An Attribute Assignment View of Non-procedural Computing Systems

TAKEHIRO TOKUDA*

This paper gives an attribute assignment view of the interpretation and evaluation of non-procedural computing systems. We first show that a number of non-procedural computing systems such as Prolog, Wijngaarden grammars, and attribute grammars can be viewed as attribute assignment systems in spite of their different original motivations. An attribute assignment view is a natural common abstraction of these non-procedural computing systems.

An attribute assignment system consists of a context-free grammar, a set of attributes for nonterminals, and a set of relations on attributes for productions. Attribute assignment systems allow us to describe both specifications and solutions of the problems relatively easily.

We then show that our attribute assignment view enables us to have a new evaluation method in a restricted case where every attribute has finite domains. This evaluation method is based on set-theoretic operations. This evaluation method has a feature that the performance is fairly stable even in worst cases.

1. Introduction

Traditionally non-procedural computing systems are usually envisioned through various formal systems such as logic systems, function systems and rewriting systems [5, 8–10, 12–14, 16, 23]. In this paper we present a new view of the interpretation and evaluation of non-procedural computing systems.

We first show that a number of existing non-procedural computing systems can be viewed as attribute assignment systems in spite of different original motivations. An attribute assignment system is a natural common abstraction of these non-procedural computing systems. We then show that our attribute assignment view enables us to have a new evaluation method in a restricted case where every attribute has finite domains. This evaluation method has a feature that the performance is fairly stable even in worst cases.

An attribute assignment system consists of a context-free grammar, a set of attributes for each nonterminal, and a set of relations for each production. Attribute assignment systems allow us to describe the specification and the solution of a problem relatively easily.

The organization of the rest of this paper is as follows. In chapter 2, we quickly illustrate the main results of this paper. In chapter 3, we briefly review preliminary concepts in context-free grammars and some non-procedural computing systems. In chapter 4, we formally define attribute assignment systems and show that some of existing non-procedural computing systems can be naturally viewed as attribute assignment systems. In chapter 5, we present an evaluation algorithm for attribute assignment systems with finite domains. In chapter 6, we discuss implications of attribute assignment systems, an experimental comparison with backtracking method, and possible generalizations of evaluation methods. In chapter 7, we give concluding remarks.

*Department of Computer Science, Yamanashi University, Takeda, Kofu 400, Japan. (Present Address: Department of Computer Science, Tokyo Inst. of Tech., Meguro, Tokyo 152, Japan)

2. Overview

In this chapter we give a quick illustration of the motivation and the main results of this paper using examples. We focus on three non-procedural computing systems: Prolog, Wijngaarden grammars and attribute grammars. Historically these non-procedural computing systems have quite different motivations and backgrounds. However there exist great structural similarities among them. Namely these systems represent structures of computation on certain types of trees (computation trees, derivation trees, and parse trees, respectively). The shape of this tree may be fixed (static) or variable (dynamic). These systems have either the distinction of trees and attributes or no distinction as shown in Figure 1.

Main results of this paper are as follows.

(1) Essentially, as a definition language, Prolog in

	Distinction of Trees and Attributes	Shape of Trees
Prolog	yes	variable
W-grammars	No Attributes	variable
A grammars	yes	fixed

Fig. 1 Comparison of three non-procedural computing systems.

the 70's is a rediscovery of what ALGOL 68 people once discovered in the power of Wijngaarden grammars in the 60's.

(2) Attribute grammars, as a definition language, are restricted forms of Prolog and Wijngaarden grammars. Hence definitions in attribute grammars can be naturally translated into definitions in Prolog or Wijngaarden grammars.

(3) As a natural abstraction of common characteristics of Prolog, Wijngaarden grammars and attribute grammars, we may define a class of non-procedural computing systems called attribute assignment systems.

(4) A general strategy of evaluating attribute assignment systems is backtracking as in Prolog. We may, however, have an efficient evaluation method in restricted cases taking advantage of the fact that underlying trees are fixed.

Now let us think of a classic example of the definition of the factorial function in terms of Prolog in Example 1. This definition consists of two relations.

Example 1.

factorial (1, 1).
factorial (I, J):—$I1$ is $I-1$, factorial ($I1$, $J1$), J is $I*J1$.

If we give the definition of the factorial in terms of Wijngaarden grammars, then the definition will look the same as in Example 2. (A formal definition of Wijngaarden grammars will be given later in the section 3.2.2) Note that there is a great similarity of two lines of Example 1 and hyperrules (2) and (3) of Example 2. In Wijngaarden grammars, everything is represented in terms of syntax. For example, "one one one one one is the factorial of one one one" is a symbol representing a nonterminal and this nonterminal derives the empty string.

Example 2.

⟨Metaproductions⟩
(1) VAL :: VAL one; one.

⟨Hyperrules⟩
(1) start: VAL1 is the factorial of VAL2.
(2) one is the factorial of one:.
(3) VAL1 is the factorial of one VAL: VAL2 is the factorial of VAL, VAL1 is the product of one VAL and VAL2.
(4) VAL is the product of one and VAL:.
(5) VAL VAL1 is the product of one VAL2 and VAL: VAL1 is the product of VAL2 and VAL.

If we give a definition of the factorial function in terms of attribute grammars, the result will be as in Example 3. Here we have a distinction between syntax and attributes unlike Wijngaarden grammars. The domain of an attribute value is integers.

Example 3.

⟨Syntax Rule⟩
(1) $S \rightarrow N$
(2) $N_1 \rightarrow N_2\ i$
(3) $N \rightarrow i$

⟨Semantio Functions⟩
(1) S. value:=N. value
(2) N_1. value:=N_2. value*N_1. number
    N_1. number:=N_2. number+1
(3) N. value:=1
    N. number:=1

In order to take another look at the distinction of syntax and attributes, we take examples of defining a parsing method. We consider a problem of parsing an input "bbabaa" according to the syntax rule of Example 6. (This method is an extremely simplified version of Earley's parsing method. Here we use a nonterminal (or terminal) symbol instead of a production rule with a dot.) In this solution by Prolog in Example 4, two integers are used for specifying the start locations and finish locations of a substring of the input.

Example 4.

```
uppers(I, J):—I1 is I+1, lowera(I, I1), upperb(I1, J).
uppers(I, J):—I1 is I+1, lowerb(I, I1), upperа(I1, J).
uppera(I, J):—J is I+1, lowera(I, J).
uppera(I, J):—I1 is I+1, lowera(I, I1), upperѕ(I1, J).
uppera(I, J):—I1 is I+1, lowerb(I, I1), upperа(I1, I2), uppera(I2, J).
upperb(I, J):—J is I+1, lowerb(I, J).
upperb(I, J):—I1 is I+1, lowerb(I, I1), upperѕ(I1, J).
upperb(i, J):—I1 is I+1, lowera(I, I1), upperb(I1, I2), upperb(I2, J).

lowerb(1, 2).
lowerb(2, 3).
lowera(3, 4).
lowerb(4, 5).
lowera(5, 6).
lowera(6, 7).
```

In a solution by Wijngaarden grammars in Example 5, integers are represented by unary notation and this forms a part of a long nonterminal name.

Example 5.

⟨Metaproductions⟩
(1) VAL :: VAL one; one.

⟨Hyperrules⟩
start: uppers with VAL1 and VAL2.
uppers with VAL and VAL1: lowera with VAL and one VAL, upperb with one VAL and VAL1.
uppers with VAL and VAL1: lowerb with VAL and one VAL, uppera with one VAL and VAL1.
uppera with VAL and one VAL: lowera with VAL and one VAL.
uppera with VAL and VAL1: lowera with VAL and

one VAL, uppers with one VAL and VAL1.
uppera with VAL and VAl2: lowerb with VAL and one VAL, uppera with one VAL and VAL1, uppera with VAL1 and VAL2.
upperb with VAL and one VAL: lowerb with VAL and one VAL.
upperb with VAL and VAL1: lowerb with VAL and one VAL, uppers with one VAL and VAL1.
upperb with VAL and VAL2: lowera with VAL and one VAL, upperb with one VAL and VAL1, upperb with VAL1 and VAL2.
lowerb with one and one one:.
lowerb with one one and one one one:.
lowera with one one one and one one one one:.
lowerb with one one one one and one one one one one:.
lowera with one one one one one and one one one one one:.
lowera with one one one one one one and one one one one one one:.

A solution by attribute grammars is shown in Example 6, The domains of attributes start and finish are integers. Handling of construction of the derivation tree is done by a syntax analyzer.

Example 6.
⟨Syntax Rule⟩
(1) $S \rightarrow aB$
(2) $S \rightarrow bA$
(3) $A \rightarrow a$
(4) $A \rightarrow aS$
(5) $A_1 \rightarrow bA_2A_3$
(6) $B \rightarrow b$
(7) $B \rightarrow bS$
(8) $B_1 \rightarrow aB_2B_3$
⟨Semantic Functions⟩
(1) B. start:=2
(2) A. start:=2
(3) A. finish:=A. start+1
(4) S. start:=A. start+1
A. finish:=S. finish
(5) A_2. start:=A_1. start+1
A_3. start:=A_2. finish
A_1. finish:=A_3. finish
(6) B. finish:=B. start+1
(7) S. start:=B. start+1
B. finish:=S. finish
(8) B_2. start:=B_1. start+1
B_3. start:=B_2. finish
B_1. finish:=B_3. finish

Note that the unification in Prolog is corresponding to the uniform replacement in Wijngaarden grammars and the identity assignment in attribute grammars.

3. Preliminaries

3.1 Context-Free Grammars

We review concepts of context-free grammars. A context-free grammar $G=(N, T, P, S)$ is a rewriting system which consists of four components: a set N of nonterminals; a set T of terminals which is disjoint from N; a set P of productions of the form $X(p, 0) \rightarrow X(p, 1), X(p, 2) \ldots, X(p, n(p))$ where $X(p, 0)$ is a nonterminal and $X(p, 1), X(p, 2) \ldots, X(p, n(p))$ is a string of nonterminals and terminals of length $n(p) \geqq 0$; and the start nonterminal S, which appears only in the left-hand side of the 0-th production of P.

Rewriting is started from the start nonterminal S, and repeated until there are no nonterminals in the string. One-step rewriting is done by replacing one nonterminal in the string by its corresponding right-hand side string of a production.

As a rewriting system context-free grammars have limited capabilities. Namely context-free grammars can define a proper subclass of recursive languages. But the condition that the left-hand side of a production consists of one nonterminal makes us treat rewriting process in easy and context independent manner. This feature will naturally correspond to, for example, Prolog system as we shall discuss later.

3.2 Non-Procedural Computing Systems

We briefly summarize some of the non-procedural computing systems, Prolog, Wijngaarden grammars, and attribute grammars.

3.2.1 Prolog

Prolog is a programming language based on the concept of Horn clause [14]. A Horn clause is a clause which is either denial. assertion, or implication. A Prolog program is a sequence of Horn clauses. Each Horn clause consists of head and body. We refer to a clause without its body as an assertion. We refer to a clause without its head as a denial or a goal. The head has zero or one predicate. The body has zero or more predicates.

A predicate is an n-tuple of terms prefixed by a predicate symbol. A term is either a variable, a constant, or a compound term. A compound term is an n-tuple of terms prefixed by a functor symbol. Prolog programs can specify any recursively enumerable sets. (As for concepts of recursively enumerable sets and predicates, we refer to [11, 15, 17], for example.) (Representability of recursively enumerable sets by Prolog programs is rather straightforward [6, 18].)

We give a definition of well-balanced parentheses consisting of "{" and "}" in terms of Prolog in Example 7. A formulation given in this example is rather redundant because of pedagogical reasons. Infix operator "|" stands for cons operation of car and cdr parts of a list.

Example 7.

```
wellbalanced (PAR):—binary (PAR), balanced (PAR).
balanced ( [ ] ).
balanced (PAR):—append ([ { | PAR1 ], [ } | PAR2], PAR), balanced (PAR1), balanced (PAR2).
binary ( [ ] ).
binary ( [ { | PAR] ):—binary (PAR).
binary ( [ } | PAR] ):—binary (PAR).
append ( [ ], LIS, LIS).
append ( [HEAD | REST0], LIS, [HEAD | REST1] ):—append (REST0, LIS, REST1).
```

3.2.2 Wijngaarden grammars

Wijngaarden grammars are systems for writing specifications [9]. Wijngaarden grammars can specify any recursively enumerable sets, usually in a concise manner. (Representability of recursively enumerable sets by Wijngaarden grammars is rather straightforward [19]. A simple proof would be to construct a Wijngaarden grammar to simulate a type 0 rewriting system [21].) A Wijngaarden grammar $W=(M, H)$ consists of a context-free grammar M called metaproductions, and another context-free grammar H called hyperrules.

Nonterminals of metaproductions are called metanotions and terminals of metaproductions are called protonotions. Metanotions are denoted by strings of uppercase letters. Protonotions are denoted by strings of lowercase letters or special characters. (Hence "{ } is equal to { }" and "is balanced" are examples of a single symbol of protonotions. We insert spaces within a single symbol to improve readability in hyperrules.) We derive possibly infinite protonotions from each metanotion using metaproductions. Grammar symbols of hyperrules are protonotions or metanotions or their composite. Terminals of hyperrules are protonotions ending with reserved word "symbol", and nonterminals of hyperrules are other protonotions.

Hyperrules are mold rules for final context-free grammars. By uniformly replacing each occurrence of metanotions in the hyperrules by its derived protonotions, we generate possibly infinite number of context-free productions from hyper-rules. We give a definition of well-balanced parentheses in terms of a Wijngaarden grammar in Example 8. In this case final context-free grammars can only generate the empty string.

Example 8.

```
⟨Metaproductions⟩
(1) PAR :: { PAR;
           } PAR;
           EMPTY.
(2) EMPTY ::.
⟨Hyperrules⟩
(1) start: PAR is balanced.
(2) PAR is balanced: PAR is equal to { PAR1 } PAR2, PAR1 is balanced, PAR2 is balanced.
(3) EMPTY is balanced: true.
(4) PAR is equal to PAR: true.
(5) true: EMPTY.
```

Note that following examples are productions derived from hyperrules using uniform replacement.

```
start: is balanced.
start: { } is balanced.
is balanced: is equal to { }, is balanced, is balanced.
{ } is balanced: { } is equal to { }, is balanced, is balanced.
{{ } is balanced: {{ } is equal to {{ }, { is balanced, is balanced.
is balanced: true.
is equal to: true.
{ is equal to {: true.
{ } is equal to { }: true.
true:.
```

Using these productions, for example, we can derive an empty string from the nonterminal "{ } is balanced".

```
start ⇒ { } is balanced
      ⇒ { } is equal to { }, is balanced, is balanced
      ⇒ true, is balanced, is balanced
      ⇒ is balanced, is balanced
      ⇒ true, is balanced ⇒ is balanced ⇒ true ⇒ .
```

Hence we know that "{ }" is balanced.

3.2.3 Attribute grammars

Attribute grammars are systems for describing computing systems [12]. An attribute grammar $A=(G, D, F)$ consists of a context-free grammar G, a family D of sets $A(X)$ of attributes for each nonterminal X, and a set $F(p)$ of semantic functions for each production p.

The set $A(X)$ of attributes for a nonterminal X is partitioned into synthesized attributes and inherited attributes. Synthesized attributes are attributes of a nonterminal when it is used as a father, i.e. the left-hand side of a production. Inherited attributes are attributes of a nonterminal when it is used as a son, i.e. a symbol of the right-hand side of a production. Disjointness of synthesized attributes and inherited attributes prevents from the situation in which the value of one attribute is dependent on more than one production.

Each semantic function defines a mapping from values of certain attributes of nonterminals in a production to the value of some attribute in the production.

Attribute grammars can specify any recursively enumerable sets provided that we may use partial recursive functions as semantic functions.

Proof is constructed in such a way that we show a type 0 language can be partially recognized by an attribute grammar system. A grammar gives a Kleene star generation of the underlying alphabet. For any given string we construct a parse tree based on that grammar. Synthesized attribute passes the given string to the root of the parse tree. At the root, we simulate all the derivations of an original type 0 grammar according to the

number of steps of derivations, to see whether or not the given string is in the set. The computation at the root will terminate and report yes, if it is eventually in the set. The computation at the root will not terminate, if it is not in the set.

We give a definition of well-balanced parentheses in terms of an attribute grammar in Example 9. For the nonterminal L we use L_1 and L_2 respectively to distinguish the occurrences in the left-hand side and the right-hand side of the production.

Example 9.

⟨Syntax Rule⟩

(1) $N \rightarrow L$
(2) $L_1 \rightarrow L_2 B$
(3) $L \rightarrow B$
(4) $B \rightarrow \{$
(5) $B \rightarrow \}$

⟨Semantic Functions⟩

(1) N. boole:$=L$. boole and (L. value$=0$)
(2) L_1. value:$=L_2$. value$+B$. value
    L_1. boole:$=L_2$. boole and (L_1. value$\geqq 0$)
(3) L. value:$=B$. value, L. boole:$=(L$. value$\geqq 0)$
(4) B. value:$=1$
(5) B. value:$=-1$

4. Attribute Assignment Approach

4.1 Attribute Assignmet Systems

We present a formal definition of attribute assignment systems. An attribute assignment system consists of a context-free grammar $G=(N, T, P, S)$, a set of attributes $A(X)$ for each nonterminal X in N, and a set of relations $R(p)$ for each production p in P. Let the p-th production of G be as follows.

$$X(p, 0) \rightarrow X(p, 1) \ldots X(p, n(p)),$$

Let an attribute attr be an element of $A(X(p, j))$. Let $V(X(p, j)$, attr) be the domain of the attribute attr associated with nonterminal occurrence $X(p, j)$. Namely $V(X(p, j)$, attr) is the set of possible values for the attribute attr. A semantic relation $R(X(p, j)$, attr) in $R(p)$ is defined for the p-th production, as follows.

$$R(X(p, j), \text{attr}) \text{ is a subset of}$$
$$V(X_0, \text{attr}_0) * V(X_1, \text{attr}_1) * \ldots * V(X_t, \text{attr}_t),$$

where attr_0, attr_1, . . . , and attr_t are attributes of nonterminals X_0, X_1, . . . , X_t of the p-th production, and one of (X_0, attr_0), (X_1, attr_1), . . . , and (X_t, attr_t) is identical with $(X(p, j)$, attr).

Because we do not partition the attributes into inherited and synthesized attributes as in attribute grammars, the value of an attribute may depend on two productions directly, and the entire tree indirectly.

We say that an attribute assignment system is strongly defined, if there exists one and only one global assignment to the attributes of any parse tree of the context-free grammar. (Global assignment is an assignment of attribute values to each node of the derivation tree.) Hence if there are zero or more than one global assignment to some parse tree, then the attribute assignment system is not strongly defined.

We say that an attribute assignment system is weakly defined, if there are one or more global assignment to any parse tree. If the assignment system is strongly defined, then the system is weakly defined.

A-assignment	Prolog
nonterminal symbol	predicate symbol
terminal symbol	empty string
attribute	position of terms
value	constant
relations	clause
definition of domains	predefined domains
unconditional relation	assertion

Fig. 2 Correspondence with Prolog.

4.2 New Interpretations

In this section we show that a number of non-procedural computing systems can be interpretated as attribute assignment systems, and there exist natural correspondences among them.

4.2.1 Prolog

A new interpretation of Prolog is as follows. A Horn clause is now considered as a context-free production with attributes. These context-free productions derive only the empty string. A predicate is viewed as a nonterminal, and arguments associated with the predicate are regarded as attributes. The condition of Horn clause that a head has at most one predicate is essential to this view. As for the left-hand side of the goal, this can be regarded as a hypothetical start nonterminal, which is invisible. Hence, in Prolog, we make an effort to build a tree whose root is an invisible symbol and leaves are all empty strings.

4.2.2 Wijngaarden grammars

A new interpretation of Wijngaarden grammars is as follows. Now a Wijngaarden grammar is not simply an infinite set of context-free productions. We can respectively regard hyperrules and metaproductions as productions with attributes, and definition of attribute domains. Hence we have a finite set of productions with attributes, whose domains are defined by metaproductions. The domain of those attributes are possibly in-

A-assignment	W-grammars
nonterminal symbol	protonotion in LHS of hyperrules
terminal symbol	protonotion ending with "symbol"
attribute	metanotion in hyperrules
value	protonotion in metaproductions
relations	hyperrule
definition of domains	metaproduction
unconditional relation	hyperrule without metanotions

Fig. 3 Correspondence with Wijngaarden grammars.

finite. (Attributes having finite domains may be treated either as a part of productions or as attributes themselves.) Without loss of generality we can restrict the type of Wijngaarden grammers to grammars deriving only the empty string.

4.2.3 Attribute grammars

A new interpretation of Attribute grammars is as follows. An attribute grammar is a restricted form of the attribute assignment system. A function is a special form of a relation, where we can determine the value of one attribute from values of the rest of attributes explicitly. In attribute grammars, value dependency is explicitly defined in terms of semantic functions. If an attribute grammar is well-defined, then the attribute grammar is a strongly defined attribute assignment system.

4.2.4 Miscellaneous Discussions

From a theoretical point of view, we have a very interesting problem when we deal with negation. In Wijngaarden grammars, negation of a recursively enumerable predicate is not necessarily recursively enumerable. (Note that a set S is recursive, if and only if S and the complement of S are both recursively enumerable.) Hence negation cannot be dealt with directly by Wijngaarden grammars in general. The construction of the negation of a predicate is not an easy job as in the example 10. A predicate "ALPHA1 is not equal to ALPHA2" stating two characters are different takes a number of lines, while its negation "ALPHA is equal to ALPHA" takes only one line.

Example 10.
⟨Metaproductions⟩
(1) ALPHA :: *a*; *b*; *c*; *d*; *e*; *f*; *g*; *h*; *i*; *j*; *k*; *l*; *m*; *n*; *o*; *p*; *q*; *r*; *s*; *t*; *u*; *v*; *w*; *x*; *y*; *z*.
(2) STRING :: ALPHA STRING;.
⟨Hyperrules⟩
(1) ALPHA is equal to ALPHA:.

A-assignment	Attribute grammar
nonterminal symbol	nonterminal symbol
terminal symbol	terminal symbol
attribute	attribute
value	value
relations	functions
definition of domains	definition of domains
unconditional relation	constant function

Fig. 4 Correspondence with attribute grammars.

(2) ALPHA1 is not equal to ALPHA2:
(STRING1 ALPHA1 STRING2 ALPHA2 STRING3) is identical with (abcdefghijklmnopqrstuvwxyz);
(STRING1 ALPHA2 STRING2 ALPHA1 STRING3) is identical with (abcdefghijklmnopqrstuvwxyz).
(3) (STRING) is identical with (STRING):.

Because of the same reason, we cannot deal with negation directly in Prolog or attribute grammars. (Negation problems in Prolog has been discussed widely concerning negation-as-failure and closed-world assumption [6].)

It is possible to transform Wijngaarden grammars, which generates only the empty string, into Prolog programs using a natural correspondence given in Figure 2 and 3. Transformation of attribute grammars into Wijngaarden grammars can be achieved naturally, Transformation of Wijngaarden grammars into attribute grammars can be achieved artificially, but cannot be achieved naturally, because it involves discovery of a natural decomposition of the whole computation explicitly.

4.3 Examples

In this section we show examples of attribute assignment systems. First we give definitions of the factorial function and the parsing method in terms of attribute assignment systems in Example 11 and 12 respectively. Note that symbol "=" represents equality of left-hand side and right-hand side rather than ordinary assignment. Domains of attributes number, value, start and finish are integers.

Example 11.
⟨Syntax Rule⟩
(1) $S \rightarrow N$
(2) $N_1 \rightarrow N_2 i$
(3) $N \rightarrow i$
⟨Semantic Relation⟩
(1) S value$=N$. value

(2) N_1. value = N_2. value * N_1. number
N_1. number = N_2. number + 1
(3) N. value = 1
N. number = 1

Example 12.
⟨Syntax Rule⟩
(1) $S \rightarrow aB$
(2) $S \rightarrow bA$
(3) $A \rightarrow a$
(4) $A \rightarrow aS$
(5) $A_1 \rightarrow bA_2A_3$
(6) $B \rightarrow b$
(7) $B \rightarrow bS$
(8) $B_1 \rightarrow aB_2B_3$
⟨Semantic Relation⟩
(1) B. start = 2
(2) A. start = 2
(3) A. finish = A. start + 1
(4) S. start = A. start + 1
A. finish = S. finish
(5) A_2. start = A_1 start + 1
A_3. start = A_2. finish
A_1. finish = A_3. finish
(6) B. finish = B. start + 1
(7) S. start = B. start + 1
B. finish = S. finish
(8) B_2. start = B_1. start + 1
B_3. start = B_2. finish
B_1. finish = B_3. finish

The next example is slightly more complex. Now we consider a problem of translating a logical expression into quadruples of short circuit evaluation form. Namely, for example, we translate a logical expression below

(A or B) and (C or D)

into the corresponding quadruples as follows. A quadruple (:=, val, , Z) stands for the assignment of a val to a variable Z. Values 1 and 0 respectively stand for true and false. A quadruple (Br, VAR, i, j) stands for branch to address i, if VAR is true, and branch to address j, if VAR is false.

1	:=1 Z
2	Br A 4 3
3	Br B 4 6
4	Br C 7 5
5	Br D 7 6
6	:=0 Z
7	

A specification (and a solution) of this problem in terms of an attribute assignment system is given in Fig. 5. The domain of attribute code is strings. The domains of attributes next, start, true, and false are integers. The attribute code holds the corresponding code. The attribute start and next hold the starting address of the code and the last address of the code + 1. The attribute true and false hold the goto address of the component

0)	$Z \rightarrow E$	Z. code = (:=, 1, , Z) + E. code + (:=, 0, , Z) E. start = 2 E. true = E. next + 1 E. false = E. next
1)	$E \rightarrow T$	E. code = T. code E. next = T. next E. start = T. start E. true = T. true E. false = T. false
2)	$E_1 \rightarrow$ TOR E_2	E_1. code = TOR. code + E_2. code E_1. next = E_2. next E_1. start = TOR. start E_2. start = TOR. next E_1. true = TOR. true = E_2. true E_1. false = E_2. false TOR. false = TOR. next
3)	TOR $\rightarrow T$ or	TOR. code = T. code TOR. next = T. next TOR. start = T. start TOR. true = T. true TOR. false = T. false = T. next
4)	$T \rightarrow F$	T. code = F. code T. next = F. next T. start = F. start T. true = F. true T. false = F. false
5)	$T_1 \rightarrow$ FAND T_2	T_1. code = FAND. code + T_2. code T_1. next = T_2. next T_1. start = FAND. start T_2. start = FAND. next T_1. true = T_2. true T_1. false = FAND. false = T_2. false FAND. true = FAND. next
6)	FAND $\rightarrow$ F and	FAND. code = F. code FAND. next = F. next FAND. start = F. start FAND. true = F. true = F. next FAND. false = F. false
7)	$F \rightarrow i$	F. code = (Br, lex ( ), F. true, F. false) F. next = F. start + 1
8)	$F \rightarrow (E)$	F. code = E. code F. next = E. next F. start = E.start F. true = E. true F. false = E. false
9)	$F_1 \rightarrow$ not F_2	F_1. code = SWAP(F_2. code) F_1. true = F_2. false F_1. false = F_2. true F_1. next = F_2. next F_1. start = F_2. start

Fig. 5 An attribute assignment description.

constructs in case true and false respectively. The function SWAP gives a new code by swapping the occurrences of F_2. true and F_2. false in a given F_2. code.

5. Evaluation Algorithms

5.1 Algorithms.

Attribute assignment systems allow us to describe the specification of a problem in a concise manner without knowing the solution explicitly. However there does not exist efficient evaluation method in general, and is usually solved by enumeration based on backtracking. In

this section we show that we have a new evaluation method for attribute assignment systems which have finite attribute domains. (If we consider the same problem on an undirected graph, this problem becomes an NP-complete problem provided that the size of a problem is the number of nodes. Namely the k-colorability problem of an undirected graph can be coded as follows. For a node X_0 and its neighbors $X_1, X_2, \ldots, X_m$, the relation is $(i_0, i_1, \ldots, i_m)$ where $i_0, i_1, \ldots, i_m$ are integers from 1 to k and $i_1, \ldots, i_m$ are not equal to i_0.)

We describe a method for a global assignment of attributes to a parse tree in case we need to obtain one assignment of an attribute assignment system with finite attribute domains. Traditional solutions of this type of problem are based on backtracking method. Performance of backtracking method depends on input instances. Our method is based on set-theoretic operations. Intuitively this method consists of two phases. We perform intersections of relations in the first phase and broadcast a solution to each node in the second phase.

[Evaluation Method]

Input: An attribute assignment system and a parse tree of an input string.

Output: An assignment of attribute values to the parse tree, if it exists. An error message, if it does not exist.

Method: Initially the set $S(X)$ is empty for each nonterminal X of the parse tree.

(1) Apply Algorithm 1 from leaves of the parse tree to the root of the parse tree.

(2) Apply Algorithm 2 from the root of the parse tree to leaves of the parse tree. Then we have an assignment $V(X)$ for each nonterminal X of the parse tree

Algorithm 1.

for each relation $(t_0, t_1, \ldots, t_n)$
at a production of nodes $(X_0, X_1, \ldots, X_n)$
do if $((t_1$ is in $S(X_1))$ or $(X_1$ is a terminal)) and
. . . and
$((t_n$ is in $S(X_n))$or $(x_n$ is a terminal))
then $S(X_0) := S(X_0) + \{t_0\}$ **fi**
od

Algorithm 2.

if X_0 is the root of the entire tree
then if size $(S(x_0)) = 0$ **then** error ( )
else $V(x_0) := \{$an element t in $S(X_0)\}$ **fi fi;**
if
for some relation $(t_0, t_1, \ldots, t_n)$
at a production of nodes $(X_0, X_1, \ldots, X_n)$ such that
$(t_0$ is in $V(X_0))$ and
$((t_1$ is in $S(X_1))$ or $(X_1$ is a terminal)) and
. . . , and
$((t_n$ is in $S(X_n))$ or $(X_n$ is a terminal))
then
$V(X_1) := \{t_1\}; \ldots; V(X_n) := \{t_n\}$ **fi**

Note that the uniqueness of global assignment to the parse tree can be determined in one-pass bottom-up evaluation of Algorithm 1, if we use multi sets instead of usual sets. A multi set is a set of occurrences of elements, and we may have many occurrences of same element in a multi set.

5.2 Example

Let us solve the example of Fig. 5 over a small finite set of addresses using our algorithm. Usually the solution of this type of problem is realized by a technique called backpatching. Backpatching is a technique of patching jump addresses of jump instructions when they are determined after generation. The parse tree of the input is shown in Figure 6. (For convenience the terminal i is replaced by corresponding variable name.)

If we directly apply our evaluation algorithm, there will be two approaches to the solution of this problem. The first solution holds a finite set of attribute values and apply intersection operations of Algorithm 1 repeatedly. Attributes at each nonterminal are (code, true, false, start, next). The size of the set of attribute values at each nonterminal decreases, when we perform intersection operations from leaves to the root. Finally, at the root, the size of the set of attribute values become one, and the value of the attribute code is identical with six quadruples shown in the section 4.3. In this particular case the broadcast phase of Algorithm 2 is rather

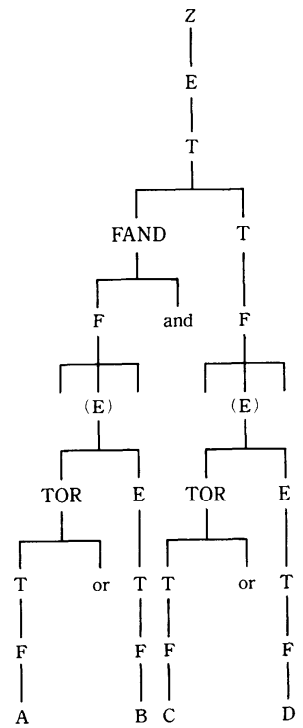

Fig. 6 Parse tree of a boolean expression.

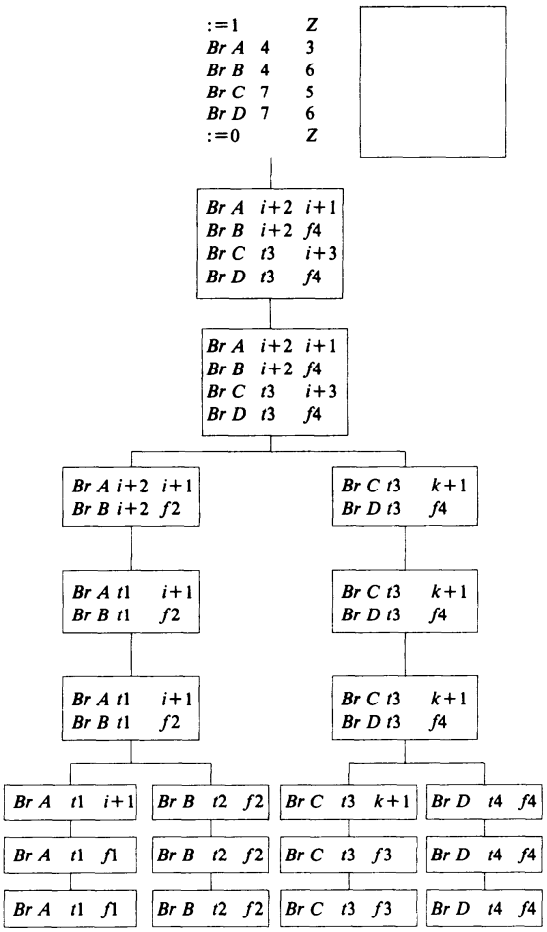

Fig. 7 Determination of attribute values.

redundant.

The second solution could use the concept of value numbers often used in the data-flow analysis. Instead of having a finite set of attribute values, we hold a value number (and value expressions) such as t_1, f_1, and i in a quadruple ($Br\ A\ t_1\ f_1$) where t_1 and f_1 are value numbers for $F.$ true and $F.$ false, and i is the value number for the start address of the code respectively. We perform intersection operations of value numbers and value expressions repeatedly, and get the final value of the code at the root. The determination of attribute values based on this approach is illustrated in Figure 7. In this particular case, the use of value numbers can be implemented using usual unification techniques.

5.3 Proofs

We give correctness proofs of algorithms in the section 5.1 by induction on the height of parse trees.

Theorem 1. The set $S(x_i)$ of Algorithm 1 gives a set of values at node X_i such that the value is assignable in global assignments of the subtree T of node x_i (with X_i as the subtree root) in terms of attribute assignment relations.

Proof. By induction on the height of the subtree T. When the height of the tree T is one, Theorem 1 holds trivially. Because the set of assignable values is a simple union of the values in relations.

Assume that Theorem 1 holds when the height is less than k. We denote the global assignment of the tree of height k by k-assignment. Assume that a production at the entire root is $X_0 \rightarrow X_1, X_2, \ldots, X_n$. By definition, a tree of height k could be considered as pasting of a tree of height one with the root X_0 and trees of height $k-1$ or less with the roots $X_1, X_2, \ldots, X_n$.
For $i \geqq 1$,

$S(X_i)$ at k-global assignment =
an intersection of $S(X_i)$ at m-global assignment ($k-1 \geqq m$) and (assignable values of X_i for nodes $(X_0, X_1, X_2, \ldots, X_n)$)

Hence we obtain the k-global assignment by our algorithm.

Corollary 1. The set $V(x_i)$ of Algorithm 2 gives a value at node x_i in a global assignment of attribute values to the parse tree, if it exists. Algorithm 2 gives an error message, otherwise.

Proof. Straightforward.

6. Discussions

6.1 Transformation

Attribute assignment systems often can be considered as starting descriptions, which will be transformed into descriptions of attribute grammars via a series of natural transformation techniques. This is true in the case of the example of Fig. 5. (We can naturally treat start, true, and false as inherited attributes, and next and code as synthesized attributes respectively.) Also in this case the description of the attribute grammar can be transformed into a description of action routines in terms of the use of backpatching. So this transformation process could be considered as an explanation of how we can come up with backpatching techniques.

6.2 Comparison

We mention an experimental comparison of backtracking approach and our set-theoretic approach to combinatorial problems. According to [8], Iwamoto measured the two evaluation methods for attribute assignment problems. The problem is assignment of integers to nodes of a complete binary tree of height n with r locally possible assignment relations at each production. Measured time in milliseconds is given in Fig. 8a and 8b. (By best cases and worst cases, we mean the case where the backtracking method works best and worst respectively. Backtracking was implemented by a simple recursive procedure.) Numbers above and below

n	2	3	4	5
$r=2$	26 58	24 68	28 82	24 94
$r=3$	48 110	42 140	46 166	42 198
$r=4$	88 222	84 282	82 344	88 414
$r=5$	184 450	186 580	186 714	182 856

Fig. 8(a) Comparison of best cases.

n	2	3	4	5
$r=2$	40 50	58 68	76 78	100 86
$r=3$	114 112	244 136	468 170	782 198
$r=4$	328 222	1022 284	2428 340	4956 418
$r=5$	988 446	4176 594	12760 728	31490 864

Fig. 8(b) Comparison of worst cases.

in a column show measured time by backtracking method and our method respectively. This experiment suggests our evaluation method is relatively efficient and stable in comparing with traditional backtracking method.

6.3 Generalization

A generalization of the evaluation method in this paper is possible. One possible extension of this method is to extend the domain of attributes from finite sets to manageable infinite sets such as regular sets or parenthesis languages. This extension would increase the capability of the evaluation method. However, it would have a large amount of overhead.

7. Conclusion

We have shown that our attribute assignment interpretation of non-procedural computing systems such as Prolog, Wijngaarden grammars, and attribute grammars is very natural and useful. We presented a formulation of attribute assignment systems. This view enables us to have a set-theoretic evaluation method of non-procedural computing systems with finite domains.

Historically it is quite interesting that what computer scientists of the 60's tried to forget was rediscovered by computer scientists of the 70's in a slightly different setting.

Acknowledgement

The author would like to express his gratitude to Mr. Norio Iwamoto for having discussions with him about his formulation of relational attribute grammars and his implementation based on object-oriented systems.

References

1. BOCHMANN, G. V. Semantic evaluation from left to right, *Comm. ACM*, **19**, 2 (Feb. 1976), 55-62.
2. CLEAVELAND, J. C. and UZGALIS, R. C. Grammars for Programming Languages, American Elsevier (1976).
3. GANZINGER, H. and RIPKEN, K. Operator identification in Ada: formal specification, complexity, and concrete implementation, *SIGPLAN Notices* **15**, 2 (1980), 30-42.
4. GOLDBERG, A. and ROBSON, D. Smalltalk-80: The language and its implementation, Addison-Wesley (1983).
5. HENDERSON, P. Functional programming: application and implementation, Prentice-Hall (1980).
6. HOGGER, C. J. Introduction to logic programming, Academic Press (1984).
7. ICHBIAH, J. D. et al. Rationale for the design of the *ADA programming language*, SIGPLAN Notices, **14**, 6 (1979).
8. IWAMOTO, N. Relational Attribute Grammars, *M. Eng. Thesis, Dept. of Comp. Sci.*, Yamanashi University (1985).
9. KATAYAMA, T. A hierarchical and functional programming based on attribute grammar, *5th International Conference of Software Engineering* (1981).
10. KAY, M. Functional Grammar, Technical Report, Xerox Palo Alto Research Center (1979).
11. KFOURY, A. J., MOLL, R. N. and ARBIB, M. A. A programming approach to computability, Springer-Verlag (1982).
12. KNUTH, D. E. Semantics of context-free languages, *Math. System Theory* **2**, 2 (1968), 127-145; *Correction*: **5**, 1 (1971), 95-96.
13. KOSTER, C. H. A. Affix grammars, Algol 68 Implementation, north-Holland (1971).
14. KOWALSKI, R. Algorithm=Logic+Control, *Comm.* **24**, 2 (1979), 147-155.
15. MANNA, Z. Mathematical theory of computation, McGraw-Hill (1974).
16. MARCOTTY, M., Ledgard, H. F. and BOCHMANN, G. V. A Sampler of Formal Definitions, Computing Surveys **8**, 2 (1976), 191-216.
17. ROGERS, H. Theory of recursive functions and effective computability, McGraw-Hill (1967).
18. REBELIK, J. and STEPANEK, P. Horn clause programs suggested by recursive functions, *Proc. Logic Programming Workshop, Hungary (1980).*
19. SINTZOFF, M. Existence of a Van Wijngaarden syntax for every recursively enumerable set, *Annale de la Societe Scientifique de Bruxelles*, 81 (1967), 115-118.
20. TOKUDA, T., TOKUDA, J., SASSA, M. and INOUE, K. Metanotion Chart for revised Algol 68, SIGPLAN Notices 12, 9 (1977), 11-14.
21. TOKUDA, T. Wijngaarden grammars as Knuthian grammars, *Proc. 20th Annual IPSJ Conference* (1979), 207-208.
22. TOKUDA, T. An Exercise in Transforming Wijngaarden Grammars into Knuthian Grammars, *Research Reports on Comp. Sci.* C-40, *Dept. of Info. Sci., Tokyo Inst. of Tech.* (1981).
23. VAN WIJNGAARDEN, A. et al. Revised Report on the Algorithmic Language ALGOL 68, Acta Informatica **5** (1974), 1-236.

(Received January 16, 1986; revised October 16, 1986)