Translation between string and graph representations of programs and data may be formally defined by means of pair grammars. A pair grammar is composed of a pair of grammars whose rules and nonterminals are paired. The pair grammar defines a correspondence between elements of the languages defined by the two grammars. This correspondence may be viewed as a definition of the translation of the elements of one language into the elements of the other. Of particular interest is the case in which the first language is a set of strings and the second is a set of directed graphs with labeled arcs and nodes.

Preliminary to the definition of pair grammars, a class of graph grammars are defined which are a generalization of ordinary context-free grammars. A graph grammar defines a language composed of a set of directed graphs. Pair grammars are constructed from pairs of graph grammars. Each unambiguous pair grammar defines a reversible function mapping one graph language onto another. Special cases of interest include string-to-graph, graph-to-string, and string-to-string mappings. In the general case a pair grammar defines a transformation on a set of graphs.

Two extensions to the elementary pair grammars allow representation of hierarchies of graphs and constructs such as labels and go to statements. Examples are given of the translation of a major subset of Algol into flowchart graphs and the translation of Lisp S-expressions into list structure graphs with structured atoms.

I. INTRODUCTION

Formal analysis of the semantics of programming languages requires a representation of programs as abstract structures. For the study of syntax, programming languages are often represented formally as sets of strings. For analysis of their semantic structure, however, a representation of programs in the form of directed graphs or trees is quite common. The mapping which relates the string representation of a program with its graph or tree representation has seldom been studied formally. This paper introduces
a type of formal grammar, called a pair grammar, which may be used to define translations of strings into graphs and graphs into strings. Pair grammars may also be used to define transformations on graphs.

The problem of formalizing translation between languages, purely formal languages, and programming languages as well as natural languages, has received much attention in the literature. However, the problem has been considered primarily in the context of string-to-string translation; that is, the language to be translated is considered to be a set of strings, as is the language into which the translation is made. A translator is simply a mapping of the strings in the first set into strings in the second set, although the specification of this mapping may be extremely complex, as in the case of a compiler. Recent work on “syntax-directed transduction” (see, e.g., Lewis and Stearns [7]) has contributed to the development of a formal theory of string-to-string translation.

Translation from strings to more complex structures such as trees, graphs, or arrays, has been treated far less frequently. Usually translation from a string to a structure is considered more as an intermediate step in a string-to-string translation than as a translation of interest itself. For example, in a translator-writing system an input string is commonly translated into a complex structure of tables, parse trees, code segments, etc., before finally producing an output string in another language (usually assembly language) which can be used for further processing. Here the translation is ultimately one of strings into strings, but the intermediate representation is as a more complex structure. Similarly, context free grammars may be viewed as defining translations of strings into their parse trees. But production of the parse tree is usually considered not as an end in itself but as an intermediate step (to be dispensed with whenever possible) in the determination of syntactic correctness or in the production of “object code” strings in some other language. Much of the work on recognition-oriented grammars, parsing algorithms, and automata recognizers has in fact been concerned with the problem of how to parse a string without constructing the parse tree explicitly (e.g., Knuth [5], Wirth [17]).

The most notable exceptions to this trend are to be found in studies concerned more with semantics than syntax. The works of Lucas et al. [8] and Landin [6], among others, emphasize that a precise definition of the semantics of a program requires its translation from string form into a structure (such as a tree or a tree-like “applicative expression”) which may then be executed by an abstract machine. A number of other studies of the semantics of programs begin with a directed graph representation (e.g., Manna [9], Floyd [2], Narasimhan [12]) as the assumed program structure without explicit consideration of how the usual string representation of programs may be translated into these structures. In all of these studies, there is an emphasis on representation of programs as structures which are more complex than strings.

On the practical side, it is clear that programming language translators, in general, expend a large amount of effort in translating programs, which are input in string form, into structures of various kinds, either structures which can be readily and
efficiently executed by an interpreter (as in the case of Lisp and Snobol 4) or structures which can be optimized for ultimate production of efficient machine code (as in the case of Fortran and Algol). Thus, while most formal studies have primarily considered programs as strings and, correspondingly, languages as sets of strings, actual language processors, and some formal work on semantics, deal with more highly structured representations.

If a program is represented by a tree or directed graph, then it is natural to consider a language as a set of these structures. There is little in the literature concerning languages composed of trees or graphs rather than strings. Graph languages (languages composed of sets of graphs) have been considered primarily in the context of picture processing, in the definition of two-dimensional languages of classes of pictures such as cloud-chamber events or characters projected on a grid (see, e.g., Miller and Shaw [11], Narasimhan [12]). Pfaltz and Rosenfeld [14] have introduced a type of "web grammar" for graph languages. Lucas et al. [8] have developed a meta-language (in fact, a grammar) for defining classes of trees in conjunction with their work on formal definitions of programming languages.

Considering the relative lack of study of languages composed of sets of structures, it is not surprising that the question of translation between languages of strings and languages of structures has received even less attention. The interest in such a study arises primarily from the observation that if a programming language is considered as a set of structures such as graphs or trees in addition to its usual definition as a set of strings, then intuitively it seems that each "program" should have two representations—one as a string and the other as a structure. Is it possible to define formally this pairing of strings and structures? If so, then such a pairing may also serve as a formal definition of the translation between strings and structures. It is these considerations that provide the motivation for the study of languages of structures and for the study of translations between languages of strings and languages of structures which is undertaken in this paper.

Directed graphs with labeled arcs and nodes are taken as the basic structures of interest here. Graphs of this type have been widely used for the representation of programs (e.g., Manna [9], Kaplan [4], Good [3], Pratt [15]) and both trees and strings form special classes of these graphs.

The first section of this paper develops the concept of graph grammars. Graph grammars are formal grammars similar to ordinary context-free grammars, except that the language defined is a set of graphs rather than a set of strings. Context-free grammars, in fact, form a subclass of the graph grammars. Graph grammars as defined here are similar to the "web grammars" of Pfaltz and Rosenfeld [14].

The second section defines the major new construct of this paper, the pair grammar. Pair grammars are types of formal grammar composed of a pair of graph grammars over the same alphabets together with a formal correspondence between the rules of the two grammars and the nonterminals in the rules. The language defined by a pair
PAIR GRAMMARS AND GRAPH LANGUAGES

The following sections develop two extensions of pair grammars needed to handle the translation of actual programming languages from string representation into graph representation. One extension is due to the need to handle nodes which contain strings rather than single terminal symbols, as well as the need for multileveled graphs for description of subprogram and block structure. These extensions allow graph languages to be composed of multileveled hierarchical graphs as well as single-leveled graphs of the original type.

The second extension is made to allow translation, into appropriate graph structures, of labels and go to statements as well as other syntactic constructs ordinarily requiring a "symbol table" for translation. Terminal nodes in a graph are allowed to contain a pair of elements: a node label and a node value. A graph is in reduced form if all nodes with identical labels have been combined. The utility of these extensions is demonstrated by a pair grammar for translating a subset of Algol into graphs and by a pair grammar for translating Lisp S-expressions into the internal list structure form which is commonly used by Lisp interpreters.

A final section gives a brief example of the use of a pair grammar to define a transformation on a set of graphs.

II. GRAPH LANGUAGES AND GRAPH GRAMMARS

A graph language is a language composed of a set of directed graphs with labeled nodes and arcs. A graph grammar is a generalization of an ordinary context-free grammar which defines a graph language. Since most of the concepts and terms commonly used in discussion of context-free grammars, such as "terminal symbol", "nonterminal symbol", "grammar rule", "derivation", "parse tree", "ambiguity", "rewriting", etc., have natural analogs in the discussion of graph grammars, these terms are carried over in the discussion here wherever possible.

A graph grammar generates a language of "terminal graphs" in much the same way a context-free grammar generates a set of terminal strings. One begins with a graph containing a single nonterminal node and proceeds through a sequence of "rewritings" to the terminal graph. At each rewriting step a single nonterminal node is replaced by a graph according to a rule of the graph grammar. The rewriting
continues until no nonterminal nodes remain. In this section these ideas are defined formally.

**Graph Languages**

Assume $\Omega_M$ and $\Omega_A$ are finite alphabets of distinct symbols. $\Omega_M$ and $\Omega_A$ are the sets of node values and arc labels, respectively.

**Definition 2.1.** A graph $G$ over $\Omega_M$, $\Omega_A$ is a triple $(N, V, E)$ where $N$ is a finite set (of nodes),

- $V : N \rightarrow \Omega_M$ (the node value function, defines the value of each node),
- $E \subseteq N \times \Omega_A \times N$ (the arc set, defines the arcs of $G$ and their labels).

If $(n, a, m) \in E$, then there is an arc from node $n$ to node $m$ with label $a$. If $G$ is a graph, we shall use $N_G$, $V_G$, and $E_G$ to denote the node set, node value function, and arc set of $G$, respectively.

**Definition 2.2.** If $\Omega_M$ and $\Omega_A$ are alphabets, then

$$\mathcal{G}^*(\Omega_M, \Omega_A) = \{G \mid G \text{ is a graph over } \Omega_M, \Omega_A\}.$$ Where $\Omega_M$ and $\Omega_A$ are clear, we shall write simply $\mathcal{G}^*$ for $\mathcal{G}^*(\Omega_M, \Omega_A)$.

**Definition 2.3.** A graph language over $\Omega_M$, $\Omega_A$ is a subset of

$$\mathcal{G}^*(\Omega_M, \Omega_A).$$

**Definition 2.4.** (Equivalence) If $G$ and $H$ are elements of $\mathcal{G}^*$, then $G \equiv H$ iff there exists a function $f : N_G \xrightarrow{1:1} N_H$ such that:

1. $\forall n \in N_G$, $V_G(n) = V_H(f(n))$, and
2. $(n, a, m) \in E_G \iff (f(n), a, f(m)) \in E_H$.

**Definition 2.5.** If $G$, $H$, and $K \in \mathcal{G}^*$ then $H$ and $K$ form a partition of $G$ if

1. $N_H \cap N_K = \emptyset$ and $N_G = N_H \cup N_K$,
2. $\forall n \in N_H$, $V_H(n) = V_G(n)$ and $\forall m \in N_K$, $V_K(m) = V_G(m)$
3. $E_H = \{(n, a, m) \in E_G \mid n, m \in N_H\}$, and
4. $E_K = \{(n, a, m) \in E_G \mid n, m \in N_K\}$.

**Graph Grammars**

**Definition 2.6.** A graph grammar is a quintuple $(\Omega_N, \Omega_T, \Omega_A, S, R)$, where
\( \Omega_N \) is a finite alphabet (of *nonterminal symbols*),
\( \Omega_T \) is a finite alphabet (of *terminal symbols*),
\( \Omega_A \) is a finite alphabet (of *arc labels*),
\( S \in \Omega_N \) (the *initial nonterminal*),

and \( R \) is a finite set of *rules*, where each rule in \( R \) is a quadruple \((G, H, I, O)\) such that
\( G \in \mathcal{G}^*(\Omega_N \cup \Omega_A) \) and \( G \) has only a single node and no arcs,
\( H \in \mathcal{G}^*(\Omega_N \cup \Omega_T \cup \Omega_A), \) and \( N_H \neq \varnothing, \)
\( I \in N_H \) (the *input node*),

and
\( O \in N_H \) (the *output node*).

A grammar rule \((G, H, I, O)\) may be written conveniently in the form
\[ A \rightarrow H^I^O, \]
where \( A \) is the nonterminal symbol which is the value of the single node in \( G \).

Each rule specifies a possible rewriting of a node whose value is a nonterminal as follows:

**Definition 2.7.** If \( \mathcal{Q} = (\Omega_N, \Omega_T, \Omega_A, S, R) \) is a graph grammar and \( G \) and \( H \) are elements of \( \mathcal{G}^*(\Omega_N \cup \Omega_T \cup \Omega_A) \), then

\[ (2.7.1) \quad G \Rightarrow H \quad (H \text{ is directly derived from } G, \text{ according to grammar } \mathcal{Q}) \text{ iff there exists a rule } (L, K, I, O) \text{ in } R \text{ such that } \]

(a) \( G \) can be partitioned into graphs \( G' \) and \( G'' \), where \( G' \equiv L, (G' \equiv L \) implies \( N_{G'} \) contains only a single node; call it \( n_a \)),

(b) \( H \) can be partitioned into graphs \( H' \) and \( H'' \) such that

(i) \( H'' \equiv G'' \),

(ii) \( H' \equiv K \),

(iii) \( E_H = E_{H'} \cup E_{H''} \cup \{(m, a, I) \mid (m, a, n_a) \in E_G \text{ and } m \neq n_a\} \)
\[ \cup \{(O, a, m) \mid (n_a, a, m) \in E_G \text{ and } m \neq n_a\} \]
\[ \cup \{(O, a, I) \mid (n_a, a, n_a) \in E_G\}. \]

The derivation of \( H \) from \( G \) according to the rule \( A \rightarrow K^I \) consists simply of replacing a node \( n_a \) in \( G \) whose value is \( A \) by the graph \( K \). Arcs leading into \( n_a \) are replaced by arcs leading to \( I \), arcs exiting from \( n_a \) are replaced by arcs exiting from \( O \), and any loop arcs on \( n_a \) are replaced by arcs from \( O \) to \( I \).
(2.7.2) \( G \overset{Q}{\Rightarrow} H \) (\( H \) is derived from \( G \), according to grammar \( Q \)) iff there exists a sequence \( X_1, X_2, \ldots, X_n \), each \( X_i \in \mathcal{F}^*(\Omega_N \cup \Omega_R, \Omega_A) \) such that \( G \Rightarrow X_1, H \Rightarrow X_n \), and \( X_i \overset{Q}{\Rightarrow} X_{i+1} \), for \( i = 1, 2, \ldots, n - 1 \).

(2.7.3) The language \( \mathcal{L}_Q \) defined by \( Q \) is \( \mathcal{L}_Q = \{ G \in \mathcal{F}^*(\Omega_R, \Omega_A) \mid S_Q \overset{Q}{\Rightarrow} G \} \), where \( S_Q \), the initial graph of \( Q \), is \( S_Q = (\{n\}, \{(n, S)\}, \varphi) \).

Ambiguity

In context-free languages, a string is ambiguous if it has two distinct "canonical" derivations. A canonical derivation is simply one in which the choice of nonterminal to rewrite is fixed at each step, as in a "leftmost" derivation where the leftmost nonterminal is always rewritten. An analogous approach may be used in defining ambiguity in graph grammars. Given a graph \( G \) in \( \mathcal{F}^*(\Omega_N \cup \Omega_R, \Omega_A) \), the nodes of \( G \) have no natural ordering analogous to that of symbols in a string. However, an ordering may easily be imposed as follows.

Let \( Q \) be a graph grammar. For each grammar rule \( A \rightarrow H \) of \( Q \), order the set \( \mathcal{N}_H \) of nodes in \( H \) arbitrarily as \( \mathcal{N}_H = (n_1, n_2, \ldots, n_k) \). Then given a graph \( G \) with some arbitrary ordering \( \mathcal{N}_G \) of its nodes and a graph \( K \) with a node ordering \( \mathcal{N}_K \), we say \( K \) is leftmost derived from \( G \) iff (1) \( G \Rightarrow K \) by rewriting the leftmost nonterminal node \( n \) in \( G \), as defined by the ordering \( \mathcal{N}_C \), and (2) \( \mathcal{N}_K \) is derived from \( \mathcal{N}_G \) by simply substituting for \( n \) in \( \mathcal{N}_C \) the sequence of new nodes introduced into \( K \) by the rewriting, ordered according to the order in the graph grammar rule used in the rewriting. As in context-free grammars, a graph grammar is ambiguous if the language it generates contains a graph with two distinct leftmost derivations.

String Languages and Grammars

The preceding definitions of graph languages and graph grammars are straightforward extensions of the ordinary definitions for context-free languages and grammars. This follows from the observation that any string may equally be considered as a simple type of graph. If \( \Omega_R \) is an alphabet, then an ordinary string \( x \) of symbols from \( \Omega_R \), \( x = a_1a_2 \cdots a_n \) corresponds to the graph \( X = (N, V, E) \) over \( \Omega_R \) and \( \Omega_A = \{A\} \) (\( A \) the null label), where

\[ N = \{m_1, m_2, \ldots, m_n\}, \]
\[ V(m_i) = a_i, \quad i = 1, 2, \ldots, n, \]
and
\[ E = \{(m_i, A, m_{i+1}) \mid i = 1, 2, \ldots, n - 1\}. \]

Under this correspondence, Definitions 2.1–2.7 become the standard definitions of context-free languages and grammars for the special case of strings. Note, in particular, that (1) \( \mathcal{F}^*(\Omega_R, \{A\}) \) corresponds to \( \Omega_R^* \) and (2) the context-free grammar rule \( A \rightarrow a_1 \cdots a_n \) becomes the graph grammar rule \( A \rightarrow X_i \), where \( X \) is the graph
Initial non-terminal symbol: \(< program >\)

<table>
<thead>
<tr>
<th>Rule Number</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(&lt; program &gt; ) ::= (&lt; compound stmt &gt;)</td>
</tr>
<tr>
<td>2</td>
<td>(&lt; compound stmt &gt; ) ::= (&lt; statement &gt;)</td>
</tr>
<tr>
<td>3</td>
<td>(&lt; compound stmt &gt; ) ::= (&lt; compound stmt &gt;)</td>
</tr>
<tr>
<td>4</td>
<td>(&lt; statement &gt; ) ::= (&lt; assignment &gt;)</td>
</tr>
<tr>
<td>5</td>
<td>(&lt; statement &gt; ) ::= (&lt; loop &gt;)</td>
</tr>
<tr>
<td>6</td>
<td>(&lt; statement &gt; ) ::= (&lt; branch &gt;)</td>
</tr>
<tr>
<td>7</td>
<td>(&lt; loop &gt; ) ::= (&lt; predicate &gt;) (&lt; compound stmt &gt;)</td>
</tr>
</tbody>
</table>

**Rule Number**

<table>
<thead>
<tr>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

**Fig. 1.** A graph grammar for a simple flowchart language.
corresponding to the string \( a_1 \cdots a_n \), and \( I \) and \( O \) are the nodes whose values are \( a_1 \) and \( a_n \) in \( X \), respectively.

**Example 1: A Simple Flowchart Language**

Figure 1 gives a simple graph grammar for a flowchart language. The language generated by this grammar consists of an infinite set of finite graphs, each of whose arcs are labeled either \( T \) or \( F \) and whose node values are either assignment statements from the set \( \{S_1, S_2, \ldots, S_n\} \), predicates from the set \( \{P_1, P_2, \ldots, P_m\} \), or the null value \#.

Figure 2 contains a typical terminal graph generated from the grammar of Fig. 1.

![Diagram of a typical terminal graph](image1)

**Fig. 2.** A typical terminal graph generated by the grammar of Fig. 1.

Figure 1 and succeeding examples use the following conventions:

1. Arcs are represented by arrows. A node is represented by
   (a) An oval if the value of the node is a non-terminal symbol;
   (b) A rectangle if the value of the node is a terminal symbol.
   (This distinction between the shapes of nodes is only made to aid the intuition, it has no formal significance.)

2. Nonterminal symbols are represented by strings within pointed brackets (e.g., \( \langle \text{program} \rangle \)) as in ordinary BNF notation. The “\( : := \)” is used to separate the left and right sides of each grammar rule.
(3) Node values are written inside the node oval. Arc labels are written on the arc arrow.

(4) In grammar rules, the input node and output node of the right side graph are designated by an I and O outside the respective nodes.

(5) Alternative rewritings of the same nonterminal are usually written as separate grammar rules in place of the "|" of BNF notation.

Conclusion

It would appear that the theory of graph grammars may be developed along similar lines to the theory of context-free grammars (although the connections with automata theory are not clear). Generating or parsing a graph according to the rules of a graph grammar yields the usual derivation tree (although the combinatorics of parsing become much more involved); graph grammars may be ambiguous or unambiguous; more restricted and more general classes of graph grammars may be defined and studied; the decidability of various questions concerning such grammars can be investigated, and so forth. While such questions are of a great deal of interest, at this point we choose to leave the formal development of graph grammars and proceed to consider instead the formalization of translation between strings and graphs (and between graphs) by means of a technique of pairing the rules in two graph grammars. The definition of this "pair grammar" concept and an investigation of some of its implications for programming language translation form the body of the remainder of this paper.

III. PAIR GRAMMARS AND FORMALIZED TRANSLATION

A pair grammar is simply a pair of graph grammars over the same alphabets, together with a correspondence defined between rules in the grammars and between nonterminal nodes in the rules so that corresponding nodes have the same nonterminal values.

Definition 3.1. If (1) $\Omega_N$, $\Omega_T$, $\Omega_A$ are alphabets, as before,

(2) $G$ and $H$ are in $\mathcal{G}^*(\Omega_N \cup \Omega_T \cup \Omega_A),$

(3) $N^NT_G = \{n \in N_G \mid V_G(n) \in \Omega_N\}$ (the nonterminal nodes of $G$)

and

(4) $N^NT_H = \{n \in N_H \mid V_H(n) \in \Omega_N\}$ (the nonterminal nodes of $H$),

then a nonterminal node pairing of $G$ and $H$ is a function $h : N^NT_G \rightarrow N^NT_H$, such that $\forall n \in N^NT_G, V_G(n) = V_H(h(n))$. 


DEFINITION 3.2. A pair grammar is a quintuple \((\Omega_N, \Omega_T, \Omega_A, S, P)\), where 
\(\Omega_N, \Omega_T, \Omega_A\) are alphabets of nonterminals, terminals, and arc labels, respectively, as 
before, \(S \in \Omega_N\), the initial nonterminal, and \(P\) is a finite set of triples \((\ell, r, h)\), where 
\begin{enumerate}
  \item \(\ell\) and \(r\) are graph grammar rules as in Definition 2.6,
  \item if \(\ell\) is the rule \(A \rightarrow G^I\) and \(r\) is the rule \(A' \rightarrow H^I\), then \(A = A'\)
\end{enumerate}
and 
\begin{enumerate}
  \item \(h\) is a nonterminal node pairing of \(G\) and \(H\).
\end{enumerate}

DEFINITION 3.3. Let \(Q = (\Omega_N, \Omega_T, \Omega_A, S, P)\) be a pair grammar.

(3.3.1) If \(p = (\ell, r, h) \in P\), then 
\(\ell\) is the left rule of \(p\), 
\(r\) is the right rule of \(p\), 
and 
\(h\) is the nonterminal node pairing of \(p\).

(3.3.2) The left (right) grammar of \(Q\) is the graph grammar \((\Omega_N, \Omega_T, \Omega_A, S, R)\), 
where \(R\) is the set of left (right) rules of \(P\).

(3.3.3) The left (right) language of \(Q\) is the language defined by the left (right) 
grammar of \(Q\).

The language defined by a pair grammar \(Q\) consists of ordered pairs of graphs from 
the left and right languages, respectively, of \(Q\). The pair grammar defines how these 
graph pairs may be generated in parallel from the same initial graph. At each inter-
mediate stage in the generation we have a pair of graphs, each containing some non-
terminal nodes, and a correspondence between these nonterminal nodes. At each 
rewriting, a corresponding pair of nonterminal nodes, one in each graph, is rewritten 
according to a rule of the pair grammar, and a new correspondence is set up between 
nonterminal nodes in the resulting graphs using the nonterminal pairing of the 
grammar rule. Formally:

DEFINITION 3.4.

(3.4.1) A graph pair \(X\) (over \(\Omega_N, \Omega_T, \Omega_A\)) is a triple \(X = (G, H, h)\), where 
\begin{enumerate}
  \item \(G, H \in \mathcal{P}^{\ast}(\Omega_N \cup \Omega_T, \Omega_A)\), and
  \item \(h\) is a nonterminal node pairing of \(G\) and \(H\).
\end{enumerate}

(3.4.2) A terminal graph pair \(Z\) (over \(\Omega_N, \Omega_T, \Omega_A\)) is a graph pair \(Z = (G, H, h)\) 
such that all nodes in \(G\) and \(H\) have values in \(\Omega_T\). Since neither \(G\) nor \(H\) contain 
nonterminal nodes, the nonterminal pairing \(h\) is empty and \(Z\) can be written 
\(Z = (G, H)\).
DEFINITION 3.5. Let $Q = (\Omega_N, \Omega_T, \Omega_A, S, P)$ be a pair grammar and let $X = (G_x, H_x, h_x)$ and $Y = (G_y, H_y, h_y)$ be graph pairs over $\Omega_N, \Omega_T, \Omega_A$.

(3.5.1) $X \overset{Q}{\Rightarrow} Y$ (Y is directly derived from X according to Q) iff $\exists$ a rule $(\ell, r, h) \in P$, a nonterminal node $n$ in $G_x$, and a nonterminal node $m \in H_x$ such that

1. $h_x(n) = m$ (i.e., the non-terminal nodes are paired),
2. $G_x \Rightarrow G_y$ by applying rule $\ell$ to rewrite node $n$,
3. $H_x \Rightarrow H_y$ by applying rule $r$ to rewrite node $m$, and
4. $h_y = h \cup h_x - \{(n, m)\}$ (i.e., the node pairing defined by $h_y$ combines the pairing defined by $h_x$ and that defined by $h$ in the obvious manner).

(3.5.2) $X \overset{Q}{\Rightarrow} Y$ (Y is derived from X according to Q) iff $\exists$ a sequence $Z_1, Z_2, \ldots, Z_n$ of graph pairs over $\Omega_N, \Omega_T, \Omega_A$ such that $X = Z_1$, $Y = Z_n$, and $Z_i \Rightarrow Z_{i+1}$, $i = 1, 2, \ldots, n - 1$.

(3.5.3) The (pair) language $L_Q$ defined by Q is $L_Q = \{X \mid X$ is a terminal graph pair over $\Omega_N, \Omega_T, \Omega_A$ and $S_Q \Rightarrow X\}$, where $S_Q$ is the initial graph pair

$$S_Q = (\{(n_0), (n_0, S)\}, \{(m_0), (m_0, S)\}, \{(n_0, m_0)\}).$$

A few elementary results concerning pair grammars and pair languages may be readily derived. Assume Q is a pair grammar with pair language $L_Q$, left grammar $Q_L$, left language $L_L$, right grammar $Q_R$, and right language $L_R$.

**Theorem 3.6.** $L_Q \subseteq L_L \times L_R$.

**Proof.** If $(X, Y) \in L_Q$, then $S_Q \Rightarrow (X, Y)$. But $S_Q = (S_{Q_L}, S_{Q_R}, h)$, where $S_{Q_L}$ is the initial graph of $Q_L$ and $S_{Q_R}$ is the initial graph of $Q_R$, and each step in the derivation of $X$ from $S_{Q_L}$ proceeds using a rule of $Q_L$. Therefore $S_{Q_L} \Rightarrow X$ in $L_L$, and thus $X \in L_L$. Similarly $Y \in L_R$. //

Thus the language $L_Q$ defines a correspondence between elements of $L_L$ and elements of $L_R$.

**Theorem 3.7.** $L_L = \{X \mid \exists (X, Y) \in L_Q\}$, and $L_R = \{Y \mid \exists (X, Y) \in L_Q\}$.

**Proof.** By Theorem 3.6, $\{X \mid \exists (X, Y) \in L_Q\} \subseteq L_L$ and $\{Y \mid \exists (X, Y) \in L_Q\} \subseteq L_R$. Assume $X \in L_L$. Then $\exists$ a derivation $S_{Q_L} = Z_1 \Rightarrow Z_2 \Rightarrow \cdots \Rightarrow Z_n = X$ using a sequence
of rules $\ell_1, \ell_2, \ldots, \ell_{n-1}$ in $Q_L$. But for each $\ell_i$, there is a rule $p_i = (\ell_i, r_i, h_i)$ in $Q$ and the derivation

$$S_0 = (S_{Q_L}, S_{Q_R}, h) \Rightarrow (Z_2, U_2, h_2) \Rightarrow (Z_3, U_3, h_3) \Rightarrow \cdots \Rightarrow (Z_n = X, U_n, h_n = \varphi)$$

using the rules $p_1, p_2, \ldots, p_{n-1}$ is a valid derivation in $Q$.

Therefore $\exists (X, U_n) \in L_Q$ and thus $L_L \subseteq \{ X \mid \exists (X, Y) \in L_Q \}$.

Similarly $L_R = \{ Y \mid \exists (X, Y) \in L_Q \}$.

By Theorem 3.7, $L_Q$ defines, for each element of $L_L$, at least one corresponding element of $L_R$, and conversely. We would like to know when there is a unique corresponding element in $L_R$ for each element of $L_L$.

**Theorem 3.8.** If $Q_L$ is unambiguous and there exist no two distinct rules in $Q$, $(\ell, r_1, h_1)$ and $(\ell, r_2, h_2)$, with the same left rule $\ell$, then $L_Q : L_L \rightarrow L_R$.

**Proof.** By Theorem 3.7, if $G \in L_L$, $\exists X = (G, H) \in L_Q$.

Suppose $\exists Y = (G, K) \in L_Q$ also. Since $L_L$ is unambiguous, $\exists$ a unique leftmost derivation of $G$, $S_{Q_L} \Rightarrow Z_1 \Rightarrow Z_2 \Rightarrow \cdots \Rightarrow Z_n = G$ using rules $\ell_1, \ell_2, \ldots, \ell_n$ of $Q_L$.

But $Q$ contains the unique sequence of rules $(\ell_1, r_1, h_1), (\ell_2, r_2, h_2), \ldots, (\ell_n, r_n, h_n)$ and

$$S_{Q_R} \Rightarrow U_1 \Rightarrow U_2 \Rightarrow \cdots \Rightarrow U_n = H,$$

using $r_1, r_2, \ldots, r_n$ in $Q_R$. But since $(G, K)$ is also in $L_Q$, this must be the derivation of $K$ as well, and thus $H = K$.

Therefore $L_Q : L_L \rightarrow L_R$ and by Theorem 3.7, $L_Q$ is onto. ///

**Definition 3.9.** A pair grammar $Q$ is **unambiguous** iff both the left grammar and right grammar of $Q$ are unambiguous, and $Q$ contains no two distinct rules with identical left rules or identical right rules.

**Corollary 3.10.** If $Q$ is unambiguous, then $L_Q : L_L \rightarrow L_R$.

The pair grammar gives us a formal technique for defining translations. If the left language is a string language, then a string-to-graph translation is specified; if the right language is a string language, then a graph-to-string translation is defined; and if both are graph languages, then a graph-to-graph translation is defined. If the left grammar is unambiguous, then the translation defined from left to right language is unique, i.e., for any element of the left language there is a unique corresponding element of the right language. Moreover, if the right grammar is also unambiguous, then the translation is reversible; one can translate from left language to right language or from right language to left language equally.
Given an unambiguous pair grammar the right language element corresponding to a given left language element is easily determined. Suppose $Q$ is an unambiguous pair grammar whose left language is a set of strings and whose right language is a set of graphs. Given a string $X$ in the left language of $Q$, the corresponding graph $X'$ in the right language of $Q$ (the translation of $X$ into its graph representation) may be constructed as follows:

1. Use the left grammar of $Q$ (which is an ordinary context-free grammar) to parse $X$ in the usual manner, producing the derivation $S \Rightarrow Z_1 \Rightarrow Z_2 \Rightarrow \cdots \Rightarrow Z_n = X$.

2. Now generate the corresponding graph $X'$, using the rules of the right grammar of $Q$ and making the generation process deterministic by invoking the correspondence defined by the pair grammar. The generation begins with the initial single node graph $S$ and requires exactly $n$ steps. At the $k$-th step the pair grammar rule to be used is (uniquely) the rule whose left rule was used in the $k$-th step of the parse of $X$.

Thus we have a formal definition of a translation between two languages without moving outside the usual concepts of parsing and generation using grammars. In the special case where both languages are ordinary "string" languages, the translation is almost precisely the "syntax-directed transduction" of Lewis and Stearns [7]. In the most general case where both languages are graph languages, the translation may be viewed alternatively as a formal specification of a graph transformation.

**Example 2: Translating Algol-like Strings into the Flowchart Language**

As translation of strings into graphs is one of our primary interests here, we shall exemplify pair grammars by a pair grammar for translating a simple Algol-like language into the flowchart language of Example 1. Figure 4 contains a BNF grammar for a simple Algol-like language with an `if ... then ... else` branching statement and a loop statement of the form `while ... do ... end`. The pair grammar defining the translation from this language into the flowchart language (and vice versa) is defined as follows: Consider Fig. 1 as defining the right grammar of a pair grammar. Figure 4 then contains the corresponding left grammar. Since the left grammar defines a "string" language, we have used the ordinary BNF notation, omitting the node ovals and arcs, in place of the more cumbersome graph notation. For example, instead of the rule in Fig. 3, we write

$$\langle\text{program}\rangle ::= \langle\text{begin}\rangle\langle\text{compound stmt}\rangle\langle\text{end}\rangle$$

![Fig. 3. Graph form of an ordinary BNF grammar rule.](image)
Rule number | Rule
---|---
1 | \( \langle \text{program} \rangle \ ::= \text{begin} \langle \text{compound stmt} \rangle \text{end} \)
2 | \( \langle \text{compound stmt} \rangle \ ::= \langle \text{statement} \rangle \)
3 | \( \langle \text{compound stmt} \rangle \ ::= \langle \text{compound stmt} \rangle ; \langle \text{statement} \rangle \)
4 | \( \langle \text{statement} \rangle \ ::= \langle \text{assignment} \rangle \)
5 | \( \langle \text{statement} \rangle \ ::= \langle \text{loop} \rangle \)
6 | \( \langle \text{statement} \rangle \ ::= \langle \text{branch} \rangle \)
7 | \( \langle \text{loop} \rangle \ ::= \text{while} \langle \text{predicate} \rangle \text{do} \langle \text{compound stmt} \rangle \text{end} \)
8 | \( \langle \text{branch} \rangle \ ::= \text{if} \langle \text{predicate} \rangle \text{then} \langle \text{compound stmt} \rangle \text{else} \langle \text{compound stmt} \rangle \text{end} \)
9 | \( \langle \text{assignment} \rangle \ ::= S_1 | S_2 | \ldots | S_n \)
10 | \( \langle \text{predicate} \rangle \ ::= P_1 | P_2 | \ldots | P_m \)

**Fig. 4.** BNF grammar for an Algol-like language.

The pairing of rules and nonterminal nodes in rules is simply defined: (1) Corresponding rules have the same rule number. (2) The correspondence between nonterminal nodes in corresponding rules is obvious (since corresponding nodes must contain the same nonterminal), except in rule 8, where subscripts on the two occurrences of \( \langle \text{compound stmt} \rangle \) specify the correspondence.

Figure 5 gives the program string which corresponds to the flowchart of Fig. 2, according to the pair grammar. As it is apparent that both the grammars of Figs. 1 and 4 are unambiguous, the pair grammar specifies a unique flowchart for each program string and a unique program string for each flowchart. Thus a reversible program-string-to-flowchart translation is defined for this simple language.

\[
\begin{align*}
\text{begin } S_1; \\
\text{if } P_1 \text{ then } S_2; S_3; \text{while } P_2 \text{ do } S_5 \text{ end} \\
\text{else while } P_3 \text{ do while } P_4 \text{ do } S_6; S_7 \text{ end; } S_8 \text{ end end;} \\
S_9 \\
\text{end}
\end{align*}
\]

**Fig. 5.** The program corresponding to the flowchart of Fig. 2.

**IV. EXTENDED GRAPH GRAMMARS I: HIERARCHIES**

In this section and the next, two extensions to the basic graph and pair grammar concepts of Sections II and III are presented. The attempt to use pair grammars as
defined in the preceding sections to define translations of actual programming languages immediately brings out some important shortcomings of the concepts for practical use. One of the shortcomings is easily seen in the following example. Suppose we wish to translate simple replacement statements into assembly language flowcharts, for example:

\[ LDA \ Y \]

\[ X := Y \]

\[ STO \ X \]

**Figure 6**

The pair grammar rule of Fig. 7 would be a natural representation. However, the right rule graph is not allowed, for the value of a node must be a single terminal or non-terminal symbol. Here a string of terminals and nonterminals is needed as the value of a node. Moreover, in deriving the final terminal graph the nonterminals in the string need to be expanded into strings of terminals as in ordinary context-free grammar rewritings. Thus it is desirable to extend the definition of graph to allow graphs whose nodes have strings of terminals and nonterminals as values rather than single terminals or nonterminals.

The extension to allow nodes to have strings as values in a sense provides an extra level of structure in a graph. At the top level we still have a graph composed of nodes and labeled arcs, but now the values of the nodes are no longer "atomic" but themselves have structure, albeit a simple string structure. As strings are simply a special case of graphs, a restriction to strings in nodes would in fact be rather arbitrary. A more natural generalization is to allow any arbitrary graph as a node value. As the nodes in such a graph might also have graphs as values, this generalization immediately leads to structures composed of hierarchies of graphs. At the top level in the hierarchy is a single graph. Each node in this graph has a value which is either a terminal symbol or

\[ <\text{stmt}> ::= <\text{identifier}> := <\text{identifier}>_1 \]

\[ <\text{stmt}> ::= \]

\[ LDA <\text{identifier}>_2 \]

\[ STO <\text{identifier}>_1 \]

**Figure 7**
a graph, which we could call a second-level graph. Each second-level graph in turn contains nodes whose values are again either terminals or third-level graphs, etc. Ultimately the lowest level in the hierarchy contains graphs whose nodes have only terminal values.

Is such a generalization to hierarchical structures useful? While the simple extension to strings is of obvious utility, the more general hierarchical structures are also useful for the representation of subroutine and block structure hierarchies, data structure hierarchies, etc. The desirability of hierarchical representations of programs and data structures has been argued at length in [15, 16] and so we will not consider the question further here. Our concern is with the manner in which one may allow, as elements of graph languages, hierarchies of graphs (having nodes containing strings as a special case).

Although the extension of graph languages and grammars to such hierarchical structures appears a major generalization of Section II, it does not entail major changes in the formal development. We wish to simply allow grammar rules to rewrite nonterminal nodes as graphs whose nodes may contain (1) terminals and nonterminals as before, or additionally (2) other graphs whose nodes in turn may contain terminals, nonterminals, or graphs, etc. to any (finite) depth. Thus, for the example, we wish to allow the rule of Fig. 8 which contains three graphs organized into a two-level hierarchy. We shall term a hierarchically organized set of graphs over the same alphabets an H-graph, defined formally as follows:

**Definition (4.1)** \( \mathcal{H}^* = \Omega_M \).

(4.2) A level-1 H-graph over \( \Omega_M, \Omega_A \) is a graph over \( \Omega_M, \Omega_A \).

(4.3) \( \mathcal{H}^*(\Omega_M, \Omega_A) = \mathcal{F}(\Omega_M, \Omega_A) \) = set of all level-1 H-graphs.

(4.4) A level-k H-graph \( (k \geq 1) \) over \( \Omega_M, \Omega_A \) is a graph over \( U_{i=0}^{k-1} \mathcal{H}^*(\Omega_M, \Omega_A), \Omega_A \) with at least one node value in \( \mathcal{H}^* \).

(4.5) \( \mathcal{H}^k(\Omega_M, \Omega_A) = \{X \mid X \text{ is a level-k H-graph}\} \).

(4.6) \( \mathcal{H}^*(\Omega_M, \Omega_A) = \bigcup_{k=0}^{\infty} \mathcal{H}^k \) = the set of all H-graphs over \( \Omega_M, \Omega_A \).

---

Fig. 8. A hierarchical graph grammar rule.
Note that we assume throughout that the graphs in an $H$-graph are over disjoint node sets, so that the same node may not belong to two graphs. The $H$-graphs defined above are not entirely identical to the $H$-graphs of [16] as recursive hierarchies are not allowed.

To extend graph grammars to $H$-graph grammars, we may simply replace $G^*$ by $H^*$ throughout the definitions of Section II. Two $H$-graphs are equivalent if they are both level-$k$ $H$-graphs for some $k$ and they are equivalent as graphs over $\bigcup_{i=0}^{2^{-1}} H_i^*$, $\Omega_A$ considering two nodes as having equal values if the values are the same symbol in $H_i^*$ or equivalent graphs in $H_i^*$, $i > 0$. In a graph grammar the rules may have right hand sides which are arbitrary $H$-graphs. The input and output nodes, however, must be restricted to nodes in the top-level of the $H$-graph. Derivation of one $H$-graph from another is as before; a nonterminal node is replaced by an $H$-graph and the connecting arcs hooked up appropriately to the input and output nodes. The definition of ambiguity is unchanged.

In pair grammars the extension to $H$-graph pair grammars requires no modification beyond allowing the extended $H$-graph grammar rules in pair grammars. An unambiguous $H$-graph pair grammar defines a translation from one set of $H$-graphs to another.

**Example 3:** Translating Algol into Hierarchical Flowcharts

The translation of Algol into flowcharts provides an interesting application of this extension. Figure 11 gives an extended pair grammar for this translation. The extensions are used in two ways in this example:

1. Certain syntactic classes, such as assignment statements, will be translated without change. Thus if an assignment statement occurs in an Algol program, e.g.,

   \[
   \ldots \quad X := Y + 2 \ast Z \quad \ldots
   \]

   the flowchart graph contains the node

   ![Figure 9](image-url)
or more simply

\[ X := Y + 2 \times Z \]

**Figure 10**

We choose not to analyze in further detail the internal structure of assignment statements (although it could certainly be done). Similarly these other Algol syntactic classes will be translated without change:

- `〈array declaration〉`
- `〈type declaration〉`
- `〈procedure heading〉`
- `〈procedure statement〉`
- `〈Boolean expression〉`
- `〈arithmetic expression〉`
- `〈variable〉`.

See rules 9, 10, 12, 17–21 in Fig. 11 for examples. In these cases we are simply using the ability to generate strings in nodes. This greatly simplifies the example, as the Algol grammar rules defining the above syntactic categories may simply be appended to the grammar of Fig. 11 to complete the definition of these classes as follows:

Each Algol grammar rule `A ::= β` used in defining the above classes in the Algol report [1] is added to the pair grammar as the rule

1. (left rule) \( A ::= β \) (with the obvious nonterminal pairing)
2. (right rule) \( A ::= β \)

interpreting the string \( β \) as a graph in the manner of Section II.

(2) The more interesting use of the hierarchical form of graph grammars is seen in rules 11 and 15, where `〈block〉` and `〈procedure declaration〉` are expanded into graphs which are not simply strings. Since the occurrence of a `〈procedure declaration〉` or `〈block〉` adds a new level of structure, and since both `〈procedure declaration〉` and `〈block〉` are defined recursively, it is thus possible to generate a graph with a number of hierarchical levels corresponding to the depth of nesting of blocks or procedure declarations in the corresponding Algol program. The reason for
representing a \langle\text{block}\rangle as a separate hierarchical level while a \langle\text{compound statement}\rangle is not so represented is simply to delimit the scope of the declarations in the block head by the structure of the graph. A similar purpose is served by the hierarchical representation of a \langle\text{procedure declaration}\rangle.

A number of features of Algol are omitted in this example, e.g., switches, \texttt{go to} statements, labels, and various alternative forms of \texttt{for} and conditional statements. Detailed representation of some of these features as well as some of the syntactic classes translated intact such as procedure statements introduces complexities which cannot be handled by the extended pair grammars of this section. The next section introduces a further extension to allow labels and \texttt{go to} statements to be handled with pair grammars.

One further note on the example. In rule 21 in Fig. 11 the nonterminal node pairing of the left and right rules is not strictly 1—1, because the controlled variable of the \texttt{for} statement appears four times in the graph representation rather than once. This useful but minor extension has the effect of making the translation of strings into
graphs still unique but not onto, i.e., there are some graphs in the right language which occur in no pair of the pair language. Thus such graphs have no corresponding string representation in the left language. This case arises only in rule 21.

Figures 12 and 13 give an Algol program (a slightly modified form of a program from the Algol report [1]) and its corresponding graph in the pair language of the grammar of Fig. 11. Note that the identity statement $Y := Y$ in the last but one line of the program is required, because for simplicity we have restricted all conditional statements to be of the form if ... then ... else ..., so that an else clause, even though superfluous in this example, is required.
PAIR GRAMMARS AND GRAPH LANGUAGES

Rule Left rule Right rule
17 \langle \text{statement} \rangle ::= \langle \text{assignment stmt} \rangle \quad \text{statement} ::= \langle \text{assignment stmt} \rangle
18 \langle \text{statement} \rangle ::= \langle \text{procedure stmt} \rangle \quad \text{statement} ::= \langle \text{procedure stmt} \rangle
19 \langle \text{conditional stmt} \rangle ::= \text{if} \langle \text{Boolean expr} \rangle \text{ then } \langle \text{statement} \rangle \text{ else } \langle \text{statement} \rangle
20 \langle \text{for stmt} \rangle ::= \text{for} \langle \text{var} \rangle := \langle \text{arith. expr} \rangle \text{ while } \langle \text{Boolean expr} \rangle \text{ do } \langle \text{statement} \rangle
21 \langle \text{for stmt} \rangle ::= \text{for} \langle \text{var} \rangle := \langle \text{arith. expr} \rangle \text{ while } \langle \text{arith. expr} \rangle \text{ do } \langle \text{statement} \rangle

Fig. 11 (continued)
procedure Absmax (a, n, m, y, i, k);
array a; integer n, m, i, k; real y;
begin integer p, q;
y := 0;
for p := 1 step 1 until n do
for q := 1 step 1 until m do
if abs(a[p, q]) > y then
begin
y := abs(a[p, q]); i := p; k := q
end
else y := y
end

FIG. 12. Algol program corresponding to the graph of Fig. 13.

Fig. 13. Flowchart graph corresponding to program of Fig. 12.
V. Extended Graph Grammars II: Node Identification

The extensions of the preceding section allow graph grammars and languages which contain hierarchies of graphs. A second extension is needed to allow natural representation of constructs such as labels and go to statements. We first describe the general problem.

In translating program strings into graphs, many situations arise in which multiple occurrences of the same substring in the program would naturally be treated as references to the same node in the graph. For example, two occurrences of “go to A” in a program might naturally be translated into arcs in a graph leading to the node corresponding to statement A. Similarly in the statement A := A + 1 the two references to the variable A would normally be translated into references to the single data node corresponding to A. In the Lisp list (A(BA)C) again, the multiple occurrences of the atom A would naturally be translated as references to the single node corresponding to atom A in the graph representation.

We cannot make this sort of translation with pair grammars due to the “context-free” nature of the grammars involved. It is well-known that no context-free grammar can define exactly the language of statements of the form A := A + 1 where A is any legal Algol identifier. At best one may have a rule

\[
\langle \text{statement} \rangle ::= \langle \text{identifier}\rangle_1 := \langle \text{identifier}\rangle_2 + 1
\]

and an extra-grammatical test to determine if \(\langle \text{identifier}\rangle_1\) is identical to \(\langle \text{identifier}\rangle_2\). This sort of matching is commonly handled by an extra-syntactic “symbol table lookup” in a syntax-directed compiler. Although graph grammars as defined in Section II are more general than context-free string grammars, they are still inherently “context-free” and thus unable to handle cases like the above. Because the problem arises often in translating actual programming languages into graphs, an extension of pair grammars to handle these situations would be of great practical utility.

Restricting our attention initially to simple (nonhierarchical) graphs, a partial solution may be obtained as follows: (1) Extend graph grammars slightly so that each terminal node has a (terminal) “label” as well as a value. Leave the remainder of the formal development unchanged. A pair grammar then defines a translation (in the case of greatest interest) of strings into graphs with labeled nodes. (2) Define a “reduction rule” for graphs with labeled nodes that specifies (roughly) that nodes with identical labels may be reduced to a single node provided both nodes have identical or null values (see Definition 5.1). A graph with labeled nodes is in “reduced form” if all such reductions have been made. For graphs in reduced form ignore node labels in determining equivalence, so that two graphs are considered equal if they are isomorphic when node labels are deleted.

In this approach, the node labels serve roughly the same function as a “symbol table”, and the reduction rule is analogous to a “symbol table lookup” in a compiler,
except that the lookup is done only after translation is complete. As would be expected, the translation of strings into graphs in reduced form is a one-way translation, the original string cannot be uniquely recovered from the reduced form of the graph in general. Note that the approach proposed here does not adequately handle situations such as the nested block structure in Algol where "nested symbol tables" are required owing to the differing scopes of the definitions of variables.

Making the preceding concepts more precise we extend the definitions of Sections II and III as follows (ignoring temporarily the H-graphs of Section IV). Assume the terminal alphabet $\Omega_r$ of Sections II and III is now defined as $\Omega_r = \Omega_L \times \Omega_V$, where

$\Omega_L$ is a finite alphabet of terminal node labels (with a designated member, $\#L$, the null label), and

$\Omega_V$ is a finite alphabet of terminal node values (with a designated member, $\#V$, the null value).

Returning to the definitions of Sections II and III, let $\Omega_r = \Omega_L \times \Omega_V$ throughout. Thus the terminal alphabet is simply the set of pairs $(x, y)$ such that $x$ is a terminal node label and $y$ is a terminal node value. The remainder of the development of graph and pair grammars is left intact. Thus the graphs are as before, except each terminal node contains a pair $(x, y)$ of label and value symbols rather than a single terminal symbol. The following definition specifies how a graph may be reduced by combining two nodes with the same label.

**Definition 5.1.** If $G$ and $H$ are graphs in $\mathcal{G}(\Omega_L \times \Omega_V, \Omega_A)$ then $H$ is a reduction of $G$ iff there exist nodes $n$ and $n'$ in $N_G$ and a node $m$ in $N_H$ such that

1. $N_G - \{n, n'\} = N_H - \{m\};$
2. (a) $\forall p \in N_G$, $V_G(p) = V_H(p)$ if $p \notin \{n, n', m\};$
   (b) $\exists \ell \in \Omega_L, \ell \neq \#L$, and $v \in \Omega_V$ such that
      $V_G(n) = (\ell, v),$
      $V_G(n') = (\ell, v) \text{ or } (\ell, \#V),$
   and
      $V_H(m) = (\ell, v);$
3. (a) $\forall x, y \in N_G - \{n, n'\}$, $(x, a, y) \in E_G \iff (x, a, y) \in E_H,$
   (b) $(x, a, m) \in E_H \iff (x, a, n) \text{ or } (x, a, n') \in E_G,$
   (c) $(m, b, y) \in E_H \iff (n, b, y) \text{ or } (n', b, y) \in E_G,$
   (d) $(m, a, m) \in E_H \iff (n, a, n), (n', a, n'), (n, a, n') \text{ or } (n', a, n) \in E_G.$

Note that two nodes with identical nonnull labels may be reduced to a single node only if one has a null value or both have the same value.
Definition 5.2. A graph \( G \in \mathcal{G}^*(\Omega_L \times \Omega_V, \Omega_A) \) is in reduced form iff there exists no graph \( H \) such that \( H \) is a reduction of \( G \).

In the reduced form of a graph, all nodes with the same label have been coalesced unless there exist two nodes with the same label both with distinct nonnull values. As the node labels may be considered simply as a device for producing graphs in reduced form and subsequently are not significant, we consider two graphs in reduced form to be equivalent if they are equivalent (in the sense of Definition 2.4) when the node labels are ignored.

Definition 5.3 (Extended equivalence). If \( G \) and \( H \) are elements of \( \mathcal{G}^*(\Omega_L \times \Omega_V, \Omega_A) \), then \( G \equiv H \) iff \( \exists \) a function \( f : N_G \overset{1-1}{\rightarrow} N_H \) such that

1. \( \forall n \in N_G \), if \( V_G(n) = (\ell, \nu) \) and \( V_H(f(n)) = (\ell', \nu') \) then \( \nu = \nu' \) and
2. \( (n, a, m) \in E_G \) iff \( (f(n), a, f(m)) \in E_H \).

There is no difficulty in integrating the concepts of this section with the hierarchical graph extensions of the preceding section. As above, a terminal node is allowed to have both a label and a value. In generating a hierarchical graph, note that whenever a node is generated whose value is a graph, that node must be a terminal node (since a node containing a graph can never itself be replaced at any later rewriting step). Thus we extend the hierarchical graph grammar rules of the preceding section as follows: A grammar rule specifies a replacement of a nonterminal node by a graph in which each node may contain

1. a nonterminal as before, or
2. a pair \((\alpha, \beta)\), where
   a. \( \alpha \) is either a terminal label in \( \Omega_L \) or an \( H \)-graph in \( \mathcal{H}^*(\Omega_L \times \Omega_V, \Omega_A) \);
   b. \( \beta \) is either a terminal value in \( \Omega_V \) or an \( H \)-graph in \( \mathcal{H}^*(\Omega_L \times \Omega_V, \Omega_A) \).

In an \( H \)-graph with labeled nodes we restrict the reduction rule to apply only to nodes in the same graph within the hierarchy, i.e., two nodes \( n \) and \( n' \) with the same label may be coalesced into a single node only if the conditions of 5.1 are satisfied and in addition there is a graph \( G \) in the \( H \)-graph such that both \( n \) and \( n' \) are in \( N_G \).

Thus we do not combine nodes with the same label if they occur in different levels of the \( H \)-graph or in different graphs in the same level.

Example 4: Translating Algol with Labels and Go To Statements Into Hierarchical Flowcharts

Extending Example 3 to include labels and go to statements is straightforward. Rather than follow the Algol 60 grammar exactly, we shall illustrate the technique by
simply extending the grammar of Fig. 11 to allow any statement to be labeled. The additional rules are given in Fig. 14. In Figs. 14–19 a labeled node is represented by a rectangular box divided into two parts, the left part containing the node label and the right part containing the node value, as

<table>
<thead>
<tr>
<th>Rule no.</th>
<th>Left rule</th>
<th>Right rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td><code>&lt;statement&gt; ::= &lt;label&gt;:&lt;statement&gt;</code></td>
<td><img src="#" alt="Diagram" /></td>
</tr>
<tr>
<td>23</td>
<td><code>&lt;statement&gt; ::= &lt;go to statement&gt;</code></td>
<td><img src="#" alt="Diagram" /></td>
</tr>
<tr>
<td>24</td>
<td><code>&lt;go to statement&gt; ::= go to &lt;label&gt;</code></td>
<td><img src="#" alt="Diagram" /></td>
</tr>
</tbody>
</table>

**Fig. 14.** Additional rules for translating labels and `go to` statements in Algol.

Using these new rules the Algol sequence:

\[
L1 : S_1; \\
\vdots \\
S_n; \\
go to L1; \\
S_{n+1} \\
\vdots
\]

translates into the graph of Fig. 15 (before reduction). In reduced form this becomes the graph of Fig. 16.

**Example 5:** Translating Lisp S-expressions into Lists with Structured Atoms

Figure 17 gives a pair grammar for the translation of Lisp S-expressions in list notation into graphs representing a close approximation to the internal representation
of lists as described in the Lisp manual [10]. Atoms occurring in the $S$-expressions are translated into property lists, which have the header flag "—1" and the PNAME attribute whose value is the name of the atom. The reduction rule applied to such graphs produces reduced graphs in which multiple occurrences of the same atom in an $S$-expression are represented by the same property list in the graph. The left grammar of Fig. 17 actually defines a language of Lisp "packets", a packet being defined here as simply a sequence of $S$-expressions terminated by "STOP". A packet is translated into a graph representing a list of the graphs for the $S$-expressions, together with graphs for the "system defined" atoms NIL and PNAME which are used in the graph representation of all $S$-expressions. In Fig. 18 the graph representing the translation of the packet "(A(BA)C) STOP" is given as it is generated by the pair grammar and before the reduction rule is applied. In Fig. 19 the reduced form of the graph with node labels omitted is illustrated.

VI. GRAPH-TO-GRAPH TRANSLATIONS

In the preceding sections the examples have been entirely of string-to-graph translation, although the formal definitions have in fact defined pair grammars for
Fig. 17. Pair grammar for translating Lisp "packets" into list structure graphs.
PAIR GRAMMARS AND GRAPH LANGUAGES  

589

FIG. 18. Graph representation of "(A(B A)C) STOP" before reduction.

graph-to-graph translation, of which strings form a special case. To illustrate the more general case, a simple graph-to-graph translation is given in this section. The study of this general class of translations is of potentially great interest, but space restrictions preclude the further development here.

The example of Algol translation into flowcharts in Fig. 1 generates nodes with null values (#) in translating conditional statements into graphs. These null-valued nodes are superfluous unless the node is the terminal node in the graph. The flowchart graphs may be "cleaned up" by deleting these nodes and joining their incoming arcs to their outgoing arcs in the obvious manner, as in Fig. 20.

This simple graph transformation may be defined formally as a graph-to-graph
FIG. 19. Reduced form of graph of Fig. 18 with node labels deleted.

FIG. 20. A simple graph transformation.
	ranslation using pair grammars. An appropriate pair grammar is given in Fig. 21. The left language of the pair grammar is identical to the right language of the pair grammar of Fig. 1. The right language of the pair grammar of Fig. 21 is identical with
PAIR GRAMMARS AND GRAPH LANGUAGES

Rule No. | Left rule | Right rule
--- | --- | ---
1 | \( (\text{program}) \) ::\( = \) \( (\text{stmtlist}) \) | \( (\text{program}) \) ::\( = \) \( (\text{stmtlist}) \)
2 | \( (\text{program}) \) ::\( = \) \( (\text{predicate}) \) | \( (\text{program}) \) ::\( = \) \( (\text{predicate}) \)
3 | \( (\text{stmtlist}) \) ::\( = \) \( (\text{stmtlist}) \) | \( (\text{stmtlist}) \) ::\( = \) \( (\text{stmtlist}) \)
4 | \( (\text{stmtlist}) \) ::\( = \) \( (\text{statement}) \) | \( (\text{stmtlist}) \) ::\( = \) \( (\text{statement}) \)
5 | \( (\text{statement}) \) ::\( = \) \( (\text{predicate}) \) | \( (\text{statement}) \) ::\( = \) \( (\text{predicate}) \)

\[ \text{Fig. 21. Graph-to-graph pair grammar to delete null-valued nodes.} \]

the left language except that rule 6 deletes null-valued nodes from graphs in the right language. Note that the left grammar of Fig. 21 is considerably simpler than the right grammar in Fig. 1 (although they define the same language) because the node deletion transformation only requires limited analysis of the graph structure.

CONCLUSION

The major concern of this paper has been the problem of formal definition of translations between strings and directed graphs. We have shown that the “pair grammar” defined in Section III provides a simple and elegant means for defining such translations without moving outside the well-known concepts of parsing and generation using formal grammars. Moreover, by extending the graphs involved to allow labeled nodes and hierarchies of graphs as is done in Sections IV and V, the class of translations definable via pair grammars may be easily extended to include a number of practically important translations of programs into directed graph representations. We have tried to show, via a number of examples, that fairly simple pair grammars are powerful...
enough to define interesting translations from program strings into graphs. We have also considered, but only briefly, the use of pair grammars to define transformations on graphs.

The concept of pair grammar is a simple one which may readily be applied to pairings of grammars of many different types. We have chosen here a particular form of "context-free graph grammar" as the basic grammar form out of which pair grammars are constructed. It is clear that had we chosen ordinary context-free "string" grammars instead, we would have had pair grammars defining string-to-string translations, similar to the "syntax-directed transductions" of Lewis and Stearns [7]. Alternatively, a more general form of graph grammar might be used (for example, without our rather stringent restriction to a single input and a single output node in each grammar rule). This would allow definition of a larger class of translations with
(perhaps) a corresponding loss in the simplicity of the formal development. One can readily construct examples where the graph grammars of Section II are too restricted. The pair grammar concept may be applied to any type of grammar in which parsing and generation are defined in such a way that rules and nonterminals in rules can be properly paired. It is relatively independent of the particular form of underlying grammar which is used.

The importance of having a method for the simple formal definition of translations between strings and graphs lies in the widespread use of graphs for representation of structure, both in formal analysis and in computer processing. For example, in the formal definition of programming languages some form of directed graph is often the basic representation of programs for execution and formal analysis. The definition of the graph representation associated with each program string is then an important part of the definition of the language, and a pair grammar provides a possible means of defining this association in a manner which is itself formally analyzable. On the practical side, many large computer programs such as data management systems, natural language processors and compilers utilize some type of directed graph representation of data or programs for internal processing, and thus must translate input in string form into graph form before processing begins. Pair grammars provide a possible means for formally defining certain of these translations as well. A simple example of this occurs in LISP. LISP S-expressions form the string representation of the programs and data on input. However, these strings are not ordinarily interpreted directly. Instead, the usual LISP system first translates them into an internal list structure which is easily derived from the input string but is much simpler to process since it makes explicit the tree structure which was only implicit in the sequential ordering of the symbols in the original S-expression. We have shown in Section V that a fairly simple pair grammar may be used to define formally much of this translation of S-expressions into internal list structures.

Returning to the question of language-to-language translation in the standard sense where both languages are string languages, the pair grammar concept provides a possible new direction for the formal specification of string-to-string translations. While string-to-string translations may be formalized in the context of "syntax-directed transduction", we can argue that pair grammars may provide a method for string-to-string translation of greater potential utility as follows. In translating between actual languages such as programming languages, one usually is only interested in translations which preserve "meaning", so that the meaning of the output string in the object language is the "same as" the meaning of the input string in the source language. Thus if the translation is from Algol-to-Fortran, the translations of interest are those which for a given Algol program produce a Fortran program which represents the same computation. While the direct production of an equivalent Fortran program from an input Algol program in a single transduction step appears quite difficult at best, the following approach might be more successful. Suppose we translate (using
a pair grammar) the Algol program string into its equivalent graph representation, representing a flowchart of the Algol program. While one would expect the resultant graph to still be rather Algol-dependent (so that it is not directly also the flowchart of a Fortran program) the graph representation represents the “meaning” of the Algol program more explicitly than the original string. For example, the flow of control in the Algol program will be represented explicitly by paths in the graph where it was represented only implicitly in the string. Also, a lot of the syntactic “chaff” of the string representation will have been discarded. Thus the graph representation is at least more language-independent than the original string. Now it is necessary to translate the “Algol graph” into a “Fortran graph” before the Fortran string may be generated. Suppose we define a sequence of graph-to-graph translations (again using pair grammars) which progressively restructure the original graph until the desired Fortran graph is created. Each of these translations would involve reparsing the graph output from the preceding translation step and generation of a new graph. Because the graph structure represents the structure of the algorithm involved more directly than the original or final program strings, each translation step has greater power to restructure the algorithm without being constrained by details of syntax. And also because each translation step is formally defined, we are in a much better position to prove the “correctness” of the entire translation by showing that each translation step “preserves the computation”, so that the resultant program can be shown to represent the same computation as the original. Thus pair grammars might be used as a basis for a generalization of the usual translator writing system. Such a system would accept a number of pair grammars, each defining one step in a multistep translation of source language to object language.

In summary, pair grammars should be of interest both to those concerned with the theory of formal languages and to those concerned with the more practical aspects of language and language processor definition.

REFERENCES

6. P. LANDIN, Correspondence between ALGOL 60 and Church’s lambda notation, *Comm. ACM* 8, 2, 3 (1965), 89–101, 158–165.