methods, and well-decomposed AGs serve as the backbones of compilers.
- incorporates with mature parser generators. AG specifications need not consider which parsing algorithm is used.
- deals with tree-structured problems: AG generators automatically generate evaluation order and algorithms. They can also optimize attribute storage.
- supports domain-specific patterns and analysis techniques. AGs can support domain-specific patterns, such as remote access and collective computing. AGs can also provide rigorous analysis techniques, such as circularity testing and grammar ambiguity checking.

The use or construction of reusable components:

- deals with problems of diverse structures, e.g., stacks, hash tables, and flow graphs.
- employs specific algorithms or optimizations that are not easily produced by program generators. Hash functions and interval data-flow analysis are representative.
- reuses components from mature domains. Many problem domains are mature enough to have been formalized and cataloged as reusable compiler components. These components can be refined and continuously improved.

We have developed a number of reusable software components for compiler construction, including an identifier table, symbol tables, data-flow analysis components, and an interface for code generation. These components can be used as attribute types in AG specifications, and the attributes defined using these components are all state transitional. Applying these well-accepted concepts and their components in AG specifications can greatly reduce specification efforts.

The only work needed to use a reusable component is specifying the dependency among the operations on the component. Each dependency can be specified explicitly and elegantly in class-level specification: An action-dependency sequence involving a number of tree nodes can be specified in the "ancestor" of these nodes. For example, let a block node be defined with a symbol table attribute, and a number of declaration nodes as its components. Each declaration node needs to append an entry to the symbol table and thus this side-effect is defined as an action. The sequence of these actions can be coordinated by a block node.

4. WRITING AN OBJECT-ORIENTED COMPILER SPECIFICATION

This section presents an example of writing an object-oriented compiler specification. Our approach consists of four steps:

1 Modularization (decomposition of syntax rules): writing token definitions and production rules.

2. Identification of semantic classes and adding the linkages between syntax rules and semantic classes.

3. Specification of semantic classes: modeling objects with inheritance, components, attributes, protocols, remote sites, and attribution rules. Advanced features of objects include object ID, type ID, dynamic cast, and target language expressions for remote accesses.

4. Use of software components: using components as attributes, using external (global) components, passing tree-node references as parameters, and specifying dependency on the actions of state-transitional components.

An example language

The example language is a sequence of definitions and expressions, an extension of [1, Fig. 4.57]. In the language, a *definition* defines the name of a constant and its value. A constant name cannot be redefined. There are five kinds of expression on floating numbers: addition, subtraction, multiplication, division, and negation. The compiler is to output a sequence of values of input expressions according to the input order. Figure 1 shows an example of the language input.

```
pi = 3.14159
radius = 100.0
area = pi * radius * radius
area
length = 2 * pi * radius
length
```

Figure 1. An input in the expression language.

Modularization

Figure 2 shows the module `expr` using the specification language in [38]. The module `expr` contains a lexical part with two token definitions (`IDENTIFIER` and `NUMBER`) and a syntactical part with the production rules for the expression. The module construct integrates scanner, parser, and AGs. Note that for a simple language, a single specification module is enough to handle the specification. For a large language (e.g., more than 500 production rules) further decomposition is needed.

```
%MODULE expr
%LEXICON
/* The following is in Lex format */
/* macro definitions in Lex */
D  [0-9]
E  ([Ee][-+]?{D}+)
DE ([Dd][-+]?{D}+)
ALPHA    [a-zA-Z_]
ALPHANUM ({ALPHA}|{D})
IDENT    ({ALPHA}{ALPHANUM}*)
%%
/* matching rules in regular expressions */
[\ \t]   ;           /* skip blanks and tabs */
{D}+("."{D}+)?{E}  @NUMBER; /* put a code to create a terminal */
{D}+("."{D}+)?{DE} @NUMBER;
{IDENT}    @IDENTIFIER;
%SYNTAX
%%
Lines: Ls
     ;
Ls   : /* empty */
     | Ls Line
     ;
Line : Expr '\n' { $$~PrintExpr }
     | Definition
     ;
Definition
     : IDENTIFIER '=' Expr '\n'
     ;
Expr : Expr '+' Expr { $$~AddExpr }
     | Expr '-' Expr { $$~SubExpr }
     | Expr '*' Expr { $$~MulExpr }
     | Expr '/' Expr { $$~DivExpr }
     | '(' Expr ')'  { $$=$2 }
     | '-' Expr { $$~NegExpr }
     | NUMBER   { $$=$1 }
     | IDENTIFIER { $$~IdExpr }
     ;
%SEMANTICS
  ...
%END expr
```

Figure 2. The `expr` module.

Identification of semantic classes

There are a number of semantic classes: `IDENTIFIER`, `Definition`, `PrintExpr`, `Lines`, and expression classes, including `Number`, `AddExpr`, `SubExpr`, `MulExpr`, `DivExpr`, `NegExpr`, and `IdExpr`. The following is a brief description on these classes. Some expression classes are discussed latter.

- Class `NUMBER` represents floating numbers.
- Class `IDENTIFIER` keeps the unique symbol for an identifier.
- Class `IdExpr` refers to a constant name using an `IDENTIFIER`.
- Class `Definition` defines a new constant name `IDENTIFIER` with a value of `Expr`.
- Class `PrintExpr` prints out an `Expr`.

- Class `Lines` contains a number of `PrintExpr`'s and `Definition`'s.

As shown in Figure 2, these semantic classes are assigned to the corresponding production rules in the syntactical part.

Class hierarchy of semantic classes

Figure 3 shows the expression classes and their inheritance relationships. Class `Expr` is defined with an attribute `value`; class `UnaryExpr` is defined with a component (part) of `Expr`; class `BinaryExpr` has two `Expr` components. Because the value of a `NUMBER` can also be used in an expression, it is also an expression.

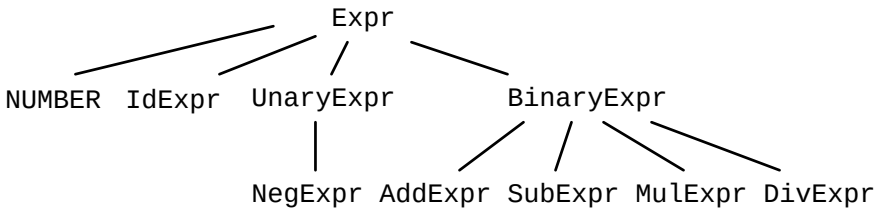

Figure 3. Semantic class hierarchy.

Using software components

The specification needs two software components: an identifier table (`IdnTable`) and a plain symbol table (`PlainTable`). The `IdnTable` stores identifiers used in the language, and the `PlainTable` is used to store constant names and values and to retrieve the values of constants defined. Figure 4 shows the `PlainTable` interface. The `add` and `lookup` operations return a `flag` indicating the result of the operation.

```
template<class keyT, class valueT>
class PlainTable : public uTable<keyT, valueT>
{
  friend class PlainTableIter<keyT, valueT>;
  typedef uEntry<keyT, valueT> Entry;
public:
  PlainTable();
  ~PlainTable();
  flag add(keyT k, valueT v);
  flag add(keyT k, valueT v, Entry*& ent);
  flag lookup(keyT k) const;
  flag lookup(keyT k, Entry*& ent) const;
};

enum flag { OK, DUPLICATE, CONFLICT, NOTFOUND, AMBIGUOUS, KEYFOUND };
```
Figure 4. The `PlainTable` interface.

Specification of semantic classes

There are two terminal classes: `IDENTIFIER` and `NUMBER` (see Figure 5). Class `IDENTIFIER` accesses an external `IdnTable` that stores the identifier text. The value of a `NUMBER` comes from the text of the matched string. The external C function (`sscanf`) is used to convert a string into a floating number.

```
%SEMANTICS
/* header to be included in semantic analysis */
/* external C function: sscanf() */
%{
#include <stdio.h>
IdnTable<Power2_Hash> idtbl();
%}
TERMINAL IDENTIFIER
ATTRIBUTE
  char* @sym;
STATIC
  @@sym = idtbl.makeIdn(@text);
END IDENTIFIER

TERMINAL NUMBER INHERITS Expr
STATIC
  BLOCK sscanf(@text, "%lf", &@@value); END
END NUMBER
```

Figure 5. Terminal classes `NUMBER` and `IDENTIFIER`.

```
CLASS Expr
ATTRIBUTE double @value;
END Expr
CLASS BinaryExpr INHERITS Expr
COMPONENT e1:Expr; e2:Expr;
END BinaryExpr
CLASS AddExpr INHERITS BinaryExpr
STATIC
  @@value = @e1.value + @e2.value;
END AddExpr
CLASS SubExpr INHERITS BinaryExpr
STATIC
  @@value = @e1.value - @e2.value;
END SubExpr
CLASS MulExpr INHERITS BinaryExpr
STATIC
  @@value = @e1.value * @e2.value;
END MulExpr
CLASS DivExpr INHERITS BinaryExpr
STATIC
  @@value = @e1.value / @e2.value;
END DivExpr
CLASS UnaryExpr INHERITS Expr
COMPONENT e1:Expr;
END UnaryExpr
CLASS NegExpr INHERITS UnaryExpr
STATIC
  @@value = - @e1.value;
END NegExpr
```

Figure 6. Expression classes.

Figure 6 shows the definition of some expression classes. Each class derived from `UnaryExpr` and `BinaryExpr` defines how the value of the expression is computed.

Class `Lines` (Figure 7) consists of two components: `exprs`, a number of `PrintExpr`'s, and `defs`, a number of `Definition`'s. Each is defined using a remote access. A `Lines` contains two additional member functions (message protocols): `define`, which defines a constant name, and `value`, which returns the value of a constant. Both member functions access the private attribute `consts` of `Lines` (a `PlainTable`). A `Lines` object coordinates the `print` actions of `exprs` in left-to-right (`PREORDER`), line by line. In addition, all `const` actions of the underlying `Const` objects are ordered as left-to-right and bottom-up (`POSTORDER`).

```
CLASS Lines
COMPONENT
  exprs = CONSTITUENTS PrintExpr;
  defs = CONSTITUENTS Definition;
SITE
  consts = CONSTITUENTS Const;
PRIVATE
  PlainTable<char*, double> @@consts;
PROTOCOL
  void @@define(char* sym, double val)
  {
    flag f = @consts.add(sym, val);
    if (f==DUPLICATE) Message("duplicate constant name");
  }
  double @@value(char* sym)
  {
    Entry* ent;
    flag f = @constns.lookup(sym, &ent);
    if (f==NOTFOUND) {
      Message("constant name not found");
      return 0;
    }
    else
      return ent->value();
  }
STATIC
  PREORDER @exprs.print;
  POSTORDER @consts.const;
END Lines
```

Figure 7. Class `Lines`.

Class `Const` (Figure 8) is an abstract class defined with the action `const` and the site `scope` for the enclosing scope. Classes `IdExpr` and `Definition` both access the root (`Lines`) and inherit from class `Const`. Class `IdExpr` has an `IDENTIFIER` component. `IdExpr` looks up the scope for the value of the constant name. Class `Definition` has two components, an `IDENTIFIER` and an `Expr`. Class `Definition` defines a constant in the enclosing scope. Class `PrintExpr` contains one expression. The duty of class `PrintExpr` is to print out the value of the expression. Since this operation causes side effect, it is defined as an action. An external C function `printf` is used.

```
CLASS Const
SITE
  scope = INCLUDING Lines;
STATIC
  ACTION const: END;
END Const
CLASS IdExpr INHERITS Expr, Const
COMPONENT
  id:IDENTIFIER;
STATIC
  ACTION const: @@value = @scope.value(@id.sym); END
END IdExpr
CLASS Definition INHERITS Const
COMPONENT
  id:IDENTIFIER;
  Expr;
STATIC
  ACTION const: @scope.define(@id.sym, @Expr.value); END
END Definition
CLASS PrintExpr
COMPONENT
  Expr;
SITE
  ids = CONSTITUENTS IdExpr;
STATIC
  ACTION print: printf("%g\n", @@Expr.value); END;
END PrintExpr
%END expr
```

Figure 8. Classes `IdExpr`, `Definition`, and `PrintExpr`.

5. IMPLEMENTATION, EFFECTIVENESS AND EFFICIENCY

5.1 Implementation

Our system has been divided into two subsystems: an ag++ generator and a libag++ library. All the components in libag++ have been developed using C++ class, class templates, and functions. The code generation interface now generates C code compiled under GNU CC.

A parser for the ag++ specification language has been developed. The scanner part of the parser has been developed using condition states in Lex. The parser needs to partially match the codes written in C++ and Lex regular expressions. There are two complete specifications, `Expr.m` and `Pascal-.m`, parsed by the parser. We are currently developing the complete generator that generates the visit-oriented attribute evaluator. For parse-time evaluation, we have developed the code to locate the free-positions in a grammar [26]. These free positions can be used to place some semantic actions for attribute computation.

5.2 The Effectiveness in Compiler Specification

The following is a comparison of two specifications for a Pascal subset, one using Lido as given in [35] and the other using ag++ as given in [38]. Our specification is much smaller due to its extensive use of

class construct, simpler explicit dependency, object ID, and type ID. Figure 9, for example, shows a specification for `ConstNameUse`. The class `ConstNameUse` multiply inherits class `Constant`, which defines the attribute `value`, and class `NameUse`, which looks up the enclosing scope for the identifier and defines the attribute `ref`. The `ref` is checked to determine whether it is a `Constant` using the type ID of the node pointed by `ref`. Most explicit dependencies are removed and the corresponding dependencies are defined in the enclosing constant definition part.

```
ATTR Type: DefTableKey;
ATTR Value: int;

RULE IdnConst: Constant ::= ConstantNameUse
COMPUTE
  Constant.Type=
    GetType(ConstantNameUse.Key,NoKey) DEPENDS_ON Constant.Objects;
  Constant.Value=
    GetValue(ConstantNameUse.Key,0) DEPENDS_ON Constant.Objects;
  IF(NE(GetKind(ConstantNameUse.Key,Constantx),Constantx),
    Message(FATAL,"Constant name required"))
    DEPENDS_ON Constant.Objects;
END;
```

```
CLASS ConstNameUse INHERITS NameUse, Constant
STATIC
  BLOCK Assert(@ref IS Constant,
           Message(FATAL, "Constant name required") ); END
  @@type = @ref.type;
  @@value = @ref.value;
END ConstNameUse
```

Figure 9. A comparison of the specifications of class (symbol) `ConstNameUse`. The specification on top is the excerpts from *Pascal-* [35, p.36]; ours is on bottom.

In addition to addressing the technical issues in Sections 2 and 3, we carefully consider the current computing environment in designing ag++ and libag++. Lex and Yacc syntax have been used in the lexical and syntactical parts of our generator, and C++ language has been used to write the code for semantic definitions. This approach, of course, can also be applied to other scanner generators, parser generators, and target languages. The current version specifically addresses Lex-Yacc-C++ to encourage compiler designers who presently use UNIX tools in their work to shift to ag++ and libag++.

5.3 The Efficiency of a Generated Compiler

We take a two-fold approach to discussing the efficiency of the generated compiler: 1) considering the efficiency of generated attribute evaluators, and 2) examining the efficiency of software components in libag++ .

The ag++ AG generator is based on a visit-oriented evaluator [20], a kind of evaluator in wide use as a very efficient subset of well-formed AGs. The parse-time evaluation technique promises to significantly improve the efficiency of the generated attribute evaluator. Another issue, attribute storage optimization, is addressed by the use of remote access, which already removes most artificial propagation attributes and reduces the amount of storage required. In addition, we have also used an object stack component to allocate memory for tree nodes and attributes in the evaluator [39].

The efficiency of software components in libag++ depends on the algorithms and the programming language (C++) used to develop the components. C++ is an efficient object-oriented language and object-oriented characteristics will not introduce much overhead if the programming guidelines in C++ are followed.

The algorithms of these components are designed carefully and are comparable with the codes in production-quality compilers, such as lcc [9] and GNU CC. For instance, symbol tables are developed using hash tables. This approach is very fast in comparison with a table that uses a linked list of declaration nodes. The data-flow analysis components now use the iterative method, which is quite efficient when the control-flow graph is simple. Moreover, because the implementation of a component can be directly replaced without changing its interface, the efficiency of those components can be improved when a new and faster algorithm is used.

6. CONCLUSIONS

We have presented an approach to improving compiler construction that integrates generative and compositional techniques. It simplifies specification and improves reuse potentially. Our preliminary result shows that integrating generative and compositional techniques promises both effectiveness and efficiency in compiler construction.

ACKNOWLEDGMENTS

This research was partly supported by National Science Council, Taiwan, R.O.C., under Contract No. NSC 89-2213-E-346-002.

References

1. Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

2. Boyland, J.T., "Conditional attribute grammars," *ACM Transactions on Programming Languages and Systems*, Vol.18, No.1, Jan. 1996, pp.73-108.
3. Demers, A., Rogers, A., and Zadeck, F. K., "Attribute Propagation by Message Passing," *ACM SIGPLAN '85 Symp. on Language Issues in Programming Environments*, June 1985, pp. 43-59.
4. De Moor, O., Backhouse, K., Swierstra, S. D., "First-class attribute grammars," *Informatica*, Vol.24, No.3, June 2000, pp.329-341.
5. Dueck, G.D.P. and Cormack, G.V., "Modular Attribute Grammars," *The Computer Journal*, Vol.33, No.2, 1990, pp. 164-172.
6. Ellis, M.A., and Stroustrup, B., *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.
7. Farrow, R., "Generating a Production Compiler from an Attribute Grammar," *IEEE Software*, Vol.1, No.4, Oct. 1984, pp. 77-93.
8. Farrow, R. and Marlowe, T.J., Yellin, D.M., "Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation," In *Proceedings of 19th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, New Mexico, USA, Jan. 19-22, 1992, pp. 223-234.
9. Fraser, C.W., and Hanson, D., "A Retargetable Compiler for ANSI C," *ACM SIGPLAN Notices*, Vol.26, No.10, Oct. 1991, pp.29-43.
10. Giegerich, R., "Composition and Evaluation of Attribute Coupled Grammars," *Acta Informatica*, Vol.25, 1988, pp.355-423.
11. Hedin, G., *Incremental Semantic Analysis*, Dept. Computer Science, Lund University, Sweden, March 1992.
12. Hedin, G., "Reference attributed grammars," *Informatica*, Vol.24, No.3, June 2000, pp.301-317.
13. Grosch, J., "Ag - An Attribute Evaluator Generator," Report No.16, Compiler Generation Project, GMD Forschungsstelle an der Universitaet Karlsruhe, 1989.
14. Grosch, J., "Multiple Inheritance in Object-Oriented Attribute Grammars," Project Compiler Generation Report No.28, GMD Forschungsstelle an der Universitaet Karlsruhe, 1992.
15. Jones, L.G., "Efficient Evaluation of Circular Attribute Grammars," *ACM Trans. on Programming Languages and Systems*, Vol.12, No.3, July 1990, pp.429-462.
16. Jourdan, M., and Parigot, D., "Internals and Externals of the Fnc-2 Attribute Grammar System," *Attribute Grammars, Applications and Systems*, LNCS No. 545, Springer-Verlag, 1991, pp. 485-504.
17. Jullig, R. K., DeRemer F., "Regular Right-Part Attribute Grammars," *ACM SIGPLAN '84 Symp. on Compiler Construction*, June 1984, pp. 171-178.
18. Kastens, U., Hutt, B., and Zimmermann, E., *GAG: A Practical Compiler Generator*, LNCS No.141, Springer-Verlag, 1982.
19. Kastens, U. and Waite, W.M., "An Abstract Data Type for Name Analysis," *Acta Informatica*, Vol.28, 1991, pp.539-558.
20. Kastens, U., "Implementation of Visit-Oriented Attribute Evaluators," *Attribute Grammars, Applications and Systems*, LNCS No. 545, Springer-Verlag, 1991, pp. 114-139.
21. Kastens, U. and Waite, W.M., "Modularity and Reusability in Attribute Grammars," Technical Report CU-CS-613-92, Dept. Computer Science, University of Colorado at Boulder, Sept. 1992.
22. Kikuchi, Y., Katayama, T., "On generalization of attribute grammars," *Systems and Computers in Japan*, Vol.27, No.9, Aug. 1996, pp.33-42.
23. Koskimies, K., Nurmi, O., and Paakki, J., "The Design of a Language Processor Generator," *Software-Practice and Experience*, Vol. 18, No. 2, Feb. 1988, pp. 107-135.
24. Koskimies, K., "Object-Orientation in Attribute Grammars," *Attribute Grammars, Applications and Systems*, LNCS No. 545, Springer-Verlag, 1991, pp. 297-329.
25. Mernik, M., Lenic, M., Avdicausevic, E., Zumer, V., "Multiple attribute grammar inheritance," *Informatica*, Vol.24, No.3, June 2000, pp.319-328.
26. Ouyang, S.-T., Wu, P.-C., Wang, F.-J., "Locating Free Positions in LR(k) Grammars," *Journal of Information Science and Engineering*, Vol. 18, No. 3, May 2002, pp.411-423.
27. Prieto-Diaz, R., "Status Report: Software Reusability," *IEEE Software*, Vol.10, No.3, May 1993, pp.61-66.

28. Reps, T. and Teitelbaum, T., *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer-Verlag, Newyork, 1989.
29. Reps, T., *Scan Grammars: Parallel Attribute Evaluation Via Data-Parallelism*, TR 1120, Computer Sciences Department, University of Wisconsin-Madison, November 1992.
30. Shinoda, Y. and Katayama, T., "Object Oriented Extension of Attribute Grammars and Its Implementation Using Distributed Attribute Evaluation Algorithm," *Attribute Grammars and their Applications*, LNCS No. 461, Springer-Verlag, 1990, pp. 177-191.
31. Taft, S.T., "Ada 9X: A Technical Summary," *Communications of the ACM*, Vol.35, No.11, Nov. 1992, pp.77-82.
32. Teitelbaum, T. and Chapman, R., "Higher-Order Attribute Grammars and Editing Environments," *ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation*, 1990, pp. 197-208.
33. Vogt, H.H., Swierstra, S.D., and Kuiper, M.F., "Higher Order Attribute Grammars," *ACM SIGPLAN'89 Conf. on Programming Language Design and Implementation*, 1989, pp. 131-145.
34. Waite, W.M., "Use of Attribute Grammars in Compiler Construction," *Attribute Grammars and their Applications*, LNCS No. 461, Springer-Verlag, 1990, pp. 255-265.
35. Waite, W.M., "A Complete Specification of a Simple Compiler", Technical Report, Department of Electrical and Computer Engineering, University of Colorado, Boulder, 1993.
36. Watt, D. A., "Extended Attribute Grammars," *The Computer Journal*, Vol. 26, No. 2, 1983, pp. 142-153.
37. Wu, P.-C., *Integrating Generative and Compositional Techniques in Compiler Construction*, Ph.D. Thesis, Department of Computer Science and Information Engineering, National Chiao Tung University, Taiwan, R.O.C., June 1995.
38. Wu, P.-C., Wang, F.-J, and Yang, J.-T., "A Semantic Specification Language for Compiler Construction," *The Proceedings of the National Science Council Part A: Physical Science and Engineering*, Vol. 20, No. 1, 1996, pp. 23-41.
39. Wu, P.-C., and Wang, F.-J., "The Experience in Fine-Tuning a Compiler Generated from an Attribute Grammar," *Journal of the Chinese Institute of Electrical Engineering*, Vol. 4, No. 1, Jan. 1997, pp.75-82.