Proceedings of International Conference on Distributed
Systems, Software Engineering and Database Systems

SSCC: A Sufficiently Smart Compiler-Compiler

Wuu Yang and Yen-Tsan Liu
Computer and Information Science Department
National Chiao-Tung University, HsinChu, Taiwan, R.O.C.

Abstract. *Attribute grammars are a formalism for specifying computations on syntax trees. SSCC is a practical attribute-grammar system based on a polynomial-time extension of Kastens's ordered attribute grammars. The system comprises of two subsystems. The generation subsystem computes the evaluation order of attribute occurrences in production rules and translates attribute equations into low-level code for a virtual stack machine. The evaluation subsystem invokes tools to perform lexical and syntactic analysis and evaluates the attribute instances during a traversal of the syntax tree. Three features make SSCC capable of performing any desired computation (within the constraints of ordered attribute grammars): user-defined data types, user-defined functions, and the* `finalize` *function. A user may define arbitrary types and functions for use in the attribution equations. After all the attribute instances are evaluated, SSCC calls the* `finalize` *function, which may be supplied by a user, and passes it the whole decorated syntax tree. This offers a user opportunities for further processing the tree and the attributes. The SSCC system is semi-strongly typed in the sense that type consistency within a specification is fully checked; however, type consistency between a specification and user-supplied functions, which are written in the C language, is not.*

Key Words and Phrases: attribute grammars, ordered attribute grammars, compiler compiler

Acknowledgement. This work was supported in part by National Science Council, Taiwan, R.O.C. under grant NSC 85-2213-E-009-051.

1. Introduction

Attribute grammars, first introduced in 1968 [9], have attracted much research interest. Attribute grammars are a very convenient and powerful framework for specifying computations based on (abstract or concrete) context-free grammars. In particular, they may be used to specify the semantics of programming languages since the meaning of a program may be viewed as attributes of its syntax tree and specified in a syntax-directed manner.

An attribute-grammar system is similar to a parser-generator such as *yacc* [6] in that it automatically generates an executable program from a declarative specification. Instead of parsing a context-free language, an attribute-grammar system is concerned with computing attributes of symbols in syntax trees specified by an (abstract or concrete) context-free grammar.

To evaluate attribute instances of a syntax tree, it is usually necessary to perform the evaluation in an order that is consistent with the dependencies among the attribute instances in the syntax tree. This gives rise to the circularity problem of attribute grammars: it is possible to produce circular (and undesirable) dependencies in a syntax tree for certain classes of attribute grammars. Thus, the well-defined attribute grammars (or non-circular attribute grammars), which creates no syntax trees with circular dependencies, are identified. For certain well-defined attribute grammars, there is a fixed evaluation order of all attributes of all symbols that can be applied to all syntax trees derived from the attribute grammar. Kastens identifies such a class of attribute grammars, called the *ordered* attribute grammars (OAG), that allows a polynomial-time decision procedure [7]. Yang et al. improve Kastens's algorithm for a larger class of grammars, called the *extended* ordered attribute grammars, that also allows a polynomial-time decision procedure [18]. Efficient evaluators may be built for (extended) ordered attribute grammars because it is not necessary to keep an attribute dependence graph and to find an evaluation order for each syntax tree. It is argued that most practical attribute grammars belong to the class of OAG [7]. The *SSCC* system is based on the extended ordered attribute grammars.

SSCC (a sufficiently smart compiler-compiler) is a practical attribute-grammar system. *SSCC* defines a specification language for attribute grammars. The language provides a few basic data types. However, a user is free to define any desired types as long as he supplies the related operations on the types. Full static type-checking within the attribute equations is performed by *SSCC*. The specification language also provides a few primitive operations for use in the attribution equations. Again, a user is free to apply any desired additional operations either to clarify the attribute equations or to manipulate user-defined types. The user-defined operations are written in the C language and are supplied by the user. The *SSCC* system automatically makes appropriate links to these user-defined functions during evaluation.

The *SSCC* system, shown in Figure 1, comprises of two subsystems: a generator and an

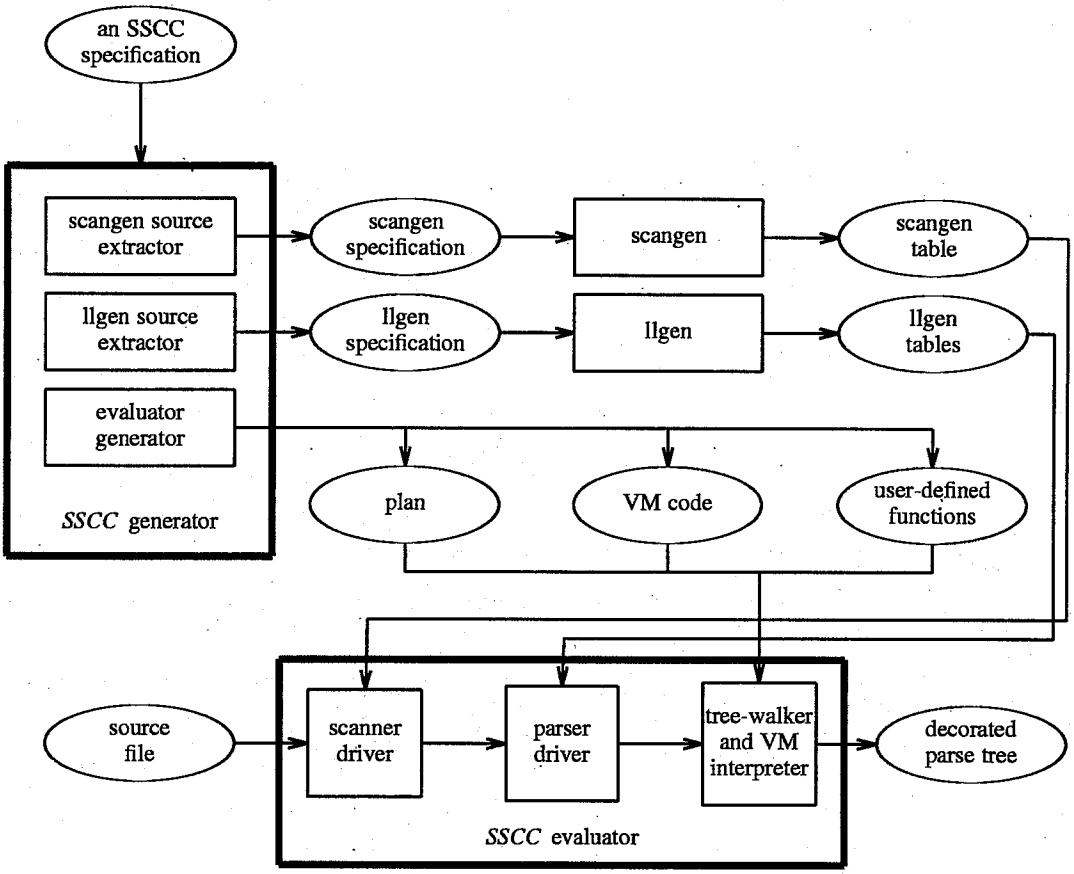

Figure 1. *SSCC* system architecture

evaluator. The generator computes the evaluation order of attribute occurrences in productions and translates the high-level attribution equations into lower-level code for a virtual stack machine. The generator also produces the token definitions and the context-free grammar. The token definitions are translated into a table for a scanner by the *scangen* tool [4]. The context-free grammar is translated into a parse table for a parser by the *llgen* tool [4]. The frontend of the evaluator consists of drivers of a lexical analyzer and a parser that make use of their respective tables. After a program is translated into the corresponding syntax tree, the evaluator traverses the tree based on the evaluation order. During the traversal, the evaluator interprets the virtual-machine code produced by the generator and makes appropriate calls to user-defined functions. The fully decorated tree is passed to a `finalize` function, which may be supplied by the user. This offers the user opportunities for further processing the tree and the attributes.

The rest of the paper is organized as follows. We introduce the definition of attribute grammars in Section 2. The *SSCC* specification language is introduced in Section 3. In Section 4, we define the virtual machine and show the implementation of the *SSCC* system. We conclude this paper in the last section.

2. Definition of an Attribute Grammar

In this section, we define attribute grammars. Basically, we adopt Kastens's notations [7].

An attribute grammar is built from a context-free grammar (N, T, P, S), where N is a finite set of nonterminals, T is a finite set of terminals, S is a distinguished nonterminal, called the *start symbol*, and P is a set of production rules of the form: $X \longrightarrow \alpha$, where X is a nonterminal and α is a string of terminals and/or nonterminals. For each nonterminal X, there is at least one production rule whose left-hand-side symbol is X. As usual, we require that the sets of terminals and nonterminals be disjoint.

Attached to each symbol X of the context-free grammar is a set of *attributes* A_X. Intuitively, instances of attributes in a syntax tree describe the properties of a specific instance of a terminal or a nonterminal. The attributes of a symbol X are partitioned into two disjoint subsets, called the *inherited* attributes (AI_X) and the *synthesized* attri-

butes (AS_X), respectively. There are semantic equations defining these attributes. For a production, there are a semantic equation defining each synthesized attribute of the left-hand-side symbol and a semantic equation defining each inherited attribute of the right-hand symbol of the production. We assume that the start symbol has no inherited attributes. Semantic conditions may also be asserted in the productions in order to express additional constraints on the context-free language. An example is given in Figure 2, which contains three production rules, eight attribution equations, and a semantic condition.

3. The *SSCC* Specification Language

An *SSCC* specification consists of seven sections: `class`, `definition`, `typedef`, `attribute`, `routines`, `rule`, and `routinebody`. The `typedef`, `routines`, and `routinebody` sections are optional. In the following discussion, we refer to the example specification shown in Figure 2. The example defines a context-free grammar for a sequence of `a`'s followed by a single `b`. There is a further constraint that the number of `a`'s must be even (in fact, this constraint can be expressed in the context-free grammar; however, we use this constraint to demonstrate a semantic condition). The attribution equations count the number of `a`'s and change the `a`'s in the even positions into `e`'s and those in the odd positions into `o`'s. The last character, `b`, is changed into `c`.

The `class` section defines character classes with symbolic names. The `definition` section defines tokens of the target language. The two sections use the same format as the one employed in *scangen* [4]. In the example, two kinds of tokens are defined, namely, `LETA` and `LETB`. White spaces will be discarded automatically by the scanner.

The specification language provides five primitive data types: `int`, `real`, `boolean`, `char`, and `string`. However, a user is allowed to define any desirable new types. These new types are listed in the `typedef` section. In the example, a new type `List` is declared. Since only the names of the user-defined types are needed by the `SSCC` system for full static type-checking, these user-defined types are also called *abstract types*. Details of the operations that access and manipulate values of abstract types are hidden in user-defined functions. The separation of attribute dependencies and attribute computations simplifies and clarifies specifications.

The names and types of attributes for symbols are declared in the `attribute` section. Following trends in modern language definitions, all attributes must be declared. In the example, the symbol `S` has two attributes and `X` has four. Note that the *SSCC* system automatically infers whether an attribute is inherited or synthesized. This piece of information is not declared by the user.

In order to write attribution equations, *SSCC* provides the following primitive arithmetic and logic operators: `+`, `-`, `*`, `/`, `^` (the exponentiation operator), `=`, `!=`, `>`, `>=`, `<`, `<=`, `and`, `or`, and `if-then-else`. It also provides type-casting operations for converting between `int` and `real` values. In addition to these primitive operators, a user is free to define new functions either to simplify the specification or to manipulate user-defined types. User-defined functions must be declared in the `routine` section together with the types of the parameters and return values. The *SSCC* system performs complete static type-checking of the attribution equations. In the example, four user-defined functions are declared.

The main part of a specification is the `rule` section, which contains the context-free syntax and the associated attribution equations and semantic conditions. In the *SSCC* system, the context-free syntax must be an LL(1) grammar [1] because *SSCC* makes use of the *llgen* tool [4] to generate parse tables for the frontend of the evaluator (discussed in the next section). The attribution equations are defined directly in terms of the concrete syntax, rather than on a separate, abstract syntax, as is done in many other attribute-grammar systems. This approach saves the overhead of transforming concrete syntax trees into abstract syntax trees. The price we paid, however, is that concrete syntax trees are usually larger (and hence occupies more storage) than abstract syntax trees.

The attribution equations define the computation of attributes. The semantic conditions are constraints on the defined language. For instance, in modern programming languages, it is usually required that a variable must be declared before it is used. This requirement does not compute new attributes; rather it is expressed as a semantic condition. In the example, there is a semantic condition for the first production rule. The condition asserts that the value of the attribute `S.c` must be even.

User-defined types and functions, written in the C language, may be put in the `routinebody` section or may be placed in a separate file. The *SSCC* system, like `yacc`, simply copies this section, together with a few pre-defined data types, into a file for compilation by the C compiler. Since *SSCC* does not analyze the C code, it is not capable of checking the type consistency between the C code and the attribution equations, though the consistency within the attribution equations is fully and statically checked.

4. Implementation of *SSCC*

SSCC is based on a polynomial-time extension to Kastens's ordered attribute grammars. The algorithm for the extended ordered attribute grammars

```
MODULE demo
  CLASS
    LETA = 'a';
    LETB = 'b';
    NEWL = 10;
    BLNK = ' ';
  DEFINITION
    IDA {1}         = LETA;
    IDB {2}         = LETB;
    WhiteSpace {0} = (BLNK{TOSS}, NEWL{TOSS})+;
  TYPEDEF            # TYPEDEF section is optional
    LIST
  ATTRIBUTE
    S { INT c; LIST out;}
    X { INT c; LIST in, out;}
  ROUTINES           # ROUTINES section is optional
    INT     finalize();
    LIST    newlist();
    BOOLEAN even(INT);
    LIST    concat(LIST, CHAR);
  RULE
    P0 = S => X
         COMPUTE
           S.c = X.c;
           X.in = newlist();
           S.out = X.out;
         CONDITION
           even(S.c)
         END;
    P1 = X => IDA X
         COMPUTE
           X[1].c = X[2].c + 1;
           X[2].in = X[1].in;
           X[1].out = IF even(X[1].c)
                      THEN concat(X[2].out, 'e')
                      ELSE concat(X[2].out, 'o') ENDIF;
         END;

    P2 = X => IDB
         COMPUTE
           X.c = 0;
           X.out = concat(X.in, 'c');
         END;
ROUTINEBODY

#include <stdio.h>
#include "finalize.h"

typedef struct LL { char ch; struct LL *next; } LinkList;
static int tmpeven; /* a global variable for the even function */

static void printlist(head)
struct AttribListNode *head;
{ LinkList *h;
  printf("  %s  ", head->type);
  h = *(LinkList **)(head->value);
  for ( ; h; h = h->next) printf("%c", h->ch);
}
```

```
int finalize(root, symtab)
struct TreeNode *root;  struct AttributeNode *symtab;
{ struct AttribListNode *tmp;
  for (tmp = root->AttribList; tmp != NULL; tmp = tmp->next) {
    printf("%s", tmp->name);
    if (strcmp(tmp->type, "INT") == 0)
       printf("  %s  %d", tmp->type, (int)*(int *)(tmp->value));
    else if (strcmp(tmp->type, "LIST") == 0)
       printlist(tmp);
    else printf("  %s is WRONT TYPE!", tmp->type);
  }
  return(0);
}

int *even(v) int *v;
{ if (*v / 2 * 2 == *v) tmpeven = 1; else tmpeven = 0; return(&tmpeven); }

static LinkList *tmpnewlist; /* a global variable for the newlist function */

LinkList **newlist()
{ tmpnewlist = NULL; return(&tmpnewlist); }

LinkList **concat(l, c)
LinkList **l;  char   *c;
{ LinkList *node, *m;
  node = (LinkList *)malloc(sizeof(LinkList));
  node->ch = *c;
  node->next = NULL;
  if (*l == NULL) { *l = node; }
  else {  m = *l;
          while (m->next != NULL) { m = m->next; }
          m->next = node;
  }
  return(l);
}

ENDM demo
```

Figure 2. A complete *SSCC* specification

is similar to that for the original ordered attribute grammar. For the sake of brevity, the reader is referred to Kastens's and Yang et al.'s reports [7, 18] for the algorithms to find evaluation orders. In this section, we will explain the implementation of the *SSCC* system.

SSCC is implemented in the *Unix* environment on Sun Sparc Classic workstations. Since the implementation uses only the standard C language, porting *SSCC* to other platforms should not cause any difficulty. *SSCC* consists of two parts: a generator and an evaluator, shown in Figure 1. The generator is implemented with the help of *lex* [12] and *yacc*. The generator processes *SSCC* specifications and prepares input for *scangen* and *llgen*. The generator also checks the specifications for correctness and produces *plans* for the production rules, virtual-machine code for the attribution equations, a symbol table of symbols and attributes, and two files containing the C code for user-defined functions and a function handler. The results produced by the generator are used in the evaluator.

A scanner driver and a parser driver, together with the tables produced by *scangen* and *llgen*, form the frontend of the evaluator. The backend of the evaluator consists of a tree-walker and a virtual-machine interpreter. The tree-walker walks through the syntax tree according to the plans produced by the generator. The interpreter executes virtual-machine code.

The *SSCC* evaluator is a *table-driven* generic evaluator for extended ordered attribute grammars. In addition to the scanner and the parser, which are driven by tables, the tree-walker and the interpreter are also driven by plans and virtual-machine code, respectively.

The attribution equations in the rule section indicate dependencies among attribute occurrences in a production rule. Since an attribute occurrence cannot be evaluated until all the attribute occurrences it depends on are evaluated, the dependencies among attribute occurrences dictate an evaluation order of the attribute

occurrences. The algorithm for the (extended) ordered attribute grammars computes a correct evaluation order. On the other hand, semantic conditions in a production rule may be evaluated as soon as all the attribute occurrences used in the conditions are evaluated. For the sake of simplicity, evaluation of all semantic conditions are delayed until all attribution equations in the production are evaluated in *SSCC*.

The evaluation order is represented by a plan for each production in the grammar. The plans for the example in Figure 2 is shown in Figure 3. There is one plan for each production in the grammar. A plan consists of a sequence of elements. There are three kinds of elements: *eval X.a*, *visit (i, j)*, and *leave* (a *FINISH* element in Figure 3 is equivalent to a *leave* element). The element *eval X.a* directs the evaluator to evaluate the compiled attribute equation for the attribute *X.a* by invoking the VM interpreter. A *visit (i, j)* element guides the evaluator to descend into the i^{th} child for the j^{th} time to perform the evaluation plan applied at that child. The evaluation in the current plan is suspended until the evaluation in the child's plan encounters a *leave* element. Thus, the evaluator switches among the plans for the nodes in the syntax tree, evaluating the attributes in a prescribed order. Due to the properties of (extended) ordered attribute grammars, it is guaranteed that all the attribute instances in the syntax tree will be evaluated and the evaluator will return to the root once its work is done.

To evaluate an attribute occurrence means to evaluate the right-handside expression of the attribution equation defining the attribute occurrence. The attribution equation is written in a high-level notation. Since there are, in general, many instances of the same attribute occurrence in a syntax tree, it may be necessary to evaluate the same attribution equation many times. In order to improve the efficiency of the evaluator, the equations are translated into assembly-like code for a

```
RULE 0 VISIT  1, 1
       EVAL   S.c
       EVAL   X.in
       VISIT  1, 2
       EVAL   S.out
       FINISH
RULE 1 VISIT  2, 1
       EVAL   X[1].c
       LEAVE
       EVAL   X[2].in
       VISIT  2, 2
       EVAL   X[1].out
       FINISH
RULE 2 EVAL   X.c
       LEAVE
       EVAL   X.out
       FINISH
```

Figure 3. Plans for the three production rules

virtual machine.

4.1. The virtual machine

The virtual machine is a stack machine. There are 24 instructions for this machine, shown in Figure 4. An instruction contains an operator and an operand. The operand might be a constant or an attribute name. An attribute name is a character string.

The `APPLY` instruction is used to implement user-defined functions. An `APPLY k` instruction applies a function to k paramenters; the name of the function and the k parameters are popped from the stack. The `COND` instruction is used to implement semantic conditions. If a `COND` instruction evaluates to *false*, the whole evaluator will halt with an error message indicating the violated semantic condition. The virtual-machine code for the example is shown in Figure 5.

Note that, in the virtual-machine code in Figure 5, attribute names are represented as character strings, rather than addresses as in conventional compilers. Therefore, the interpreter kernel needs a symbol table, which is also produced by the generator.

The interpreter is a combination of a generic, precompiled kernel and separately compiled and linked user-defined functions and a function handler. The generic kernel interprets virtual-machine code and calls appropriate user-defined functions when necessary. The link between user-defined functions and their implementation in C code is established by the names of functions. This is discussed in the next subsection.

4.2. User-defined functions

User-defined functions are invoked by the generic kernel of the interpreter based on the names of functions. A user-defined function such as `concat(1, c)` is implemented as follows. The function call is translated into three consecutive `PUSH`es that push the function name `concat` and the values of the parameters `1` and `c` into the stack. These are followed by an `APPLY 2` instruction, which pops three elements from the stack (two examples are shown in the virtual-machine code for the second production in Figure 5). The name of the function, as a character string, is passed to the function handler that calls a compiled function with the same name. The function handler is essentially a nested *if* statement with one branch as follows:

```
else if (strcmp(Funcname,
                "concat") == 0)
 return (void *)concat(para[0],
                       para[1]);
```

The function handler is a link between symbolic function names known in the specification and the compiled functions that are supplied by the user. Since it needs to know the names of the functions, the function handler must be synthesized from an

```
arithmetic operations
  ADD, SUB, MUL, DIV, EXP:      perform the indicated operation on the top
                                two elements in the stack

logical operations
  AND, OR, NOT:                 perform the indicated operation on the top
                                two elements in the stack

comparison operations
  EQU, NONEQU, GT, GE, LT, LE:  perform the indicated operation on the top
                                two elements in the stack

condition branches
  COMP:                         branch to the else-label if the top element
                                is false
  ELSE:                         indicate a position for the COMP instruction
  ENDIF:                        indicate the end of an if expression
function application
  APPLY k:                      pops k cells and a function name from the stack
                                and call the function with k cells as parameters

miscellanea
  PUSH arg:                     push arg into the stack
  ASSIGN:                       assign the top element on the stack to the
                                element just below it
  COND:                         check whether the top element is false
  WIDEN, SHORTEN:               convert between int and real values
  MINUS:                        negate a value
```

Figure 4. Instruction set of the virtual machine

SSCC specification. The function handler, like user-defined functions, is not part of the generic kernel of the interpreter; it must be compiled and linked with the precompiled kernel of the interpreter.

Parameters to user-defined functions must be in a generic form in order to bypass the type-checking of the C compiler when compiling the interpreter. In *SSCC*, parameters are passed as pointers. The `concat` function in Figure 2 is such an example.

4.3. The `finalize` function

After all the attribute instances in a syntax tree are evaluated, the default action of the evaluator is to print the syntax tree and the attributes. However, *SSCC* provides a hook—the `finalize` function—so that a user can specify additional post-processing on the decorated tree by providing his own version of `finalize`. The `finalize` function accepts a pointer to the root of the syntax tree and a pointer to the symbol table; all nodes and attribute instances are accessible through the two pointers. Since the user may needs to access the internal data structures of the evaluator, the required data type definitions are collected in the *finalize.h* file.

5. Conclusion

We have implemented a practical attribute-grammar system *SSCC* based on an extension of ordered attribute grammars. Due to the inclusion of the user-defined types and functions, *SSCC* is capable of processing any desirable attribute computation. Furthermore, a user can process the fully decorated syntax trees by supplying his own version of the `finalize` function.

Although *yacc*-like parser generators can also implement systems that are implemented by attribute-grammar systems, attribute-grammar systems are a more declarative tool in that a user need not worry about the evaluation order for obtaining a correct result. By contrast, *yacc*-like tools require a user to explicit write pieces of code that are executed during parsing.

There are many existing attribute-grammar systems [5, 8, 10]. These systems implement editors[15], compilers[3, 16, 17]. compiler-compilers, programming environments, and proof checkers [14]. *SSCC* has been used as the frontend of a parallel-evaluation system for attribute grammars in the PVM environment [11].

There are a few improvements that we wish to implement. First we wish to decouple the abstract syntax from the concrete syntax. Currently, the attribute equations are closely bound to the concrete syntax. This approach requires the whole parse tree be maintained in the evaluator.

REFERENCES

1. A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, Reading, MA (1986).
2. G.V. Bochmann, Semantic evaluation from left to right, *Comm. ACM* 19(2) pp. 55-62 (February 1976).
3. R. Farrow, Generating a production compiler from an attribute grammar, *IEEE Software* 1(4) pp.

```
RULE 0 PUSH   S.c
       PUSH   X.c
       ASSIGN
       PUSH   X.in
       PUSH   newlist
       APPLY  0
       ASSIGN
       PUSH   S.out
       PUSH   X.out
       ASSIGN
       PUSH   even
       PUSH   S.c
       APPLY  1
       COND
RULE 1 PUSH   X[1].c
       PUSH   X[2].c
       PUSH   1
       ADD
       ASSIGN
       PUSH   X[2].in
       PUSH   X[1].in
       ASSIGN
       PUSH   X[1].out
       PUSH   even
       PUSH   X[1].c
       APPLY  1
       COMP
       PUSH   concat
       PUSH   X[2].out
       PUSH   'e'
       APPLY  2
       ELSE
       PUSH   concat
       PUSH   X[2].out
       PUSH   'o'
       APPLY  2
       ENDIF
       ASSIGN
RULE 2 PUSH   X.c
       PUSH   0
       ASSIGN
       PUSH   X.out
       PUSH   concat
       PUSH   X.in
       PUSH   'c'
       APPLY  2
       ASSIGN
```

Figure 5. Virtual-machine code for the three production rules

77-93 (July 1984).

4. C.N. Fischer and R.J. LeBlanc, Jr., *Crafting a Compiler with C,* Benjamin/Cummings, Reading, MA (1991).
5. R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane, and W.M. Waite, Eli: A complete, flexible compiler construction system, *Comm. ACM* **35**(2) pp. 121-131 (February 1992).
6. S.C. Johnson, YACC-Yet another compiler compiler, CSTR 35, Bell Labs., Murray Hill, N.J. (1979).
7. U. Kastens, Ordered attribute grammars, *Acta Informatica* **13** pp. 229-156 (1980).
8. U. Kastens, B. Hutt, and E. Zimmermann, *GAG: A Practical Compiler Generator,* Springer-Verlag, New York (1982).
9. D.E. Knuth, Semantics of context-free languages, *Mathematical System Theory* **2**(2) pp. 127-145 (June 1968). Correction. *ibid. 5*, 1 (March 1971), 95-96.
10. K. Koskimies and J. Paakki, *Automating Language Implementation,* Ellis Horwood, New York (1990).
11. S.-H. Lee, Y.-T. Liu, J.-T. Chan, and W. Yang, Automatic generation of parallel compilers in the PVM environment, pp. 51-57 in *Proceedings of the 2nd Workshop on Compiler Techniques for High-Performance Computing,* (Taiwan, R.O.C., March 20-22, 1996), (January 1996).
12. M.E. Lesk and E. Schmidt, LEX — A lexical analyzer generator, Computer Science Technical Report 39, Bell Labs., Murray Hill, N.J. (1975).
13. J. Paakki, Attribute grammar paradigms—A high-level methodology in language implementation, *ACM Computing Surveys* **27**(2) pp. 196-255 (June 1995).
14. T. Reps and B. Alpern, Interactive proof checking, pp. 36-45 in *Conference Record of the Eleventh ACM Symposium on Principles of Programming Languages,* (Salt Lake City, UT, Jan. 15-18, 1984), ACM, New York (1984).
15. T.W. Reps, *Generating language-based environments,* MIT Press, Cambridge, MA (1984).
16. W.M. Waite, Use of attribute grammars in compiler construction, *Workshop on Attribute Grammars and Their Applications, Lecture Notes in Computer Science* **461** pp. 255-265 (1990).
17. W.M. Waite, A complete specification of a simple compiler, CU-CS-638-93, Computer Science Dept., Univ. of Colorado at Boulder, Boulder, CO (January 1993).
18. W. Yang and W.-C. Cheng, A polynomial-time extension to ordered attribute grammars, submitted for publication, Computer and Information Science Dept., National Chiao-Tung Univ., Hsinchu, Taiwan (June 1996).