A Specification Language Based on Attribute Grammar for Compiler Construction

Ji-Tzay Yang Feng-Jian Wang

Department of Computer Science and Information Engineering
National Chiao-Tung University
Hsinchu, Taiwan, R.O.C.

Abstract

Attribute grammars (AGs)[2][8] are a useful formalism for the specifications of compiler. Recently, success on compiler construction systems based on AG[4][5] shows the feasibility to use AG as the core of a compiler construction. In this paper, a specification language AG++ is presented to help specify a compiler in AGs. AG++ supports object orientation and helps integration between compiling phases such as lexical scanning, syntactic analysis, parsing tree construction, and semantic analysis. In addition, AG++ introduce several new constructs for help with writing terse compiler specification. A prototype of AG++ specification translator -- AG++/C is also presented to transform these constructs into AG primitives and generate compiler source files

Keywords: attribute grammars, compiler generation, object orientation, specification language

1. Introduction

Although the compiler structures are getting complicated, the compiler technologies are getting mature to overcome the complexity. It is common to apply compiler generators such as scanner generator, parser generator in several stages of compiler construction. Traditional generators for compiler construction were usually regarded as convenient but slow tools before. [3] showed that a program generated scanner, whose source code is automatically optimized by applying complicated analysis, gives better performance over hand-coded ones. Successful experiences on producing commercial compiler product by applying compiler generating tool were also reported. Thus, it is feasible to construct an efficient compiler product with a specification language based on AG theory and state-of-art compiler generating tools.

We present an AG-based specification language AG++ which introduces several new high-level AG constructs as well as facilitates the progress of compiler construction theory. In an AG system using AG++, compiler desingers can use off-the-shelf compiler generators and AG++ together to construct a complete compiler with the flexibility of AG++.

Features of AG++ also support software development methodologies, such as modular decomposition and object-orientation. The *modular* construct adopted in AG++ helps a large and complicated compiler specification to be decomposed into several smaller managable modules. For example, a C++ grammar[11] can be decomposed into the following modules: keywords, expressions, declarations, declarators, class declaration, statements, pre-processor, templates, and exception handling. The object-orientation in AG++ is characterize by its class construct, inheritance and message passing mechnism, used for describing parsing tree nodes. Consequetly, primitives used in object-oriented design methodology such as class, inheritance, message passing can be directly expressed by AG++. Reuse of AG++ specifications can be also achieved by the modular and object-orientation constructs of AG++.

By inspecting compiler specifications written in AGs such as [7], we can extract several useful AGs computation patterns. Examples include those for accessing non-local attributes and duplicating code for nodes with similar behavior. These patterns are abstracted in AG++ by means of higher-level constructs. The abstraction can be implemented with the higher-level AG constructs, as in section 4, to produce more concise compiler specification.

In section 2, the attribute grammar and compiler construction systems based on AG are introduced. The AG++ model and its design decision are presented in section 3. In section 4 a set of higher-level constructs and their examples are presented. Finally, an implementation of AG++/C and a complete AG++ specification are presented in section 5.

2. Attribute grammars

An attribute grammar is based on a context-free grammar $G=(N,T,P,Z)$, where N is a finite set of nonterminals, T is a set of terminals, P is the set of production rules, and Z is the starting symbol. In an attribute grammar, each symbol $X \in N \cup T$ is associated with a set of attributes $A(X)$ which represents the property of the symbol X. The attribute a, which is an element of $A(X)$, is denoted by $X.a$.

A set of of attribution rules $R(p)$ which defines the value of attributes is associated with a production rule p. Each attribute should be defined by exactly one attribution rule r in $R(p)$ of the form:

r: $X_{m_0}.a_0 \leftarrow f(X_{m_1}.a_1, \ldots, X_{m_k}.a_k)$, where
p is defined as $X_0 \rightarrow X_1 \ X_2 \ .. \ X_n$,
a_i is the attribute associated with symbol X_{m_i},
$0 \le m_i \le n$, and $0 \le i \le k$.

An evaluator for AG computes the attribute values defined by the attribution rules according to the evaluation plan which can be obtained by analyzing the non-circular data dependency graph of attributes.

Several tools and systems based on AG have been developed to help compiler construction. Eli[4] and Cocktail[5] are famous AG-based systems from which some significant compilers were built. Many AG systems are associated with a structured language to specify the attributes declaration and attribution rules. LIDO is a specification language in Eli system, which provides high-level AG constructs and inheritance. An extensive survey for AG systems[10] has summarized the features appeared in various AG-based compiler generating system.

3. The characteristics of AG++

3.1. AG++ model

AG++ is a language mainly focused on semantic specification; it also supports lexical and syntax specification. An AG++ compiler such as AG++/C reads the AG++ specification, and then generates the compiler (source) code in traditional programming language such as C or C++.

Figure 1 shows an AG++ environment based on C++ and one external compiler generator. It depicts the components and the relationships among them when building a compiler product. A compiler designer can specify his compiler in AG++ for constructing compiler front-end. When writing a specification in AG++, the desinger can store the compiler specification facilitated with AG++'s object-orientation for future reuse. The specification can also be helped with existing AG++ library. The C++ class library in the figure can be any "off-the-shelf" one. Without loss of generality, for the rest context, we use *lex* and *yacc* to replace the scanner and parser generators in the figure respectively.

Here, we choose AG++/C as the role of AG++ compiler in Figure 1. With respect to the AG++ users, AG++/C can be regarded as a code generator which abstracts the analysis of AG dependency and generation of evaluation sequence. In addition, AG++/C translates our higher-level constructs in the AG++ specification into AG primitives. The AG evaluator stub performs the scheduled attribute evaluation rules at run time of the compiler product.

As in Figure 2, AG++ compiler consists of the following basic components:

- AG++ front-end, which parses the AG++ specification and constructs the partially-decorated parse tree.
- Attribute analyzer, which analyzes and flattens the attributes and attribute evaluation rules described in the semantic class hierarchy. It also generates auxiliary attribute evaluation rules and dependency to support the higher-level constructs of AG++.
- Dependency analyzer, which computes the visit sequences of the AG evaluator according to the attribute dependency flattened by the attribute analyzer.
- Code generator, which outputs the code of lexical analyzer, parser generator, and AG evaluator for an AG++ specification.

3.2. Design decisions for AG++ language

The syntax and functionality of AG++ has experienced several refinements over years[13][14][15][16]. During previous years, AG++ was mainly improved by adding new features and lifting the restriction of the langauge. Many design decisions have been made behind the revision of AG++:

Application domain with a specific pattern: AG++ is a specification language for semantic analysis and translation for compiler construction. When writing a semantic specification for semantic analysis, the compiler designer usually needs to access attributes of remote nodes and collects attributes scattered in the parsing tree nodes. Such programming patterns can be supported directly by the AG++ language constructs.

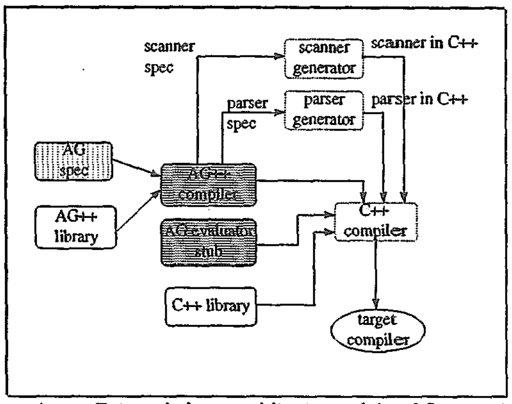

Figure 1. External view: architecture of the AG++ system

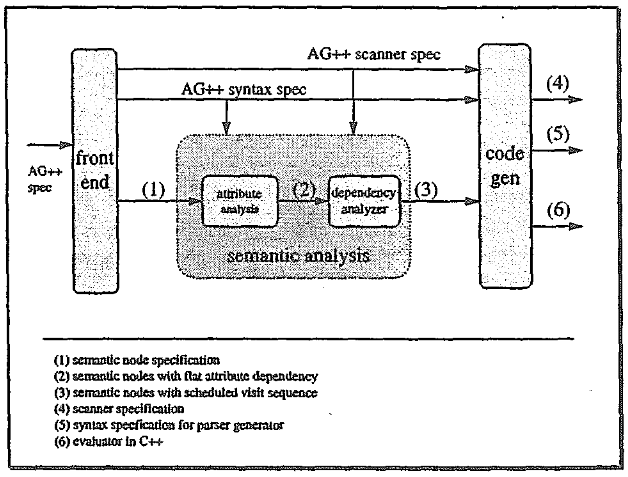
AG++ scanner spec
AG++ syntax spec
AG++ spec
front end
(1)
attribute analysis
(2)
dependency analyzer
(3)
semantic analysis
code gen
(4)
(5)
(6)
(1) semantic node specification
(2) semantic nodes with flat attribute dependency
(3) semantic nodes with scheduled visit sequence
(4) scanner specification
(5) syntax specfication for parser generator
(6) evaluator in C++

Figure 2. Internal view: basic components of the AG++ compiler

Design guideline: Keep the language features as small as possible. A new feature is included into AG++ only if no exsiting features to achieve the new feature efficiently. Nevertheless, current version of AG++ is inevitably larger than any other contemporary AG specification notation due to the rich language constructs proposed by AG++.

- ***Object Abstraction***: Although the goal of AG++ does not become an object-oriented specification lanauge, its underlying paradigm is much influenced by the object-oriented concept. For instance, the inheritance, aggregation, message passing in object-oriented paradigm are mapped into AG++ primitive *INHERITS*, *COMPONENT*, and *PROTOCOL* respectively.
- ***Modularity***: Modular constructs of an AG specification language help the decomposition and reusability of AG specification. [17] also discussed the modularity and reusability of attribute grammars. AG++ provides the primitives *MODULE*, *IMPORT*, and *EXPORT* for this reason.
- ***Integration With Reuse***: Some tasks in compiler construction are not suitable when applying generative techniques totally. AG++ is designed to support compositional-reuse technique such that the reusable software component can be integrated with AG++ seamlessly.
- ***Attribute Evaluation In Target Language***: Most AG specification langauge request the users to write attribute evaluation rules in a specialized langauge. In contrast, AG++ is designed as extensions to the hybrid of lex, yacc, C++, and our higher-level constructs, users with lex, yacc, and C++ experience can easily become acquainted with constructing compiler in AG++.

4. Language constructs introduced in AG++

Most works on AG system are concerned more with the efficiency of the system (i.e. on the performance of the AG evaluator). However, in designing AG++ system, we paid more attention on providing higher-level constructs to reduce the efforts for writing semantic analysis specification. The following subsections give an overview the AG++ language constructs. Appedix A lists part of the AG++ grammar in BNF.

4.1. Modularity

AG++ allows a compiler specification to contain multiple AG++ (semantic) modules. The constructs *EXPORT* and *IMPORT* define the scope of identifiers in semantic classes. The following code fragment describes that the identifier *SymA* for some semantic class in the module *foo* is globally accessible, while the identifier *SymB* for some semantic class is imported from the module *bar*.

```
MODULE foo
EXPORT SymA ;
FROM bar IMPORT SymB ;
```

Figure 3. An example of using modular construct

4.2. Lexical and syntactical specification

Constructs in Figure 4. demonstrates that users can specify the lexical and syntactical specification in a similar way like lex and yacc in AG++. The abstract, context, and terminal semantic classes mentioned here are to be described in section 4.3.

The syntax of attribute specification for AG++ can be described as follows. That a *terminal semantic class* `item` is to be returned can be described as `@<item>`, i.e., a special symbol `'@'` followed by the term `<item>`. In addition, the string recognized by the scanner must be stored in the pre-defined attribute of `item`. `@<>` denotes that the ASCII value associated with the scanned character is returned to the parser. `@@<abstract_S>` defines the *abstract semantic class* of non-terminal symbol S. `@<one_item>` and `@<two_item>` are the *context semantic classes* assigned to the two production rules { `S` $\rightarrow$ `item` } and { `S` $\rightarrow$ `item item`} respectively.

```
%LEXICON
%{
// place to include libraries for lexical
scanner
%}
White          [\ \n\r\t]
A              [a-zA-Z]
%%
{A}+           @<item>
{White}        @{ cout << "skip\n"; @}
[,]            @<>

%SYNTAX
%{
// place to include libraries for parser
// generator
%}
%%
S                                  @@<abstract_S>
   : item                          @<one_item>
   | item ',' item                 @<two_item>
   ;
```

Figure 4. Specifying lexical and syntactic rules in AG++.

4.3. Semantic classes

Due to object-orientation, the constructs mentioned in this subsection can usually find their corresponding counterparts in the C++ class constructs [17]. A semantic class is a C++-like class describing the behaviors and semantics of compiler objects such as terminal symbols, non-terminal symbols, and production contexts. In AG++, terminal symbols, non-terminal symbols, and production context are described by *terminal semantic classes*, *abstract semantic classes*, and *context semantic classes* repectively. By distinguishing above semantic classes, AG++ compiler can choose proper manipulation for grammar objects according to the their semantic classes. For example, if class `A` is declared as a terminal semantic class, AG++ compiler will automatically assign the scanned string to the inherited variable `text` of semantic class `A`'s instance to help the integration of scanning and parsing stages.

A traditional C++ class consists of two parts to describe the behavior of objects: (1) data member declaration, and (2) function member declaration. A typical semantic class also consists of the following subsections declaring either data or function members of the semantic class:

- *COMPONENT:* The aggregation relationship of a context semantic class is specified.
- *ATTRIBUTE:* The attributes for the semantic class are

declared.

- *PRIVATE:* Local attributes for a semantic class is declared.
- *STATIC*: Attribute evaluation rules are described.
- *SITE:* Reference paths to access remote attributes are defined.
- *PROTOCOL:* Message handlers for a semantic class are defined.

COMPONENT, ATTRIBUTE, and *SITE* subsections together are analog to the data memebers part of a C++ class, while *STATIC* and *PROTOCOL* subsections together are analog to the member functions part of a C++ class. PROTOCOL and SITE will be described further in section 4.4.

A primitive *INHERITS* is introduced to AG++ to support the inheritance hierarchy of semantic classes. In Figure 5, ***binary_expr*** defined in `%SEMANTIC` section is an abstract semantic class using INHERITS construct. It specifies the behaviors of the nonterminal symbol `binary_expr` defined in %SYNTAX section. In addition, ***add_expr*** and ***max_expr*** defined in `%SEMANTIC` section are example use of context semantic classes; they describe the behaviors of the production rules {`binary_expr → expr + expr`} and { `binary_expr → 'max' '(' expr ',' expr ')'` } respectively.

```
%SYNTAX
expr                        @@<expr>
    : binary_expr           @<....>
    | uniary_expr           @<....>
    ;
binary_expr                 @@<binary_expr>
    : expr '+' expr         @<add_expr>
    | 'max' '(' expr ',' expr ')' @<max_expr>
    ;
// ....
%SEMANTIC
#include <iostream.h>
const   int         OP_ADD     = 1;
const   int         OP_SUB     = 2;
#define        max(a,b)
    ((a)>(b)?(a):(b))
// stuff for C++
%%
// abstract class for symbol expr
ABSTRACT CLASS expr
%ATTRIBUTE
@{ int @<result> @}
%STATIC
@{ cout << "The result is" << @<result> <<
endl ; @}
%END
// abstract, for symbol binary_expr
ABSTRACT CLASS binary_expr INHERITS expr
%ATTRIBUTE
@{ int @<lhs>, @<rhs>, @<op> @}
%STATIC
@{
swith(@<op>) {
  case OP_ADD:
    @@<result> = @<lhs> + @<rhs>;
    break;
  case OP_MAX:
    @@<result> = max(@<lhs> , @<rhs>);
    break;
default:
    cout << "Warn: Bad op_type" << endl ;
}
@}
%END
CLASS add_expr INHERITS binary_expr
%COMPONENT    e1 : expr(1), e2 : expr(3)
// alias expr(1) to e1
//          expr(3) to e2
%STATIC
@{ @@<lhs> = @<e1.result> ;        @}
@{ @@<rhs> = @<e2.result> ;        @}
@{ @@<op> = ADD_OP;          @}
%END
CLASS max_expr INHERITS binary_expr
%COMPONENT    e1 : expr(3), e2 : expr(5);
%STATIC
@{ @@<lhs> = @<e1.result> ;        @}
@{ @@<rhs> = @<e2.result> ;        @}
@{ @@<op> = MAX_OP;          @}
%END
```

Figure 5. Using the semantic class inheritance construct.

4.4. Higher-level constructs

According to the definition of traditional AG, a paring tree node can only access the attributes resides in the nodes the node directly connects. Without the help of proper constructs, users of AG systems need introduce lots of auxiliary attributes declaration and attribution rules additionally. The following constructs in AG++ replaced those awkward auxiliary attributes and attribution rules to help the users concentrate on the semantic analysis and produce terse AG specification.

4.4.1. Remote access

For each non-local attribute access, a set of temporary attributes and auxiliary attribution rules should be defined along the accessing path to pull in the non-local attribute. In Eli, the specification language LIDO provides simple constructs *INCLUDING* and *CONSITUENT* to describe part of above remote access. AG++ adopts these constructs and provides a more powerful expression of remote access -- *site expression*[12], for this AG programming pattern.

The construct *INCLUDING(X)* is used to access the nearest parsing tree node X along an straight up-warding path, while the construct *CONSTITUENTS(X)* is used to collect a group of nodes with symbol X in the subtree rooted at the accessing node. Both constructs can only access the nodes along straight paths (i.e., the access path can either going upward or downward.) The site-expression is a regular expression on the alphabet defined as the set of labels for parsing tree arcs. *Site expression* gives more description power over *INCLUDING* and *CONSTITUENTS* in terms of the set of attribute access paths described.

A remote access can be prefixed by a cardinality specifier such as *SINGULAR, AT_MOST_ONE, AT_LEAST_ONE, RANGE(N1, N2)* and *EXACT(N)* to restrict the number of nodes accessed by the remote access expression. Those cardinality specifiers are analog to the regular expression *(pattern), (pattern)?, (pattern)+,* and *(pattern){N1,N2}, and (pattern){N}* respectively.

Figure 6 demonstrates the use of *INCLUDING* and *CONSTITUENT* in AG++. The site expression `remote_c = %INCLUDING(X) Up Downto_3rd_Child` specifies a typical attribute access path which goes up to the nearest node labeled X, goes up to X's parents via the arc labeled `Up`, then goes down to the third child via the arc labeled Downto_3rd_Child.

```
// examples of simple Remote Access and
site-expression
%COMPONENT
  fields = %AT_LEAST_ONE NameList
%CONSTITUENTS(Name)

// %AT_LEAST_ONE is cardinality test
// %POST_ORDER is the traverse order
%SITE
remote_a = %INCLUDING(X);
remote_b = %CONSTITUENTS(Y);
remote_c = %INCLUDING(X) Up
             Downto_3rd_Child   ;
remote_d = %AT_LEAST_ONE %POST_ORDER
             %CONSTITUENTS(Y) ;
....
```

Figure 6. Using remote access and site expression construct

4.4.2. PROTOCOL

A protocol defined inside a *PROTOCOL* section provides an interface for foreign nodes to invoke a sequence of attribute computations of the callee and returns the result to the protocol caller. The *PROTOCOL* section defines which messages to be accepted and how to handle these messages.

Figure 7. shows the application of *PROTOCOL* to compute the difference between the maximum and minimum elements in a set of attributes of two nodes. Instead of using remote access for all the four attributes in *foo* from *bar1* or *bar2* to compute the difference of the maximum and the minimum, a protocol *max_min_diff* is defined in *foo*. The application of protocol can (1) reduce the number of remote access channels and the required auxiliary attributes, (2) reduce the coding efforts, because the code in a protocol such as *max_min_diff* needn't be duplicated into bar1 and bar2, and (3) hide the implementation detail of *foo*'s *max_min_diff* from *bar1* and *bar2*.

```
%SEMANTICS
%{
#include <LEDA/list.h>
// import the C++ class library LEDA
%}
...
// in semantic class's protocol
// section CLASS foo
%ATTRIBUTE
  @{ int @<a1>, @<a2>, @<a3>, @<a4>; @}
%PROTOCOL
   max_min_diff: @{
    int max_min_diff(int b){
    list<int>lst;
    lst.append(@<a1>);
    lst.append(@<a2>);
    lst.append(@<a3>);
    lst.append(@<a4>);
    lst.append(b);
    return lst.max() - lst.min()
    }
    @}
// END OF PROTOCOL max_min_diff

// -------------------------------
CLASS   bar1
%ATTRIBUTE @{ int @<b1>; @}
%SITE se = (site expression to access foo)
%STATICS
   %ACTION @{
   cout << "The max_min diff \
             of node foo and me is:"
        << @@<se.max_min_diff>(@<b1>)
        << endl ;
   @}
....

// -------------------------------
CLASS   bar2
%ATTRIBUTE @{ int @<b2>; @}
%SITE se = (site expression to access foo)
%STATICS
   tag2:
  @{
   @<diff>=@@<se.max_min_diff>(@<b2>);
  @}
....
```

Figure 7. Using *PROTOCOL* construct

4.4.3. SEQUENCE and PARALLEL

The *SEQUENCE* and *PARALLEL* constructs inside a *STATIC* section enclose the attribution rules as nestable blocks. Both constructs describe the evaluation sequence of attribute evaluation rules and indirectly introduce the ***inter-rule attribute dependency*** which involves attribute dependency across attribute evaluation rules, contrast to the ***intra-rule*** attribute dependency due to a single attribute evaluation rule. The rules enclosed by a *SEQUENCE* block must be evaluated sequentially; the rules enclosed by a *PARALLEL* block can be evaluated in parallel.

The attribute evaluation rule following keyword *ACTION* denotes that it has no DO (defined-occurrence) attributes.

Figure 8 is an example demonstrating the use of the constructs to describe the dependencies between the attribute evaluation rules.

```
// a trivial example of sequence and
// parallel constructs to:
//  1.construct a new list joined by
//     two lists from RHS symbols e1 and e2,
//  2.find the maximum and minimum elements
//  3.compute the difference of max and min
%SEQUENCE %{
   tag1:
  @@<elt_list> =
     @<e1.elt_list> + @<e2.elt_list>;
  @}

   %PARALLEL %{
   tag2: @{
   @@<max_elt> =
      find_max_elt(@<elt_list>) ;
   @}
   tag3: @{
   @@<min_elt> =
      find_min_elt(@<elt_list>) ;
   @}
   %}

   tag4:
    @<min_max_diff> =
       @<max_elt> - @<min_elt> ;
   @}
%}
```

Figure 8. Using *SEQUENCE* and *PARALLEL* constructs

5. An AG++/C implementation and a specification example

AG++/C is a language processor for AG++ specification langauge. It should translate and integrate lexical, syntactic, and semantic specifications written in AG++ to the source of correspoding generators or evaluators. Currently, we choose lex, yacc as the output for lexical and syntactic analysis respectively. The output target is changeable when necessary, i.e., it is possible to change the target from lex to more efficient one such as RE2C[3]. The evaluator for AG++ is output as C++ coded after the translation of higher-level constructs and analysis of attribute dependency.

5.1. Environment

A prototype for AG++/C is developmented under UNIX. The user interface for the AG++/C is basically in command line fashion with consideration for furthur integration into GUI such as X-windows. The C++ class library LEDA[9] is chosen as an aid for our implementation due to the massive manipulation on tree structure and data dependency analysis for attributes in AG++/C.

The transformation algorithms for the higher-level constructs of AG++ are developed and under implementation in AG++/C. The tranformation of remote access construct described in section 4.4.1. are generalized by the algorithm proposed by [12]. The transformation alogrithm for constructs described in section 4.4.2. and 4.4.3. can be found in [17].

The algorithms for attribute dependency analysis and evaluator design can be found in [1]. We choose to implement static attribute evaluator that accepts strongly non-circular AG for AG++/C, which gives more efficiency over dynamic attribute evaluator and accepts larger subset of AG according to the survey in [10].

5.2. An AG++ specification example

Figure 9 shows a simple but complete AG++ specification for an arithmatic expression calculator. AG++/C translated the specification and analyzed the attribute dependency to produce a set of compiler source codes in lex, yacc, and C++.

```
%MODULE    expr
// Input for the expression calculator:
//    100 + max (1, 2) + min( 3, 4 )
//         - 10 - max (1 , 2)
// Result:
//    93
%LEXICON
D[0-9]
W[ \t\n\r]
%%
{D}+              @<NUMBER>
("max"|"min")     @<OP_MINMAX>
[+-\*/]           @<>
{W}+              @{
   cout << "Nothing to do.\n" ; @}

%SYNTAX
%start      ans
%left       '+' '-'
%%
ans         @@<ANS>
```

```
 : expr    @<do_ans>
 ;
expr       @@<expr_ctx>
 : OP_MINMAX '(' expr ',' expr ')'
   @<compare_expr>
 | expr '+' expr     @<add_expr>
 | expr '-' expr     @<sub_expr>
 | NUMBER            @<number_to_expr>
 ;
%SEMANTIC
%{
#define    GetMax(a, b)    \
     ((a) > (b)) ? (a) : (b) ;
#define    GetMin(a, b)    \
     ((a) > (b)) ? (b) : (a) ;
%}
%%
//-------------------------------
ABSTRACT CLASS ANS
%ATTRIBUTE @{ int @<result>; @}
%END
//-------------------------------
CLASS do_ans     INHERITS ANS
%COMPONENT expr ;
%STATIC
    @{ @@<result> = @<expr.value>; @}
    @{ cout << "The value is %d\n" <<
       @<result> ; @}
%END
//-------------------------------
ABSTRACT CLASS  expr_ctx
%ATTRIBUTE @{
   int @<value>;
@}
%END
//-------------------------------
CLASS compare_expr INHERITS expr_ctx
%COMPONENT  op_type : OP_MINMAX,
            left : expr(3),
            right : expr(5) ;
%STATIC
@{
  string s = @<op_type.text>;
    if (s == "max") {
      @@<value> =
        @<left.value>, @<right.value> );
    } else if ( s == "min" )
      @@<value> =
        @<left.value>, @<right.value> );
    }
@}
%END
//-------------------------------
CLASS add_expr INHERITS expr_ctx
%COMPONENT
  left:expr(1),  // alias expr(1) to left
  right:expr(3); // alias expr(3) to right
%STATIC
  @{
    @@<value> = @<left.value> +
                @<right.value> ;
  @}
%END
//-------------------------------
CLASS sub_expr INHERITS expr_ctx
%COMPONENT
  left : expr(1), right : expr(3) ;
%STATIC
  @{ @@<value> = @<left.value> -
                 @<right.value> ;
  @}
%END
//-------------------------------
CLASS number_to_expr INHERITS expr_ctx
```

```
%COMPONENT   n : NUMBER ;
%STATIC
   @{ @@<value> = atoi( @<n.text> ) ; @}
%END
//--------------------------------
%END  expr
```

Figure 9. A complete AG++ specification for expression evaluation

6. Conclusion and future work

We present a semantic specification language AG++ which provides higher-level constructs which can avoid writing tedious specification. A prototype implementation of AG++/C which accepts subset of AG++ specification is also presented. In addition to make AG++/C to accept full set of AG++, future directions of the AG system include to provide more useful higher-level constructs as well as to integrate it with GUI environment and other aided tools for a comfortable compiler construction environment. The efficiency of the AG++ system will be achieved by adopting proper compiler generating techniques into AG++ instead of inventing new ones.

Bibliography

[1] Alblas, H., "Attribute Evaluation Methods," *Attribute Grammars, Applications, and Systems,* Lecture Notes in Computer Science, No. 545, Springer-Verlag, 1991.

[2] Alblas, H., "Introduction to Attribute Grammars," in *Attribute Grammars, Applications, and Systems,* Lecture Notes in Computer Science, No. 545, Springer-Verlag, 1991.

[3] Bumbulis, P. and Cowan D.D., "RE2C: A More Versatile Scanner Generator," *ACM Letters on Programming Language and Systems*, Vol 2, Number 1—4, March—December, 1993, pp. 70—84.

[4] Gray, R.W., Heuring, V.P., Levi, S.P., Sloane, A.W. and Waite, W.M., "Eli: A Complet, Fexible Compiler Construction System," *Communications of the ACM*, Vol 35, No. 2, Feb. 1992, pp.121—131.

[5] Grosch, J., "GMD Toolbox for Compiler Construction, also known as Cocktail", Compiler Generation Project, GMD Forschungsstelle an der Universitaet Karlsruhe, 1989. (via anonymous ftp://ftp.gmd.de//gmd/cocktail/)

[6] Kastens, U. and Waite, W.M., "Modularity and Reusability in Attribute Grammars", *Acta Informatica*, Vol. 31, 1994.

[7] Kastens, U., Hutt, B., and Zimmermann, E., "Appendix A: Attribute Grammar for PASCAL" *GAG: A Pratical Compiler Generator*, Lecture Notes in Computer Science, No. 141, Springer-Verlag, 1982.

[8] Knuth, D.E., "Semantics of Context-Free Languages," *Mathematical Systems Theory*, Vol. 2, No. 2, 1968, pp. 127—145.

[9] Mehlhorn, K. and Naher, S., "LEDA: A Platform for Combinational and Geometric Computing," *Communications of the ACM*, Vol. 38, No.1, 1995, pp. 96—102.

[10] Sloane, A.M., "An Evaluation of an Automatically Generated Compiler," *ACM Transactions on Programming Language and System*, Sep, 1995, pp. 691—703.

[11] Stroustrup, B., *The C++ Programming Langguage*, 2nd ed., Addison-Wesley, 1991.

[12] Wu, P.-C. and Wang, F.-J.: "A Generalized Model of Remote Access for Attribute Grammars," Technical report No. CSIE-93-1005, Department of Computer Science and Information Engineering, National Chiao-Tung Unviersity, Taiwan, R.O.C., Oct 1993.

[13] Wu, P.-C. and Wang, F.-J.: "A Semantics Specification Method for Compiler Construction," Technical report No. CSIE-93-1004, Department of Computer Science and Information Engineering, National Chiao-Tung Unviersity, Taiwan, R.O.C., Oct 1993.

[14] Wu, P.-C. and Wang, F.-J.: "An Object-Oriented Specification for Compiler," *ACM SIGPLAN Notices*, Vol. 27, No. 1, Jan. 1992, pp. 85—94.

[15] Wu, P.-C. and Wang, F.-J.: "Applying Classification and Inheritance into Compiling,", *ACM OOPS Messenger*, Vol. 4, No. 4, Oct. 1993, pp. 33—43.

[16] Wu, P.-C., Young, K.-R, and Wang, F.-J., "Generating Compilers Based on an Object-Oriented Specification (OOCS)," in the *Proceedings of 1992 International Computer Symposium (ICS '92)*, Taiwan, pp. 1041—1048.

[17] Yang, J.-T., "The Design and Implementation of a Semantic Specification Language for Compiler Construction," Master Thesis, Department of Computer Science and Information Engineering, National Chiao Tung University, Taiwan, 1995.

Appendix A. AG++ grammar excerption

```
/*------------------------------------   */
/*  BNF of AG++                          */
/*---------------------------------------*/
AG
   : AG_modules
   ;
AG_modules
   : AG_modules AG_module
   |
   ;
AG_module
   :MODULE VAR
    lexicon_section
    syntax_section
    semantic_section
    END_MODULE  opt_VAR
   ;
/* #### Note #### */
/* lexicaon_section and syntax_section
   are omitted in appendix A */
/* #### SEMANTICS #### */
semantic_section
   : SEMANTIC PP_SEPARATOR class_defs
   ;
class_defs
   : class_defs class_def
   | class_def
   ;
class_def
   : class_qualifier  VAR
```

```
     inherit_clause
     class_body
     END_MODULE
   ;
class_qualifier
   : ABSTRACT CLASS
   | CLASS
   | TERMINAL
   ;
inherit_clause
   : INHERITS bases
   |
   ;
bases
   : bases ',' VAR
   | VAR
   ;
class_body
   : class_body class_item
   | class_item
   ;
class_item
   : attr_section
   | component_section
   | class_static
   | site_section
   | private_section
   | protocol_section
   ;
/* #### COMPONET #### */
component_section
   : COMPONENT component_lists ';'
   ;
component_lists
   : component_lists ';' component_list
   | component_list
   ;
component_list
   : component_list ',' component
   | component
   ;
component
   : VAR
   | VAR_MOU VAR
   | VAR_MOU VAR '(' NUMBER ')'
   | site
   ;
/* #### SITE #### */
site_section
   : SITE sites
   ;
sites
   : sites    site ';'
   | site ';'
   ;
site
   : VAR_EQ cardinal travel site_alphabets
   ;
cardinal
   : SINGULAR
   | AT_MOST_ONE
   | AT_LEAST_ONE
   | EXACT '(' NUMBER ')'
   |
   ;
travel
   : PRE_ORDER
   | POST_ORDER
   | IN_ORDER '(' NUMBER ')'
   | REVERSE PRE_ORDER
   | REVERSE POST_ORDER
   | REVERSE IN_ORDER '(' NUMBER ')'
   |
   ;
```

```
site_alphabets
   : site_alphabets site_alphabet
   | site_alphabet
   ;
site_alphabet
   : INCLUDING '(' VAR ')'
   | CONSTITUENTS '(' VAR ')'
   | VAR
   ;
/* #### ATTR #### */
attr_section
   : ATTRIBUTE ATTR_BLOCK
   ;
/* #### PRIVATE #### */
private_section
   : PRIVATE ATTR_BLOCK
   ;
/* #### PROTOCOL #### */
protocol_section
   : PROTOCOL protocols
   ;
protocols
   : protocols protocol
   | protocol
   ;
protocol
   : opt_tag ATTR_BLOCK
   ;
/* #### STATIC #### */
class_static
   : STATIC  static_list
   ;
static_list
   : static_list static_item
   | static_item
   ;
static_item
   : static_compound
   | static_simple
   ;
static_simple
   : opt_tag  ATTR_BLOCK
   | opt_tag  FOREACH VAR OF ATTR_MARK
     ATTR_BLOCK
   | opt_tag ACTION ATTR_BLOCK
   ;
static_compound
   : opt_tag PARALLEL LPBRACE  static_list
     RPBRACE
   | opt_tag SEQUENCE LPBRACE  static_list
     RPBRACE
   ;
opt_tag
   : VAR_MOU
   |
   ;
/* #### End of Grammar #### */
```