Acta Cybernetica, Tom. 7, Fasc. 4, Szeged, 1986.

A new programming methodology using attribute grammars

E. Simon

Keywords: attribute grammars, automatic program generation, modular decomposition, programming methodology, program specification.

Abstract

Attribute grammars have been constructed for describing the static semantics of programming languages and have been shown useful in a wide variety of automatic compiler generations. This paper presents a new application of attribute grammars to specify hierarchical and functional programs. An algorithm to evaluate attribute grammars is demonstrated. Several attributes can be evaluated in parallel too. A simple model for generating PASCAL like programs is given. A new metalanguage PLASTIC is introduced as an adequate tool for specifying hierarchical and functional programs. A simple PLASTIC program is presented to help attain the new programming methodology.

1. Introduction

Over the last decade there has developed an acute awareness of the need to introduce abstraction and mathematical rigour into the programming process. This increased formality allows for the automatic manipulation of software, increasing productivity, and, even more importantly, the managebility of complex systems. Along those lines, attribute grammars (AG) of Knuth [6] constitute a formal mechanism for specifying translations between languages [2, 8, 11]. By automatically generating the inverse translators we would be able to translate any program written for one processor into the command language of any other processor [13]. There are some methods for incremental evaluation of AG to produce so called incremental compilers [3]. An essential question is how to verify the correctness of the AG specification. In contrast with the attribute evaluation problem, this has not been studied well and only a few results have been reported up to now [1, 5].

Although several efforts have been made to obtain efficient evaluators, the first good algorithm for attribute evaluation has been proposed by T. Katayama [4]. Principally this algorithm accepts absolutely noncircular AG although extension to general noncircular AG is straightforward. In the model nonterminal symbols are considered to be functions which map their inherited attributes to their synthesized attributes and associate procedures to realize these functions with the nonterminal symbols. The entire AG is then transformed into a set of mutually recursive procedures. When applied to an AG whose attribute evaluation process can be performed in a single pass from left-to-right, the algorithm can generate an evaluator which can be combined with the top-down parsers to result in the so-called recursive-descendent compilers if the underlying CF grammars are LL(k). However data dependency sometimes allows several attributes be evaluated parallel supposing that we have associated one procedure for each synthesized attribute.

As it is widely recognized, hierarchical specification techniques are the most promising methods in constructing complex and large softwares in well structured way, and in fact they are the most successfully used ones in practice as it is represented for example by SYCOMAP [10]. In these methodologies softwares are hierarchically decomposed into modules and they are successively refined until concrete and machine executable programs are obtained from their abstract specifications cf. CDL2. Although they are extremely natural and useful the current states seems to be that automatic program generation from the specifications and their verification are prevented due to the lack of strict formalization.

The hierarchical and functional programming methodology presented in this paper is based on attribute grammars. Applying the results of [4], we obtain a new program specification technique which stands mechanical program generation. In our approach we consider a program specification as an AG where program modules are represented by nonterminal symbols of the grammar, module decompositions correspond to production rules, input and output data of the modules correspond to attributes of the nonterminal symbols and computations done in the modules are specified by the semantic rules. Our methodology has the following three desirable properties. It allows hierarchical descriptions of complex functional programs in a very natural way. We have means to mechanically generate efficient procedural type programs from the descriptions and verification of their correctness can also be performed hierarchically.

In this paper we give our formalism and then the metalanguage PLASTIC is stated. Before presenting the program generation algorithm a simple example is shown. The PLASTIC system, implemented in PASCAL is now under development. The PLASTIC compiler is specified in HLP/PASCAL metalanguage [12].

2. Formal description

Essence of our approach is to use a mechanism based on the Knuth's attribute grammar [6] to describe programs. Therefore a hierarchical and functional program (or simply HFP) is a 6-tuple

$$(M, m_0, A, D, V, F)$$

where

(1) M is a set of modules. We assume that M contains the special modul called a *null module* which is used to terminate decomposition. The null modul is denoted by null symbol.

(2) $m_0 \in M$ is an *initial module*.

(3) A is a set of input and output *attributes* of modules. With any modul except the null module, there is associated a set of input and output data called attributes and the set of attributes of $X \in M$ is denoted by $A[X]$. $A[X]$ is a disjoint union of the set $\mathrm{IN}[X]$ of *input* attributes and the set $\mathrm{OUT}[X]$ of *output attributes*. They are called inherited and synthesized attributes, respectively, in the AG terminology.

(4) $D \subset M \times M^*$ is a finite set of *module decompositions*. An element $d \in D$ is called a decomposition and is denoted by

$$d\colon X_0 \to X_1 X_2 \ldots X_n \quad cond \quad C_d$$

for $X_0, \ldots, X_n \in M$. We say that the module X_0 can be decomposed into modules $X_1, X_2, \ldots, X_n$ if a *decomposition* condition C_d is satisfied. C_d specifies the condition in terms of input attributes of X_0. When a is an attribute of X_k, that is, $a \in A[X_k]$, $X_k \cdot a$ is called an attribute occurrence of the decomposition d. It is called an *input occurrence* (by an alternative denotation $X_k{\downarrow}a$) if $a \in \mathrm{IN}[X_k]$ and an output occurrence $(X_k{\uparrow}a)$ if $a \in \mathrm{OUT}[X_k]$.

(5) V is a set of value domains of attributes.

(6) F is a set of *attribute mappings* for describing functional equalities among attributes. Let d be a decomposition $X_0 \to X_1 X_2 \ldots X_n \in D$. For each output occurrence $v = X_0{\uparrow}a$ with $a \in \mathrm{OUT}[X_0]$ and input occurrence $v = X_k{\downarrow}a$ with $a \in \mathrm{IN}[X_k]$, $1 \leqq k \leqq n$, there exists a function $f_{d,v}$ to compute the value of v from the values of other attribute occurrences $v_1, \ldots, v_m$ in d. The set $D_{d,v} = \{v_1, \ldots, v_m\}$ is called *dependency set* of $f_{d,v}$. If we denote the value domain of v by domain (v), $f_{d,v}$ is a mapping domain $(v_1) \times \ldots \times$ domain $(v_m) \to$ domain (v).

That is, in every decomposition functions are specified to compute the values of outputs for main module and inputs to submodules.

Let us define a *decomposition tree* which shows the result of all decompositions applied to the initial module m_0. It corresponds the derivation tree of CF grammars and is defined recursively by the following

(1) the null module is a decomposition tree, and

(2) if $T_1, \ldots, T_n$ are decomposition trees with the root module $X_1, \ldots, X_n$, respectively, and $X_0 \to X_1 \ldots X_n$ *cond* C is a decomposition, then the tree

$$X_0[T_1, \ldots, T_n]$$

which consists of the root X_0 and the subtrees $T_1, \ldots, T_n$ is a decomposition tree.

A *computation tree* T is a decomposition tree whose nodes are labelled by attribute values in such a way that for any module X_0 in T and the decomposition $d\colon X_0 \to X_1 \ldots X_n$ *cond* C_d applied at the module the following conditions are satisfied

(i) the decomposition condition C_d is true,

(ii) for any output occurrence v of X_0 or input occurrence v of X_k $1 \leqq k \leqq n$, the following functional equality holds

$$v = f_{d,v}(v_1, \ldots, v_m) \quad \text{where} \quad D_{d,v} = \{v_1, \ldots, v_m\}.$$

It should be noted that a computation tree represents a particular execution of an HFP corresponding to the particular values of input data fed to the initial module.

3. The PLASTIC metalanguage

PLASTIC is a new metalanguage designed to support the use of abstractions in program construction. Work in programming methodology has led to the realization that three kinds of abstractions — procedural, control, and especially data abstractions — are useful in the programming process. Among these, only the procedural abstraction is supported well by conventional languages, through the procedure or subroutine. ALPHARD [9] and CLU [7] provide, in addition to procedures, novel linguistic mechanisms that support the use of data and control abstractions. In contradiction to these languages the PLASTIC system is altogether based on a few results of AG. In the module specifications, control abstraction is realized by the semantic functions and decomposition conditions. Data types can be refined successively as the decomposition proceeds.

A PLASTIC program consists of five parts. We first define some global data types for the procedures and functions. The auxiliary functions and procedures that are used in decomposition rules are declared in procedure declarations. The allowed primitive functions and procedures form a subset of those of PASCAL, since both the procedure type and the parameter types are restricted to allowed input-output attribute types. The interpretation of procedures and functions is the same as in PASCAL. Comments are indicated by the character %, whose appearance outside a proper string means that the rest of the line is interpreted as a comment and is skipped by the system. The strings belonging to the token class IDENTIFIER begin with a letter which is followed by letters or digits or underscores.

Before the module specifications the name of the initial module is given. The values of the input attributes of the initial module are assigned by read operations. The main part of a PLASTIC program is the module specification. We associate a set of input and output data with each module X. Computations done in the module X_0 is specified decompositionwise by giving a set of functional equalities which hold among attributes of X_0 and its submodules $X_1, \ldots, X_n$, and thus they are reduced to the computations done in submodules. Repeating the module decomposition process until terminal modules are reached completes the program design. If there are recursive modules or if there are modules whose decompositions are not unique there may occur numbers of trees each of which corresponds to a specific computation. We have attached declarations for data types of attributes to decompositions. They are refined successively as the decomposition proceeds. Different decompositions for a module are separated to versions. The input attribute occurrence can be denoted by $\downarrow$ while the output occurrence by $\uparrow$. In the attribute occurrences the name of the module to be decomposed must not be specified.

Simple copy rules of the form "$X \cdot a := Y \cdot b$" can often be left unwritten by applying the so-called elimination principle, if so desired. It is applicable in two situations. First, if a is an output attribute, then X must be the left-hand side of the decomposition and Y must be the only module on the right-hand side of the decomposition having an occurrence of attribute a. Alternatively, if a is an input attribute, then Y must be the left-hand side of the decomposition and X can be any of the modules on the right-hand side of the decomposition. In both cases the nonexistence of a rule for $X \cdot a$ is an indication to the PLASTIC system to include the copy rule in the decomposition. In the module and submodule specification the input and output attributes are separated by semicolon. The keywords "description", "specification", "module", "submodule", "version", "condition", etc. can be abbreviated to "descr", "spec", "mod", "submod", "vers", "cond" etc. We assumed that a PLASTIC program is deterministic, that is, decomposition conditions of distinct decompositions with the same left-hand side module do not become true simultaneously for any value of its input attributes.

In the last part of a PLASTIC description the user can prescribe the implementation commands. As we shall see data dependency sometimes allows several attributes to be evaluated simultaneously. In our system these attributes are evaluated in a single procedure call, because this reduces overheads due to procedure activations and increases chances of parallel execution. The keyword "parallel" stands for these output attributes which have to be evaluated simultaneously if it is possible. The default option for attribute evaluation is sequential. One of the major goals of PLASTIC is to provide a mechanism to support the use of good programming methodology. To meet this goal, we must provide more than just the language mechanism for the generator: we must also provide a way to specify their effects. A natural means of doing this for implementation is to specify how to realize the evaluation of an attribute. There are three different kinds of realization. The default option is procedural. In this case for each module X and output attributes a single procedure will be generated. The keyword "macro" stands for those output attributes which are evaluated by executing a macro call. If there are same precompiled procedures for so called null modules, they can be activated by a call "statement".

The problem of data abstraction and its detailed discussion is beyond the scope of this paper except giving a comment that every hierarchical specification methodology should be equipped with a hierarchical data abstraction mechanism and in the case of PLASTIC the algebraic abstraction would be most appropriate.

Figure 1 shows a PLASTIC solution of binary conversion. Suppose we have a file containing record of binary characters. In order to verify the conversional algorithm we have to compute the value of binary number $b=b_1b_2...b_n$ in two ways. Design a program that reads the character file and compute the binary numbers val1 and val2. The initial modul is START. We have attached declarations for data types of attributes to decompositions. We have assumed the existence of several functions on primitive data types, which are denoted by bold-face type letters. Their meaning will be selfexplanatory from their names. The common declarations for types, symbols and rule are written in the head of module descriptions. Copy-rules should not be specified, because they are generated automatically by the system.

4. Translation of PLASTIC program

Besides its static description, one of the outstanding features of PLASTIC specification technique is that we have means to translate mechanically the specification into machine executable forms. This is called attribute evaluation in the attribute grammar theory.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%     PLASTIC description for computing the value of binary     %%%%
%%%%          number b = b1b2...bn   in two ways given by          %%%%
%%%%          val1 (b1b2...bn) = b1*2↑(n-1)+val1 (b2...bn)         %%%%
%%%%          val2 (b1b2...bn) = 2*val2 (b1...bn-1)+bn             %%%%
%%%%          val1 () = val2 () = 0                                %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
begin description bin_conv
common data types
val1, val2, pos: integer; neg: boolean;
procedures
procedure read (var input: file of elem); ...;
function last (input: file of elem); elem; ...;
function remain (input: file of elem): boolean; ...;
...
initial module is start
specifications
 %%1%%
module start (↓input; ↑val1, ↑val2);
types input: file of elem;
submodule sign (↓elem; ↑neg);
          list (↓input, ↓pos; ↑val1, ↑val2);
version: 1
rule start=sign list;
do input<=read (input);
   list↓pos:=0;
   val1:=if sign↑neg then -list↑val1 else list↑val1;
   val2:=if sign↑neg then -list↑val2 else list↑val2;
   sign↓elem:=head (input);
   list↓input:=tail (input);
cond not empty (input);
version: 2
rule start=; do val1:=0; val2:=0;
cond always;
end start;
```

```
%%2%%
module sign(↓elem; ↑neg);
types elem: character;
rule sign = ;
version: 1
do neg:=true;
cond elem="—";
version: 2
do neg:=false; cond elem="+"; end sign;
%%3%%
module list (↓input, ↓pos; ↑val1, ↑val2);
submodule list, digit (↓input, ↓pos; val1, ↑val2);
version: 1
rule list=digit;
do % digit↓pos:=pos;              copy-rule
% digit↓input:=input;              copy-rule
% val1:=digit↑val1;                copy-rule
% val2:=digit↑val2;                copy-rule
% copy-rule will be generated without specification
cond empty (remain (input));
version: 2
rule list=list digit;
do digit↓input:=last(input);
   list↓input  :=remain(input);
   list↓pos    :=pos+1;
   val1        :=list↑val1+digit↑val1;
   val2        :=2*list↑val2+digit↑val2;
cond always;
end list;
%%4%%
module digit (↓elem, pos; ↑val1, val2);
types elem: character;
rule digit =;
version: 1
do val1:=0; val2:=0;
cond elem="0";
version: 2
do val1:=2**pos; val2:=1;
cond elem="1";
end digit;
implementation
val1, val2: parallel;
%          : statement;
sign, digit: macro;
start, list  : procedure;
end description bin_conv.
```

Figure 1

4.1. Notations

Let d: $X_0 \to X_1 X_2 \dots X_n$ be a decomposition. A *dependency graph* DG_d *for the decomposition* d, which gives dependency relationship among attribute occurrences of d, is defined by

$$\mathrm{DG}_d = (\mathrm{DV}_d, \mathrm{DE}_d)$$

where the node set DV_d is the set of all attribute occurrences of d and the edge set DE_d is the set dependency pairs for d. Formally

$$\mathrm{DV}_d = \{X_k \cdot a \mid k = 0, \dots, n \text{ and } a \in A[X_k]\}$$

$$\mathrm{DE}_d = \{(v_1, v_2) \mid v_1 \in D_{d, v_2}\}.$$

When a computation tree T is given a *dependency graph* DG_T *for the computation tree* T is defined to represent dependencies among attributes of nodes in T. DG_T is obtained by merging together DG_d's according to the decompositions in T.

Let T be a computation tree with root node $X \in M$. DG_T determines an *IO graph* $\mathrm{IO}[X, T]$ of X with respect to T. It gives an I/O relationship among attributes of X, which is realized by the decomposition tree T. That is

$$\mathrm{IO}[X, T] = (A[X], E_{\mathrm{IO}})$$

when an edge (i, s) is in $E_{\mathrm{IO}} \subset \mathrm{IN}[X] \times \mathrm{OUT}[X]$ iff there is in DG_T a path connecting the attribute occurrences $X{\downarrow}i$ and $X{\uparrow}s$ of the root T.

For general PLASTIC programs there may be finitely many IO graphs for $X \in M$ and we denote the set of these IO graphs by $\mathrm{IO}(X)$, that is

$$\mathrm{IO}(X) = \{\mathrm{IO}[X, T] \mid T \text{ is a computation tree}\}.$$

Let $\mathrm{IO}(X) = \{\mathrm{IO}_1, \dots, \mathrm{IO}_N\}$ where $\mathrm{IO}_k = (A[X_k], E_k)$. A *superposed IO* graph $\mathrm{IO}[X]$ is defined by

$$\mathrm{IO}[X] = (A[X], E), \quad E = \bigcup_{k=1}^{N} E_k$$

to represent possible IO relationship.

In order to define a set of attributes to be evaluated in parallel, let us introduce an OI graph the dual concept of IO graph, which specifies how the values of inherited attributes are effected by other attributes.

Let T be a computation tree which contains $X \in M$ as one of its leaf nodes. An *OI graph* $\mathrm{OI}[X, T]$ of X with respect to T is given by

$$\mathrm{OI}[X, T] = (A[X], E_{\mathrm{OI}}[T]), \; E_{\mathrm{OI}}[T] \subset A[X] \times \mathrm{IN}[X]$$

where $(a, i) \in E_{\mathrm{OI}}[T]$ iff there is in DG_T a path from v_a to v_i, where v_a and v_i are nodes for attributes a and i of the leaf node X. A *superposed OI graph* is defined in a similar way as $\mathrm{IO}[X]$.

We further define a *dependency graph* $\mathrm{DG}[X]$ *of the module* X as the union of IO graph and OI graph, that is

$$\mathrm{DG}[X] = (A[X], E_{\mathrm{IO}} \cup E_{\mathrm{OI}}).$$

For an absolutely noncircular PLASTIC description D a set $O \subset \mathrm{OUT}[X]$ of output attributes is said evaluable in parallel iff no $s_1, s_2 \in O$ are connected in $\mathrm{DG}[X]$.

An augmented dependency DG_d^* for the decomposition d is

$$\mathrm{DG}_d^* = (\mathrm{DV}_d^*, \mathrm{DE}_d^*)$$

where $\mathrm{DV}_d^* = \mathrm{DV}_d$, the set of attribute occurrences in d, and $e \in \mathrm{DE}_d^*$ iff $e \in \mathrm{DE}_d$ or $e = (X_k \cdot i, X_k \cdot s)$ for some $(i, s) \in \mathrm{IO}[X_k]$ and $k = 1, \ldots, n$. DG_d^* represents a relationship among attribute occurrences in d which is realized partly by attribute mappings and partly by computation trees.

A PLASTIC description is said to be absolutely noncircular [2] iff DG_d^* does not contain cycles for any $d \in D$. For an output attribute s of a module X of a PLASTIC program, its input set in $[s, X]$ is defined to be a set of input attributes which are required to evaluate s, that is

$$\mathrm{in}\,[s, X] = \{i \mid (i, s) \text{ is an edge of } \mathrm{IO}\,[X]\}.$$

We extend the function in $[s, X]$ to allow such O as its first argument

$$\mathrm{in}\,[O, X] = \bigcup_{s \in O} \mathrm{in}\,[s, X].$$

4.2. Translation algorithm

Let X be a module of an absolutely noncircular PLASTIC description $P = (M, m_0, A, D, V, F)$ and s an output attribute of X. We associate with each pair X, s a procedure

$$R_{X,s}(v_1, \ldots, v_m;\ v)$$

where $v_1, \ldots, v_m$ are parameters corresponding to the input attributes in $I = \mathrm{in}\,[s, X]$ and v is a parameter for s. It should be noted that input and output parameters are separated by semicolon. This procedure is intended to evaluate the output attribute when supplied the values of input attributes in I.

When given the value of the inherited attribute i_0 of the initial module m_0 we begin to evaluate the output attribute s_0 of m_0 by executing the procedure call statement

$$\textbf{call}\ R_{m_0, s_0}(u_0;\ v_0)$$

where u_0 and v_0 are variables corresponding to i_0 and s_0, respectively.

Now we are ready to describe how to construct the procedure $R_{X,s}(v_1, \ldots, v_m; v)$. The first thing the procedure $R_{X,s}$ must do in its body is to know the decomposition d which is applicable to the module X and perform a sequence $H_{d,s}$ of statements to compute the value of attribute occurrences in d, therefore $R_{X,s}$ is constructed in the following form,

```
procedure $R_{X,s}(v_1, \ldots, v_m;\ v)$
if $C_{d_1}$ then $H_{d_1,s}$ else
if $C_{d_2}$ then $H_{d_2,s}$ else
...
end
```

where $d_1, d_2, \ldots$ are decompositions (versions) with left side module X. We have assumed that the PLASTIC description is deterministic, that is decomposition conditions of disctinct decompositions with the same left side module do not become simultaneously true for any value of its input attributes.

The sequence $H_{d,s}$ is obtained in the following steps.

(1) Make the augmented dependency graph DG_d^*.

(2) Remove from DG_d^* nodes and edges which are not located on any path leading to $X_0 \uparrow s$ for $i \in I = \mathrm{in}[s, X_0]$. Denote the resulting graph by

$$\mathrm{DG}_d^*[s] = (V, E).$$

(3) To each attribute occurrence $x \in V' = V - \{X_0 \downarrow i | i \in \mathrm{IN}[X_0]\}$ assign a statement $\mathrm{st}[x]$ for evaluating X as follows.

Case 1. If $x = X_k \downarrow i$ for some $i \in \mathrm{IN}[X_k]$ and $k = 1, \ldots, n$ or $x = X_0 \uparrow s (=v)$ for the attributes $s \in \mathrm{OUT}[X_0]$, then $\mathrm{st}[x]$ is the assignment statement

$$x := f_{d,x}(z_1, \ldots, z_r)$$

where $f_{d,x}$ is the attribute mapping for the attribute occurrence x and $D_{d,x} = \{z_1, \ldots, z_r\}$.

Case 2. If $x = X_k \uparrow t$ for some $t \in \mathrm{OUT}[X_k]$ and $k = 1, \ldots, n$, then $\mathrm{st}[x]$ is the procedure call statement

call $R_{X_k,t}(w_1, \ldots, w_k;\ x)$

where $w_1, \ldots, w_k = \{X_k \downarrow i \,|\, i \in \mathrm{in}[t, X_k]\}$.

(4) Let $x_1, \ldots, x_N$ be elements in V' which are listed according to the topological ordering determined by E, i.e., if $(x_a, x_b) \in E$ then $a < b$. Then $H_{d,s}$ becomes as follows.

$$\mathrm{st}[x_1];\ \ldots;\ \mathrm{st}[x_N]$$

Note that statements in $H_{d,s}$ satisfy the single assignment rule. It is easy to see that the ordering $x_1, \ldots, x_N$ ensures values of attribute occurrences are determined consistently if the PLASTIC description is absolutely noncircular.

We first construct the procedure R_{m_0,s_0} by the algorithm we have stated. Body of R_{m_0,s_0} may contain calls for other procedures $R_{X,s}$'s and they are constructed in the same way. Repeat this process until no more new procedures appear.

In the case of parallel evaluation we assign a single procedure

$$R_{X,O}(v_1, \ldots, v_m;\ u_1, \ldots, u_n)$$

to each set O which is evaluable in parallel instead of assigning n procedures, where $u_1, \ldots, u_n$ are parameters corresponding to output attributes in O and $v_1, \ldots, v_m$ are those for attributes in $\mathrm{in}[O, X]$.

Construction of $R_{X,O}$ parallels to that of $R_{X,s}$ except a few points. As in the case of $R_{X,s}$, the procedure $R_{X,O}$ has the following form.

```
procedure R_{X,O}($v_1, \ldots, v_m;\ u_1, \ldots, u_n$)
if $C_{d_1}$ then $H_{d_1,O}(v_1, \ldots, v_m;\ u_1, \ldots, u_n)$ else
if $C_{d_2}$ then $H_{d_2,O}(v_1, \ldots, v_m;\ u_1, \ldots, u_n)$ else
...
end
```

For a decomposition $X_0 \to X_1 X_2 \dots X_n$ and $O \in S[X_0]$ which is evaluable in parallel, construction of statement sequence $H_{d,O}$ proceeds in the following steps.

(1) Make DG_d^*.

(2) Make $\mathrm{DG}_d^*[O]=(V, E)$ by removing from DG_d^* nodes and edges which are not located on any path leading to $X_0{\uparrow}s$ for $s \in O$.

(3) For each $k=1, \dots, n$ decompose the set

$$\mathrm{OUT}^*[X_k] = \mathrm{OUT}[X_k] \cap \{t \mid X_k \cdot t \in V\}$$

into a set of mutually disjoint subsets

$$O_{k1}, O_{k2}, \dots, O_{kr}$$

such that each O_{kj} is evaluable in parallel. When the decomposition is not unique, we should choose a maximal decomposition, that is, one where the number v becomes minimum, to attain high efficiency of evaluation.

(4) Let $\mathrm{DG}_d'[O]=(V', E')$ be a graph obtained from $\mathrm{DG}_d^*[O]$ by grouping elements of each O_{kj} into a single node $v_{kj} \in V$. Formally

$$V' = \{g[v] \mid v \in V\}$$

$$E' = \{(g[u], g[v]) \mid (u, v) \in E\}$$

where g is a function defined by

$$g[v] = \begin{cases} v_{kj} & \text{if } v = X_k \cdot s \text{ for some } s, k \text{ and } j \text{ such that } s \in O_{kj} \\ v & \text{otherwise.} \end{cases}$$

(5) To each element x in $V_0 = V' - \{X_0 \cdot i \mid i \in \mathrm{IN}[X_0]\}$ assign a statement $\mathrm{st}[x]$ as follows.

Case 1. If $X = X_k{\downarrow}i$ for some $i \in \mathrm{IN}[X_k]$ and $k=1, \dots, n$, or $X = X_0{\uparrow}s$ then $\mathrm{st}[x]$ is the assignment statement

$$x := f_{d,x}(z_1, \dots, z_r)$$

where

$$D_{d,x} = \{z_1, \dots, z_r\}.$$

Case 2. If $X = v_{kj}$ then $\mathrm{st}[x]$ is the procedure call statement

$$\textbf{call}\, R_{X_k, O_{kj}}(w_1, \dots, w_h;\ x_1, \dots, x_c)$$

where

(1) $$\{w_1, \dots, w_h\} = \{X_k{\downarrow}i \mid i \in \mathrm{in}[O_{kj}, X_k]\}$$

and

(2) $$\{x_1, \dots, x_c\} = \{X_k{\uparrow}t \mid t \in O_{kj}\}.$$

(6) Same as 4. for $H_{d,s}$ in the sequential case.

Translation of the entire attribute grammar into the corresponding program is similar to the one given in this section. Let O be a set of output attributes of the initial modul. We start from constructing the procedure $R_{S,O}$ and then proceed to procedures which are called in it.

RESEARCH GROUP ON THEORY OF AUTOMATA
HUNGARIAN ACADEMY OF SCIENCES
SOMOGYI B. U. 7
H—6720, SZEGED

References

[1] DERANSART, P., Logical attribute grammars, *In:* Information Processing' 83, E. Mason ed., North-Holland, 463—470.
[2] GYIMÓTHY, T., E. SIMON, and Á. MAKAY, An implementation of the HLP, Acta Cybernetica 6, 3, (1983), 316—327.
[3] JALILI, F., A general incremental evaluator for attribute grammars, Science of Computer Programming 5, 1 (1985), 83—96.
[4] KATAYAMA, T., Translation of attribute grammar into procedures, Department of Comp. Science, Tokyo Institute of Technology, TR CS-K8001.
[5] KATAYAMA, T., and Y. HOSHINO, Verification of attribute grammars, Department of Comp. Science, Tokyo Institute of Technology, TR CS-K8003.
[6] KNUTH, D. E., Semantics of context-free languages, Math. Syst. Theory 2, 2 (1968), 127—145.
[7] LISKOV, B., et al, Abstraction mechanisms in CLU, CACM 20, 8 (1977), 564—572.
[8] RÄIHÄ, K-J., et al, Revised report on the compiler writing system HLP78, Department of Comp. Science, University of Helsinki, TR A-1983-1.
[9] SHAW, M., et al, Abstraction and verification in ALPHARD: Defining and specifying iteration and generators, CACM 20, 8 (1977), 553—563.
[10] SIMON, E., Formal definition of the SYCOMAP system, *In:* Preprints of the Second Hungarian Computer Science Conference, Budapest, 1977, 738—759.
[11] SIMON, E., Language extension in the HLP/SZ system, Acta Cybernetica 7, 1 (1984), 89—97.
[12] TOCZKI, J., et al., On the PASCAL implementation of the HLP, to be published *In:* Proc. of 4th Hungarian Computer Science Conference, Győr, 1985 (to appear).
[13] YELLIN, D., and Eva-Maria. M. Mueckstein, Two-way translators based on attribute grammar inversion, to be published *In:* Proc. of 8th International Conference on Software Engineering, 1985, 17 pp.

(Received Aug. 27, 1985.)