EDGE-LABEL CONTROLLED GRAPH GRAMMARS*

Michael G. Main and Grzegorz Rozenberg

CU-CS-349-86 October 1986
Revised June 1987

* G. Rozenberg has been supported in part by National Science Foundation Grant MCS-83-05245. M. Main's research has been supported in part by National Science Foundation grant DCR-8402341.

EDGE-LABEL CONTROLLED GRAPH GRAMMARS*

Michael G. Main
Department of Computer Science
University of Colorado
Boulder, CO 80309 U.S.A.

Grzegorz Rozenberg
Institute of Applied Mathematics
and Computer Science
University of Leiden
Leiden, The NETHERLANDS

and

Department of Computer Science
University of Colorado
Boulder, CO 80309 U.S.A.

ABSTRACT

We introduce a graph-grammar model based on edge-replacement, where both the rewriting and the embedding mechanisms are controlled by edge labels. The general power of this model is established – it turns out to have the complete power of recursive enumerability (in a sense to be made precise in the paper). In order to understand where this power originates, we identify three basic features of the embedding mechanism and examine how restrictions on these features affect the generative power. In particular, by imposing restrictions on all three features simultaneously, we obtain a graph-grammar model that was previously introduced by Kreowski and Habel.

Keywords: edge-rewriting, label-control, NLC, recursive enumerability.

* G. Rozenberg has been supported in part by National Science Foundation Grant MCS-83-05245. M. Main has been supported in part by National Science Foundation Grant DCR-84-02341.

1. Introduction

In recent years, node-label controlled (NLC) graph grammars have been intensively studied as a method for generating node-labeled graphs [6,7,8,9,12]. The key feature of NLC-grammars is that both the rewriting of a subgraph, and the embedding of a newly introduced subgraph are controlled by node labels. (This is in contrast to the algebraic approach [1], where rewriting and embedding are based on structural properties of graphs.) Node-labeled graphs, which are the subject of NLC-grammars, are fundamental objects with numerous applications in computer science and other areas. Within the realm of graphs, one also has edge-labeled graphs, which are equally fundamental. It is natural to ask how the label-based control mechanism carries over to edge-labeled graphs. The aim of this paper is to initiate systematic research in this area.

The model of edge-label controlled (ELC) graph grammars we are presenting is influenced by our experience with NLC-grammars, and by the research of H.J. Kreowski on edge replacement systems [2,3]. However, we should warn that making a proposal for ELC grammars involves much more than simply taking the dual of NLC grammars.

The paper begins by presenting our basic ELC model and illustrating its features by examples. It turns out that our basic model is "too powerful" in the sense that it generates all the class of recursively enumerable graph languages. In order to formally state and prove this result, we have to formalize the notion of a recursively enumerable graph language. We do this by introducing a simple algorithmic language for constructing graphs.

In analyzing ELC grammars more closely, one can distinguish three basic parameters used in the rewriting process. Roughly speaking, the first parameter determines whether neighboring edges can be deleted when an edge is rewritten. The second parameter determines whether the source and target nodes of a rewritten edge can "merge" during the rewriting process. The third parameter controls whether multiple copies of the source and target nodes of a rewritten edge can appear. We investigate the impact of these three parameters. It is interesting to notice that if we make all three

parameters "restrictive", then we get the original edge-replacement systems proposed by Habel and Kreowski [2,3]. Other combinations of the parameters yield other classes of languages.

Terminology and notation: Throughout the paper, the term *graph* refers to a directed, edge-labeled, finite graph with at least one node (and with no self-loops). Multiple edges between the same pair of nodes are allowed. For a finite alphabet Δ, the set of all graphs with labels chosen from Δ is denoted G_Δ. Formally, such a graph is a tuple $(V, E, l, source, target)$, where V is a non-empty finite set of nodes, E is a non-empty finite set of edges, $l:E \rightarrow \Delta$ is a function assigning labels to edges, and $source, target: E \rightarrow V$ are functions assigning source and target nodes to edges. We do not distinguish between isomorphic graphs, so that the term "graph" actually refers to a class of isomorphic graphs.

The graph with exactly one node is denoted by $\bullet$. An edge labeled by A is called an A-edge. We say that an edge is *incident* to its source and target nodes, and two edges which are incident to a common node are called *adjacent*. Similarly, two nodes which are incident to a common edge are called *adjacent*. The set of nodes of a graph α is denoted by $nodes(\alpha)$ and the set of edges of α is denoted by $edges(\alpha)$. The cardinality of a set S is denoted $|S|$.

2. Edge-Label Controlled Graph Grammars

Definition 2.1. An *edge-label controlled graph grammar* (ELC *grammar*) is a 5-tuple $(\Sigma, \Delta, P, S, C)$, where

— Σ is a finite set of edge *labels.*

— Δ is a proper subset of Σ, called the *terminal* labels; elements of $\Sigma - \Delta$ are *nonterminals.*

— P is a finite set of *productions;* each production has the form

$$A \ := \ (\alpha, \alpha_{source}, \alpha_{target})$$

where A is a nonterminal label, $\alpha \in G_\Sigma$, and α_{source} and α_{target} are nonempty subsets of $nodes(\alpha)$.

— S is a nonterminal label called the *start label.*

— $C \subseteq (\Sigma \cup \{\textit{ISOLATED}\}) \times \Sigma$ is the *connection relation.*

□

The use of the connection relation and the meaning of *ISOLATED* — which is not in Σ — will be explained in a moment. An ELC grammar without a specified start label is called an *ELC grammar scheme*. The way that an ELC grammar generates a set of graphs (its *graph language*) is similar to the mechanism of node-label controlled (NLC) graph grammars. But before the description of this mechanism, we need a short discussion of the productions.

The graph α, on the right side of a production, is nonempty since G_Σ contains only nonempty graphs. These graphs are called *daughter graphs*, and intuitively a rule such as $A := (\alpha, \alpha_{source}, \alpha_{target})$ will be used to replace an A-edge by the subgraph α. In this replacement procedure, the labels of edges adjacent to various nodes in α will be important. For this reason, we assign a "type" to each node in a daughter graph, based on the labels of the node's adjacent edges. These types are not an intrinsic part of the graph (*i.e.*, the nodes remain unlabeled), but merely an

aid in describing the replacement procedure. The "type" of a node with no incident edges is an *ISOLATED*-node. A node with an incident A-edge is an A-node. A node with several incident edges may have several types; for example a node could be both an A-node and a B-node if it has both A and B edges incident to it. Throughout the replacement procedure, new edges will be added, connecting a daughter graph to the rest of the graph. But when we add these new edges, we will not change the "types" that we have identified with each node — *i.e.,* if we add an A-edge, this will not make new A-nodes.

Now we can describe the *ELC replacement procedure*, which consists of *rewriting* an edge with a production rule that introduces a new subgraph, and *embedding* the new subgraph by adding new edges. Let $G = (\Sigma, \Delta, P, S, C)$ be an ELC grammar. A production $A \; := \; (\alpha \;, \alpha_{source}, \alpha_{target})$ of P is used to transform a graph in the following way.

Part 1.

Start with a graph μ; within μ, choose any occurrence of an A-edge, which we will call the *mother edge*. Let s be the source node of the mother edge, and let t be the target node. All edges which are incident to s or t (except the mother edge) are called *neighborhood* edges. The mother edge, the neighborhood edges, plus s and t are now **deleted** from the graph μ, yielding a new graph μ'.

Part 2.

Add a copy of α (the daughter graph) to the graph μ' (where the edges and nodes of the copy of α are disjoint from μ').

Part 3.

Now we **introduce new edges between μ' and the daughter graph.** The introduction of these edges is controlled by the connection relation C, and by the "types" of the nodes in the daughter graph. The process consists of adding edges, according to one of the following rules:

(i) *Outgoing edges:* For each neighborhood edge $\underset{s}{\bullet} \xrightarrow{Z} \underset{u}{\bullet}$ (where $u \neq t$) and each node

$v \in \alpha_{source}$, if there is some Y such that v is a Y-node and $(Y,Z) \in C$, then connect v to u with a Z-edge: ($\underset{v}{\bullet}\xrightarrow{Z}\underset{u}{\bullet}$). Similarly, for each neighborhood edge $\underset{t}{\bullet}\xrightarrow{Z}\underset{u}{\bullet}$ (where $u \neq s$) and each node $v \in \alpha_{target}$, if there is some Y such that v is a Y-node and $(Y,Z) \in C$, then connect v to u with a Z-edge: ($\underset{v}{\bullet}\xrightarrow{Z}\underset{u}{\bullet}$).

(ii) *Incoming edges:* For each neighborhood edge $\underset{u}{\bullet}\xrightarrow{Z}\underset{s}{\bullet}$ (where $u \neq t$) and each node $v \in \alpha_{source}$, if there is some Y such that v is a Y-node and $(Y,Z) \in C$, then connect u to v with a Z-edge: ($\underset{u}{\bullet}\xrightarrow{Z}\underset{v}{\bullet}$). Similarly, for each neighborhood edge $\underset{u}{\bullet}\xrightarrow{Z}\underset{t}{\bullet}$ (where $u \neq s$) and each node $v \in \alpha_{target}$, if there is some Y such that v is a Y-node and $(Y,Z) \in C$, then connect v to u with a Z-edge: ($\underset{u}{\bullet}\xrightarrow{Z}\underset{v}{\bullet}$).

Part 4.

The final thing that remains to be done is to **establish some new internal edges in the daughter graph**; these edges correspond to edges that were originally parallel to the mother edge. Specifically, for each neighborhood edge $\underset{s}{\bullet}\xrightarrow{Z}\underset{t}{\bullet}$ and each pair of nodes $u \in \alpha_{source}$ and $v \in \alpha_{target}$, if there are two pairs $(Y,Z) \in C$ and $(Y',Z) \in C$ such that u is a Y-node and v is a Y'-node, then connect u to v with a Z-edge: ($\underset{u}{\bullet}\xrightarrow{Z}\underset{v}{\bullet}$). Similarly, for each neighborhood edge $\underset{t}{\bullet}\xrightarrow{Z}\underset{s}{\bullet}$ and each pair of nodes $u \in \alpha_{target}$ and $v \in \alpha_{source}$, if there are two pairs $(Y,Z) \in C$ and $(Y',Z) \in C$ such that u is a Y-node and v is a Y'-node, then connect u to

v with a Z-edge: ($\underset{u}{\bullet} \xrightarrow{Z} \underset{v}{\bullet}$). Since self-loops are not allowed, these connections are only done when $u \neq v$.

The final result has been to replace the mother edge with a copy of α. Edges were transferred from the mother's source and target to α_{source} and α_{target}. Let η be the graph resulting after Part 4. We write $\mu \underset{G}{\Rightarrow} \eta$ to denote the relation "η is *directly* derived from μ in G". If there exists a finite sequence of transformations:

$$\mu_0 \underset{G}{\Rightarrow} \mu_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} \mu_m$$

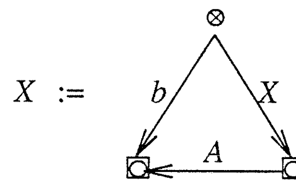
then we write $\mu_0 \overset{*}{\underset{G}{\Rightarrow}} \mu_m$ and say that μ_m is *derived from* μ_0 in G; the finite sequence is called a *derivation of length m*, and each application of the replacement procedure is one *derivation step*. The *language generated by the grammar G,* also called an ELC *language,* is the set of all graphs in G_Δ which can be derived from a graph with a single S-edge connecting two nodes; that is,

$$L(G) = \{\mu \in G_\Delta \mid \bullet \xrightarrow{S} \bullet \overset{*}{\underset{G}{\Rightarrow}} \mu\}.$$

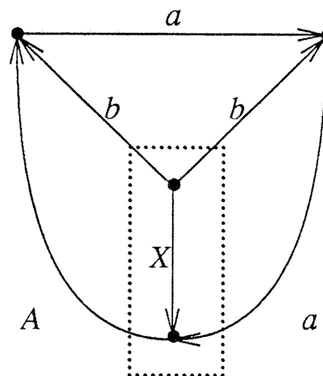
Any graph which can be derived from the start graph $\bullet \xrightarrow{S} \bullet$ is called a *graphical form* of G. Whenever the grammar G is understood from the context, the notation will be simplified to $\Rightarrow$ and $\overset{*}{\Rightarrow}$.

Examples. We present several examples of productions and their application in a replacement procedure. The nodes of a daughter graph α will be represented as follows: ○ is a node which is in neither α_{source} nor α_{target}; ⊗ is a node which is in only α_{source}; ◘ is a node which is in only α_{target}; and ⊠ is a node which is in both α_{source} and α_{target}.

Example 2.1. Consider a grammar with the connection relation $\{(b,A),(X,a),(X,b)\}$ and the following production (among others):



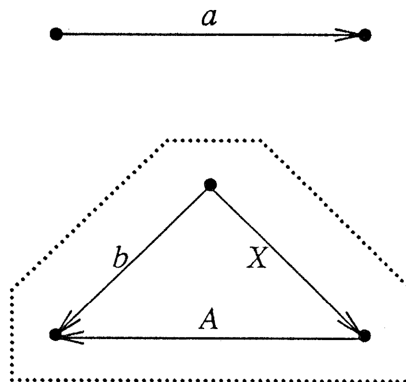
Let α be the right side of the above production. Here is an example of how the production is used in a replacement procedure. The example begins with the following four-node graph – the mother edge for this example is indicated by the dotted box.



In Part 1 of the replacement procedure, the mother edge is removed, along with its source, target and

neighborhood edges. In this case there are four neighborhood edges (the two curved edges which are adjacent to the mother target and the two b-edges which are adjacent to the mother source). When these are removed, only a single edge remains, from the top of the previous drawing:

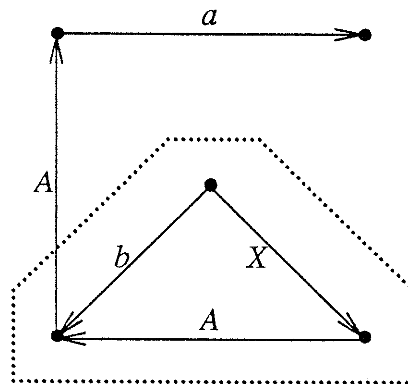
In Part 2 of the replacement procedure, a disjoint copy of the daughter graph is added, yielding this disconnected graph (the dotted line indicates the new daughter graph):



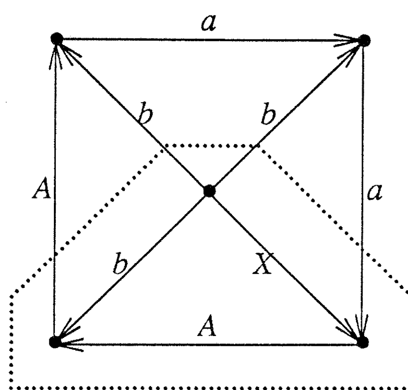
Also in Part 2 of the replacement procedure, we assign "types" to the nodes of the daughter graph, according to the labels of the adjacent edges. In this case, the upper node of the daughter graph is an X-node and a b-node; the lower left node is a b-node and an A-node; and the lower right node is an A-node and an X-node.

In Part 3 of the replacement procedure, edges are established between the daughter graph and the rest of the graph. These edges are established by examining the connection relation, and by examining each of the neighborhood edges that were deleted in Part 1. For example, the pair (b, A) is

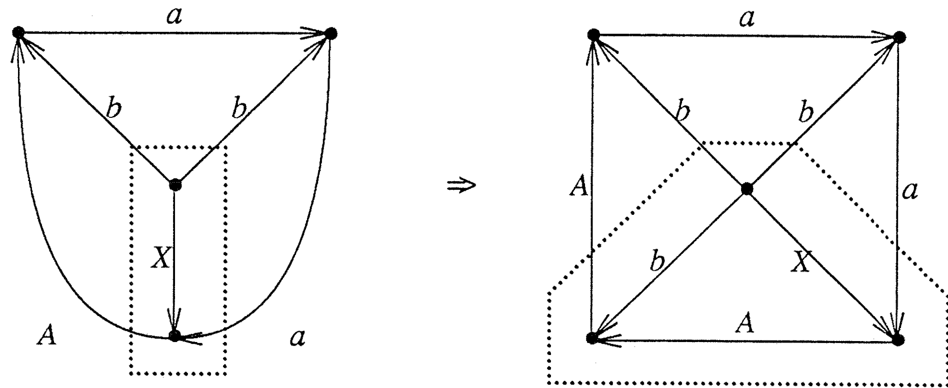
in the connection relation, and there is a neighborhood A-edge from the mother target to the upper left node of the original graph. This implies that a new A-edge is established from every type b-node in α_{target} to the upper left node of the original graph. This gives one new edge:



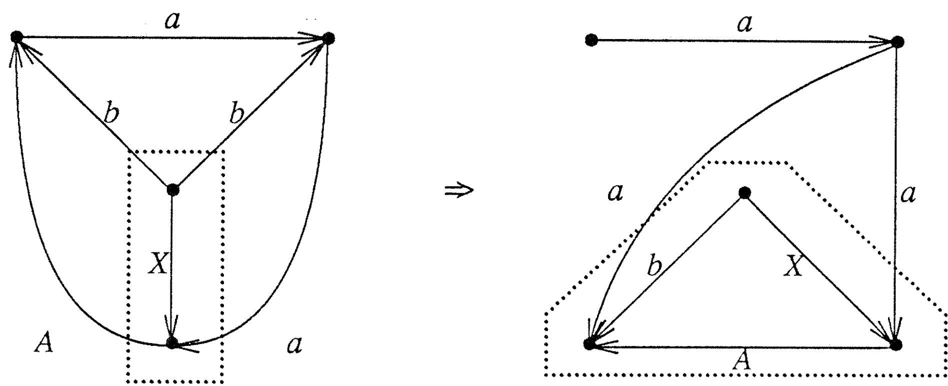
Notice that there is no A-edge from the second node of α_{target} to the upper left node of the graph, since this second node is not a type b-node. In all, Part 3 will add four edges between the daughter graph and the rest of the graph. At the end of Part 3, the graph looks like this:



In this case, Part 4 does nothing (since there were no edges parallel to the mother edge), so the entire replacement procedure looks like this:

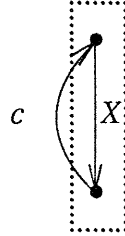


Example 2.2. Consider the same production as the previous example, but suppose that the connection relation contains only the pair (A, a). The production can be applied to the same graph as before, but the result is different. In particular, the daughter graph is reconnected to the rest of the graph in a different way. The pair (A, a) is in the connection relation, and there is a neighborhood a-edge from the upper right node of the original graph to the mother target node. Therefore a new a-edge is established from the upper right node of the original graph to every type A-node in α_{target}. The complete replacement procedure is drawn below, with the mother edge and daughter graph outlined.

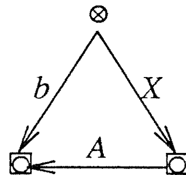


Example 2.3. This example shows how an edge which is parallel to the mother edge results in edges that connect source and target nodes of the daughter graph. These edges appear in Part 4 of

the replacement procedure. For this example, we start with the following two node graph:

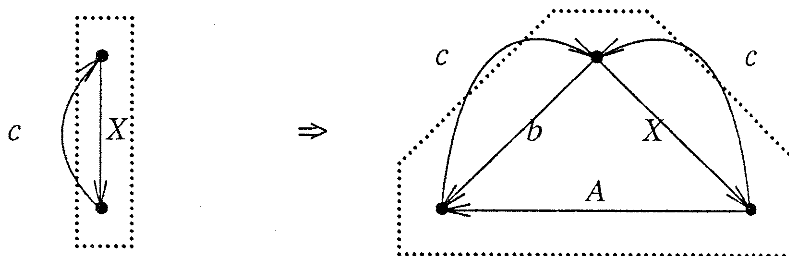


We will rewrite the X-edge from this graph, using the production from Example 2.1, and with the connection relation containing only (A, c) and (X, c). In Part 1 of the derivation, the X-edge is removed, along with its source and target nodes and the neighborhood c-edge. This leaves the empty graph, to which we add the daughter graph in Part 2 of the replacement procedure. So, at this point the graph consists of the three nodes of the daughter graph – two of which are in α_{target}, and one is in α_{source}:



Part 3 of this replacement procedure adds no new edges (since there were no neighborhood edges between the daughter graph and the rest of the graph). However, Part 4 adds two new edges from the two nodes of α_{target} to the one node of α_{source}. Both these edges arise from the c-edge which originally connected the mother target to the mother source. Since (X, c) is in the connection relation, we connect the lower right node of α_{target} (an X-node) to the one node of α_{source} (an X-node).

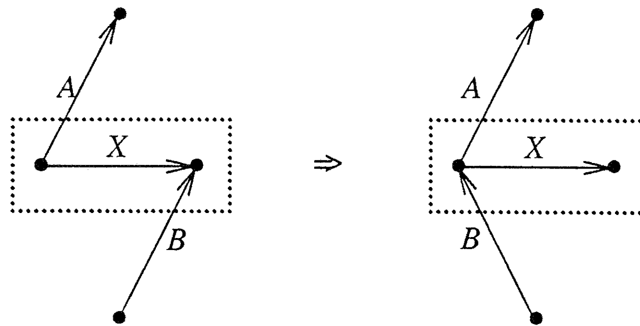
Since (A, c) is also in the connection relation, we also connect the lower left node of α_{target} (an A-node) to the one node of α_{source} (an X-node). In general, with this connection relation and the indicated parallel c-edge, we add a new c-edge from each A-node or X-node in α_{target} to each A-node or X-node in α_{source}. The complete replacement procedure is drawn here:



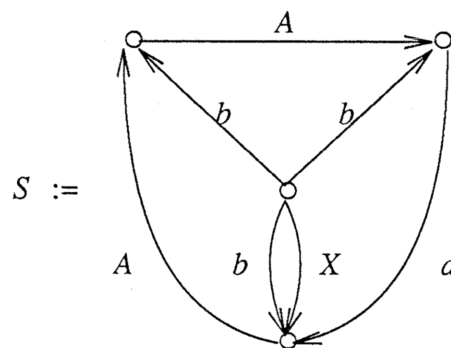
Example 2.4. This is an example of a production rule with a node in the daughter graph (α) which is in both α_{source} and α_{target}. The connection relation for this example contains all possible pairs, and the production rule is:

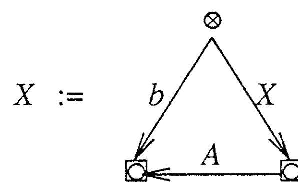
X :=

When this production is applied to an X-edge, a new X-edge is introduced, and all connections to the old X-target node are transferred to the new X-source node. Here is an example replacement procedure with the mother edge and daughter graph outlined:

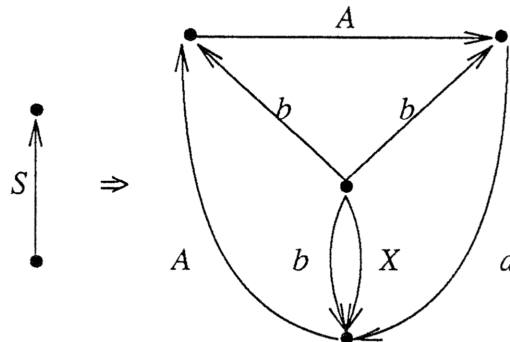


Example 2.5. This is an example of a complete grammar. The nonterminal labels are $\{S, X, A\}$, with start label S. There are two terminal labels $\{a, b\}$. The connection relation is $\{(a,a), (a,A), (a,b), (b,A), (X,a), (X,b)\}$. There are these three productions: (The source and target nodes for the daughter graph of the first production are not indicated because they are irrelevant; this production is only applied as the first step of a derivation, where there are no neighborhood or parallel edges.)

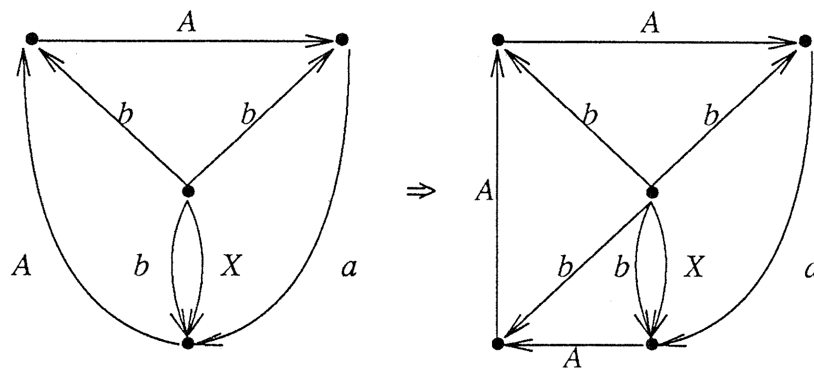




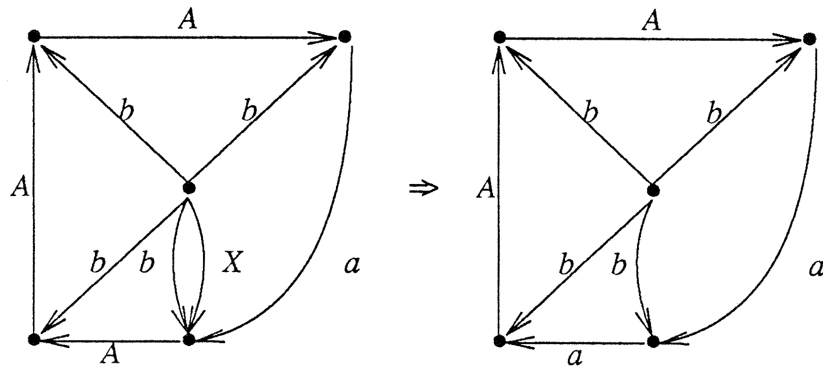
A derivation in this grammar begins by replacing the start label with the first production rule:



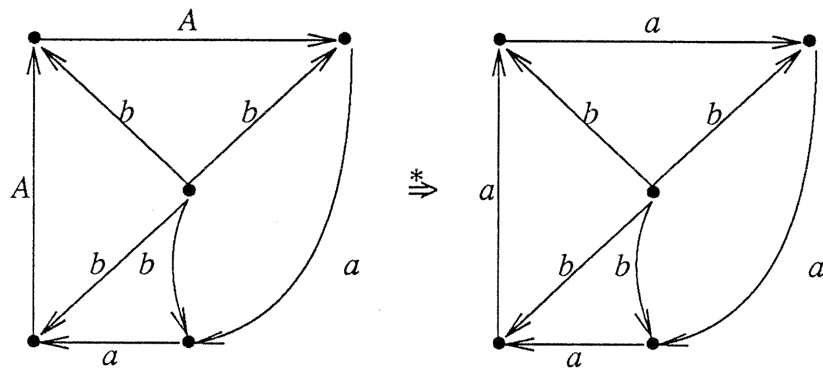
At this point in the replacement procedure, the graph has precisely one X-edge, and two A-edges. The X-edge may be replaced using the second production rule, which will keep the total number of X-edges at one and add one new A-edge in a particular way. For example, applying the second production rule to the above graph results in this replacement procedure:



The third production rule changes an A-edge to an a-edge. Because of the connection relation, an application of the third production also causes any X-edge adjacent to the mother source or target to "disappear". This is because (a, X) is not in the connection relation. For example, here is a replacement procedure using the third production rule:



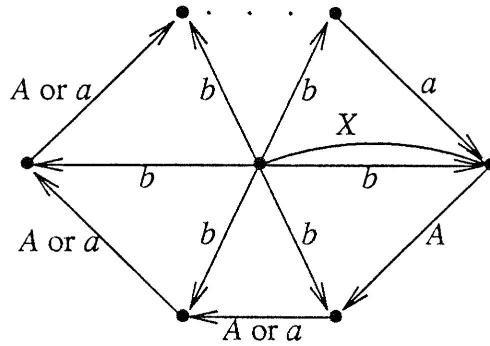
This particular derivation can be completed by applying the third production rule to the remaining two A-edges. This gives the final two steps of the derivation like this:



More generally, a derivation in this example grammar has these three stages:

1. The first derivation step where the start-edge is replaced using the first production rule.

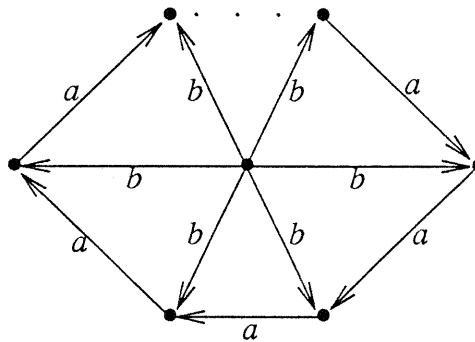
2. Applications of the second and third production rules, without replacing the A-edge adjacent to the X-edge. At each point in this stage, the graph has the form:



The edges around the "rim" are all labeled by either A or a. The only restrictions are that the edge which is immediately counter-clockwise from X is an a-edge, and the edge which is immediately clockwise from X is an A-edge.

3. Eventually, in any derivation, the A-edge which is adjacent to the X-edge will be rewritten to an a. At this point, the X-edge disappears, since (a, X) is not in the connection relation.

4. The remainder of the derivation replaces any remaining A-edges with a-edges, using the third production rule. The end result is a graph of this form:



From this, it is easy to see that the grammar generates all graphs of the above form, with at least three a's. In Section 8, we will show that this graph language cannot be generated

without a connection relation which breaks some edges (such as the way the X-edge is broken by replacing the adjacent A-edge).

Example 2.6. This is an example of a grammar which generates only discrete graphs. (There are no terminal labels.) The source and target sets for the productions of this grammar are always distinct single nodes of a daughter graph. Despite this simple form for productions, the grammar is interesting because the only graphs it generates have exactly $3 \times (8^n)$ nodes. This "exponential growth" in graph size will be used in Section 8.

The start label for the grammar is S and the other nonterminal labels are $\{A_0, A_1, A_2, X, B\}$. The connection relation contains these pairs:

(A_i, A_i) for $i = 0, 1$ or 2

(A_{i+1}, A_i) for $i = 0, 1$ or 2; addition is modulo 3

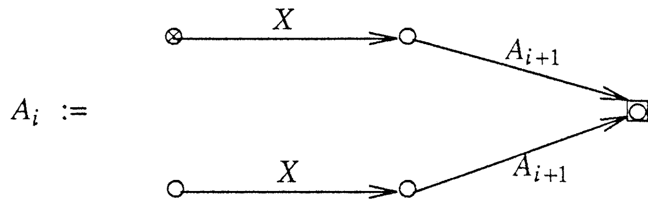
(B, A_2)

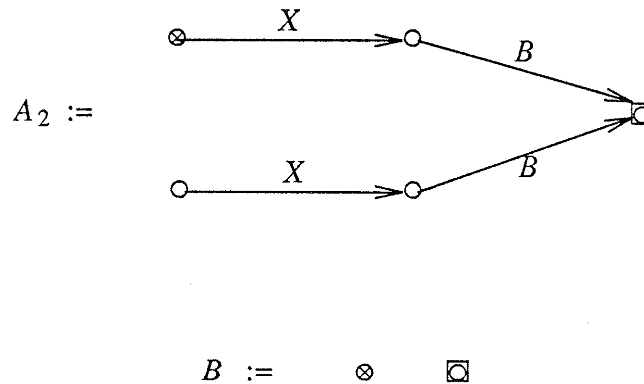
(ISOLATED,B).

Here are the productions of the grammar. As in the previous example, the source and target nodes of the first production are omitted since this production is only used at the start of a derivation.

S :=

A_i :=





Since there are no terminal labels, the grammar generates only discrete graphs. As mentioned above, the sizes of these discrete graphs are growing exponentially. To prove this, let $S \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_m$ be a derivation of a discrete graph α_m. We prove four properties which hold for any of the graphical forms α_k:

1. Every non-X-edge in α_k is adjacent to every other non-X-edge.
2. Every non-X-edge in α_k is adjacent to one X-edge, and this X-edge is not adjacent to any other edge.
3. Let $A_0^{\#}(\alpha_k)$ be the number of A_0 edges in α_k, and similarly for $A_1^{\#}$, $A_2^{\#}$ and $B^{\#}$. If α_k contains at least one A_i-edge, then

$$|\,nodes(\alpha_k)| \;=\; 3 \times (A_0^{\#}(\alpha_k) + A_1^{\#}(\alpha_k) + A_2^{\#}(\alpha_k) + B^{\#}(\alpha_k)).$$

4. Define the *value* of α_k to be the following integer:

$$\text{If } A_0^{\#}(\alpha_k) = 0 \text{ then } value(\alpha_k) = 9A_1^{\#}(\alpha_k) + 3A_2^{\#}(\alpha_k) + |\,nodes(\alpha_k)|$$

$$\text{else } value(\alpha_k) = 21A_0^{\#}(\alpha_k) + 9A_1^{\#}(\alpha_k) + 45A_2^{\#}(\alpha_k) + |\,nodes(\alpha_k)|\,.$$

   Then there exists an integer n such that $value(\alpha_k) = 3 \times (8^n)$.

The proof of these properties is by induction on k. The base step (k=1) is easy since α_1 is the fixed graph $\bullet \xrightarrow{X} \bullet \xrightarrow{A_0} \bullet$, and properties 1-4 are easily verified for this graph. The induction step

is a case analysis, based on the five possible productions which can be applied at $\alpha_{k-1} \Rightarrow \alpha_k$.

The fourth property implies that every graph generated by this grammar has $3 \times (8^n)$ nodes for some n. In fact, any such graph ($n \geq 1$) can be generated by this grammar.

3. Explicit ELC Grammars

In order to use an ELC production, we associate "types" with each node of the daughter graph. These types are the labels of the edges which are incident to the node, and these types are used (in Parts 3 and 4) to determine which new connections are made between this node and the rest of the graph. Thus, the labels in daughter graphs serve two purposes: they label the edges, and they also control the "types" that affect the new connections.

It may sometimes be convenient to separate these two roles of the edge labels. As an alternative to this method of associating types, we could explicitly assign types to the nodes of each daughter graph – without regard to the labels of incident edges. In particular, we could assign a set of edge labels $L(u)$ to each node u of each daughter graph. If $L(u)$ is the empty set, then u is to be treated as if it is an *ISOLATED*-node. Otherwise it will be treated as an A-node for each A in $L(u)$. This explicit assigning of types to daughter graph nodes permits a simplier design of grammars in many cases. We call these grammars *explicit* ELC grammars (since each node of each daughter graph has an explicit set of labels assigned to it).

Can explicit ELC grammars generate graph languages which cannot be generated by any ELC grammar? No! The following theorem shows how an explicit ELC grammar can be converted to an ordinary ELC grammar, without affecting the language that is generated.

Theorem 3.1. *The class of languages generated by explicit ELC grammars is precisely the class of ELC languages.*

Proof: Clearly every ELC language can be generated by an explicit ELC grammar. For the other direction, let $G = (\Sigma,\Delta,P,S,C)$ be an explicit ELC grammar. We will construct an ELC grammar G' which generates the same language as G. The label set of G' is $\Sigma \cup P(\Sigma)$, where $P(\Sigma)$ is the power-set of Σ. The terminals are still Δ and the start symbol is still S. The productions of G' are the same as those in G, but each daughter graph is modified. For each node u in a daughter graph, we add one extra node m_u, and there is an edge labeled by $L(u)$ from m_u to u. The *source* and *target* subsets of the daughter graph remain unchanged. For each new nonterminal label $X \in P(\Sigma)$, there is also one new production: X := ⊠. Finally, the connection relation of G' is the union of these five sets:

$$\{(X,A) \mid X \in P(\Sigma), \text{ and for some } B \in X : (B,A) \in C\}$$

$$\{(\varnothing,A) \mid (\textit{ISOLATED},A) \in C\}$$

$$\{(\textit{ISOLATED},A) \mid A \in \Sigma\}$$

$$\{(X,Y) \mid X,Y \in P(\Sigma)\}$$

$$\{(\textit{ISOLATED},X) \mid X \in P(\Sigma)\}$$

Each production of a derivation in G can be simulated in G' by a sequence of productions: the first production corresponds to the step from G, but with the extra nodes m_u, and the rest of the productions in the sequence apply the new productions to get rid of the m_u nodes, by collapsing them into other nodes. In this way, each graph generated by G is also generated by G'. It is straightforward to show that these are the only graphs G' generates. □

The explicit form is useful for showing that specific languages are ELC. In particular, we will use it to show that ELC languages have the complete power of recursive enumerability. But first we must define R.E. graph languages, which is the subject of the next section.

4. Recursively Enumerable Graph Languages

This section defines recursively enumerable graph languages over a fixed terminal alphabet Δ. Intuitively these are languages whose elements can be effectively enumerated.

4.1 Linear Descriptions of Graphs

We start with a small "algorithmic language" which can be used to construct graphs, using a simple set of primitive instructions. The instructions allow operations like "*add a new node*" or "*connect two specific nodes with a Y-edge.*" The language is designed with one goal in mind: sequences of instructions should be easy to "simulate" by a fixed ELC grammar. Because of this goal, the collection of allowable instructions may not be the most natural possible, but the set is complete enough to construct any graph (with 2 or more nodes).

Each sequence of instructions can be thought of as a little algorithm to dynamically generate a graph. At the beginning (before any instructions are executed), the "dynamic graph" has exactly two nodes, labeled 1 and 2, and no edges. Each instruction adds some edges, or nodes, or does some other alteration to the graph. At each step we assume that the nodes are labeled by consecutive integers starting at 1. Also, we keep track of one distinguished node number in a variable called *current* (initially *current*=1). As will be shown, instructions can change the value of *current*, and also renumber the nodes. We begin with a list of the different instructions, and then specify how they can be put together to form "graph programs".

LEFT and RIGHT:

The LEFT instruction subtracts one from the value of *current*. The RIGHT instruction adds one to the value of *current*.

EDGE_Y (Y is any label):

This connects node number *current* to node number *current*+1, using a Y-edge.

SWAP:

This instruction swaps the node numbers of node number *current* with node number *current*+1. The value of *current* remains unchanged.

ADD:

This instruction adds a new node. The number of the new node is *current*+1. Any node with a number higher than *current* has its node number incremented by 1. The value of *current* remains unchanged.

JOIN:

This instruction causes two nodes to merge. In particular, the nodes numbered *current* and *current*+1 are joined together. Any edges between the two nodes disappear. Any other node with a number higher than *current*+1 has its node number decremented by 1.

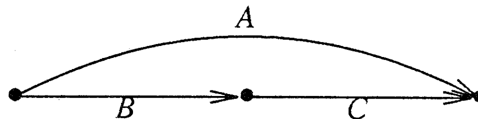
SKIP, IGNORE and END:

The SKIP instruction does nothing. The IGNORE instruction causes the next instruction to be ignored. END is a special instruction which must appear twice at the end of each sequence. It has no effect on the construction of the graph. The "operational" reason for these instructions will become apparent later.

These instructions can be put together sequentially to form "graph programs". We require these programs to meet certain restrictions. In particular, RIGHT and EDGE_Y can only be given if *current* is less than the number of nodes in the dynamic graph. A LEFT instruction requires *current*>1, and the IGNORE instruction cannot be the final instruction in the algorithm, nor can it precede another IGNORE instruction. Also, we require that the total number of nodes in the graph being constructed is never less than 2 — hence JOIN cannot appear unless there are at least three nodes in the graph. For reasons to be explained later, we also require that the SWAP and JOIN instructions appear only when *current* is less than $n-1$ (where n is the number of nodes in the dynamic graph). Finally, the END instruction must twice appear at the end of the sequence, and nowhere else. When these END instructions occur, the value of *current* must be 1.

A *graph program* is any sequence of instructions which meet these rules.

Example. Here is an example graph program: CONNECT_A, ADD, CONNECT_B, RIGHT, CONNECT_C, LEFT, END, END. This program constructs the following three node graph:



4.2 Formal Definition of R.E. Graph Languages

We are now ready to give the definition of a recursively enumerable graph language. For any graph α, there may be many graph programs which construct α. Among these we need to fix a shortest program (fewest instructions) which constructs α. The way we choose this fixed program is not important, so long as it is effective. For example, we may choose the lexicographically first program. The chosen program will be denoted by **program**(α). The one-node graph doesn't have a graph program, so we will define **program(•)** = ONE (where ONE is a new symbol).

For a graph language L over an alphabet Δ, we can define a string language over the alphabet $\{\text{LEFT}, \text{RIGHT}, \text{SWAP}, \text{ADD}, \text{END}, \text{ONE}\} \cup \{\text{EDGE}_Y \mid Y \in \Delta\}$ by

$$\mathbf{program}(L) = \{\mathbf{program}(\mu) \mid \mu \in L\}.$$

Note that the alphabet of **program**(L) does not contain IGNORE, SKIP or JOIN, since these will never appear in a shortest program for some graph.

Definition 4.1. A graph language L is called *recursively enumerable* iff **program**(L) is a recursively enumerable string language.

□

The next section shows that the ELC languages are precisely the recursively enumerable graph languages.

5. The Power of ELC Grammars

This section demonstrates that ELC grammars generate precisely the recursively enumerable graph languages. The proof uses a technique similar to an approach taken for "handle NLC" grammars in [11]. These are graph grammars which generate node-labeled graphs, but where the item replaced in a derivation step consists of an edge plus the edge's source and target node. (These three things form a "handle"). The proof of the result begins with an ELC grammar scheme S_Δ which can "simulate" graph programs.

5.1 Executing Graph Programs

This section gives an ELC grammar scheme S_Δ which "simulates" the graph programs of Section 4. The simulation occurs by starting with a graph which corresponds to some arbitrary graph program $I_1, I_2, \cdots, I_n$. From this starting point, the only terminal graph which S_Δ can derive is the graph which the program $I_1, I_2, \cdots, I_n$ constructs. This is obtained by having the productions of S_Δ execute the instructions of the graph program in a particular way.

The terminal labels for S_Δ are the labels of Δ. The non-terminals are the instructions (LEFT, RIGHT, EDGE$_Y$, SWAP, ADD, IGNORE, SKIP, JOIN and END) plus five extra symbols (X, A, B, G and T). The connection relation is $\Sigma \times \Sigma$, where Σ is the set of all labels. The productions will be given after a short discussion.

Suppose we have a graph program $I_1, I_2, \cdots, I_n$, which encodes a graph μ. When the graph program is executed, we can keep track of the graph as it is being constructed, and also keep track of the value of *current* at each step. Specifically, let μ_i be the graph that exists after executing i instructions of the graph program (together with the information about how its nodes are numbered). Also, let *current*$_i$ be the value of *current* at this point. Thus, μ_0 is the beginning graph, with two nodes (numbered 1 and 2) and no edges, while μ_n is the final graph μ.

Now define some auxiliary graphs $\eta_0, \eta_1, \cdots, \eta_{n-2}$. The graph η_i contains enough information to specify μ_i, the ordering on μ_i's nodes, the value of *current*$_i$ and the list of remaining instructions $(I_{i+1}, \cdots, I_n)$. In particular, we define η_i by adding some new nodes and edges to μ_i. The graph η_i is shown in Figure 5.1.

The line of uncircled nodes at the bottom of the figure are the nodes of μ_i. They are connected in order by a sequence of X-edges, so that node number 1 is on the left and the highest numbered node is on the far right. The edges drawn below these nodes in the figure are the actual edges of μ_i. Above each node of μ_i is a "stick" of extra nodes. All but one of these sticks is a chain of two A-edges. The other stick is a chain of the remaining instructions $I_{i+1}, \cdots I_n$. The node number of the node which has the special "instruction stick" is also the value of *current*$_i$.

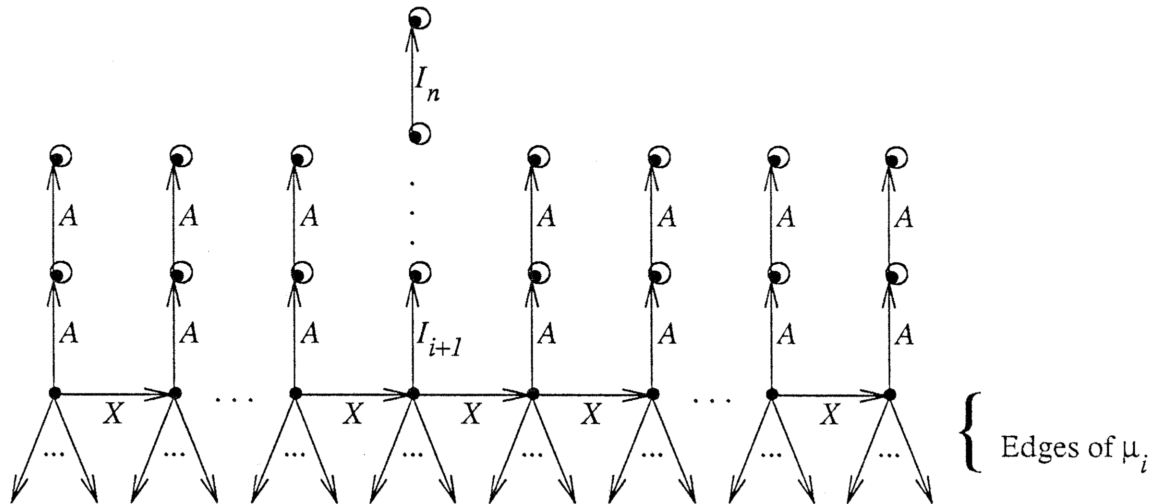


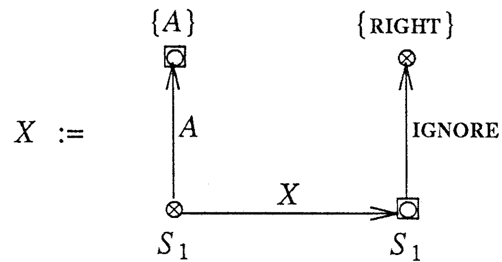
FIG. 5.1. The graph η_i. The uncircled nodes are the nodes of μ_i.

The grammar is constructed so that only one possible production can be applied to a graph like η_i. This one production is completely determined by the next instruction in the "instruction stick". For example, if the instruction I_{i+1} is RIGHT, then there is a production which corresponds to the RIGHT instruction which can be applied to η_i. The effect of applying this production is to move the instruction stick right one node. In this way, η_i is transformed into η_{i+1}. Finally, when η_{n-2} is reached, there are only two instructions (END, END) remaining. At this point, some extra productions must be applied to η_{n-2} to get rid of the extra nodes and edges. The final result will be the graph $\mu_n = \mu$.

One more item should be mentioned about the η_i graphs. The nodes which have attached "*A*-sticks" (*i.e.*, those which are not $current_i$) may have a more complicated structure attached. This structure will be a graph of A and B edges. We require that this attached graph has at least one edge which is not connected to the uncircled node from μ_i, and that no B-edges are adjacent to this uncircled node. So, η_i can actually be a bit more complicated than Figure 5.1.

Figures 5.2 through 5.12 show the productions which correspond to each of the instructions, together with an example of how these instructions would transform η_i into η_{i+1}. Some additional comments on the productions are given after the figures. The productions are given in the explicit ELC form. Recall that in the explicit form, each node u of a daughter graph has an explicit set $L(u)$ of edge-labels which determines when it is an A-node. In the figures, we will write $L(u)$ next to each node u of a daughter graph. S_1 denotes $\Delta \cup \{X\}$; S_2 denotes $\Delta \cup \{X, A\}$, and S_3 denotes the set of instructions (LEFT, RIGHT, EDGE$_Y$, SWAP, ADD, IGNORE, SKIP, JOIN and END). We will use the notation from Section 2 to denote the *source* and *target* subsets (○,⊗,▢,⊠).

Production P1.



Example derivation step: The edges C and D in this example are from the graph which is being constructed. The $\cdots$ indicates the instruction stick. The mother edge and daughter graph are indicated by dotted boxes.

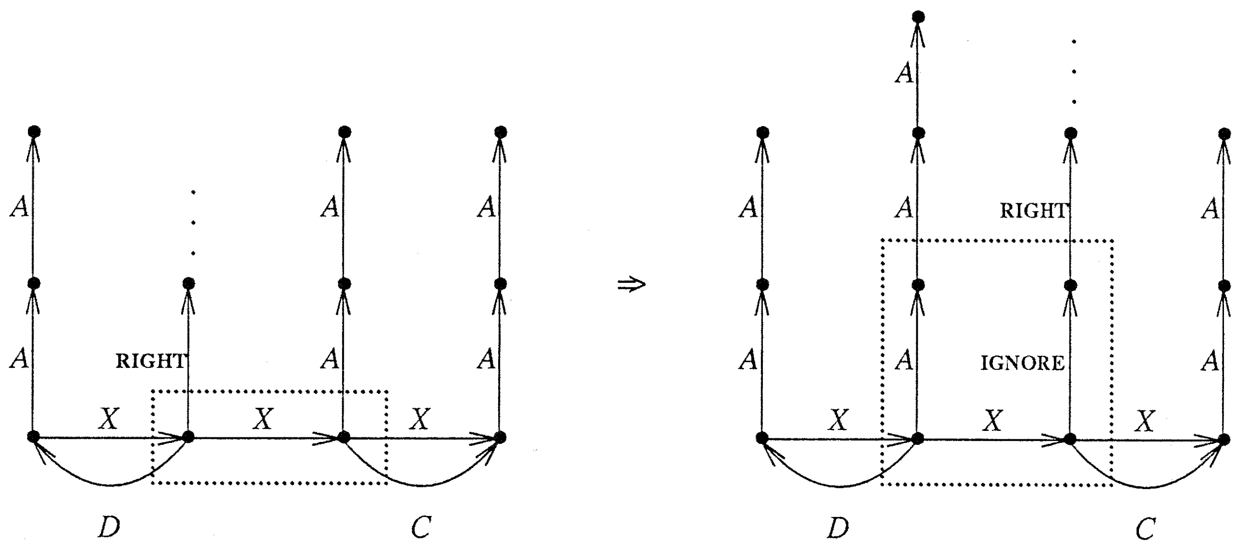
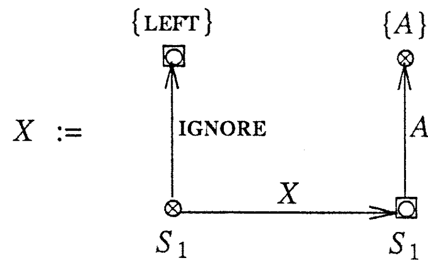


FIG. 5.2. The RIGHT instruction.

Production P2.



Example derivation step: The edges C and D in this example are from the graph which is being constructed. The $\cdots$ indicates the instruction stick. The mother edge and daughter graph are indicated by dotted boxes.

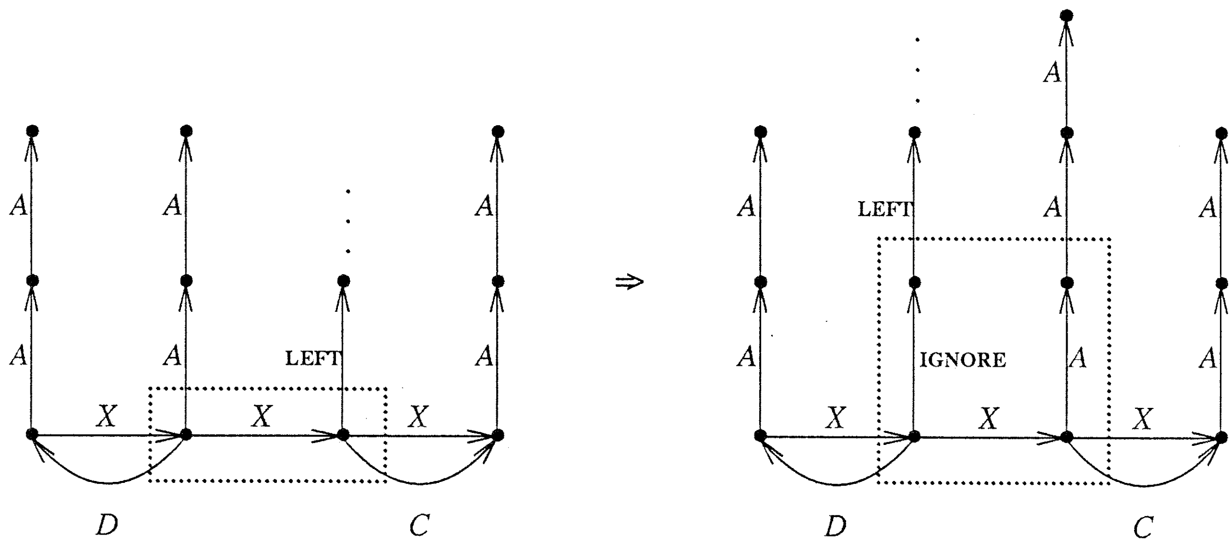
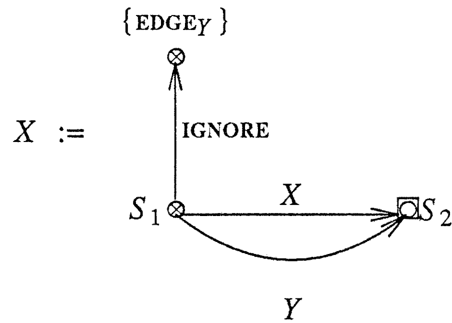


FIG. 5.3. The LEFT instruction.

Production P3. (For each $Y \in \Sigma$)



Example derivation step: The edges C and D in this example are from the graph which is being constructed. The $\cdots$ indicates the instruction stick. The mother edge and daughter graph are indicated by dotted boxes.

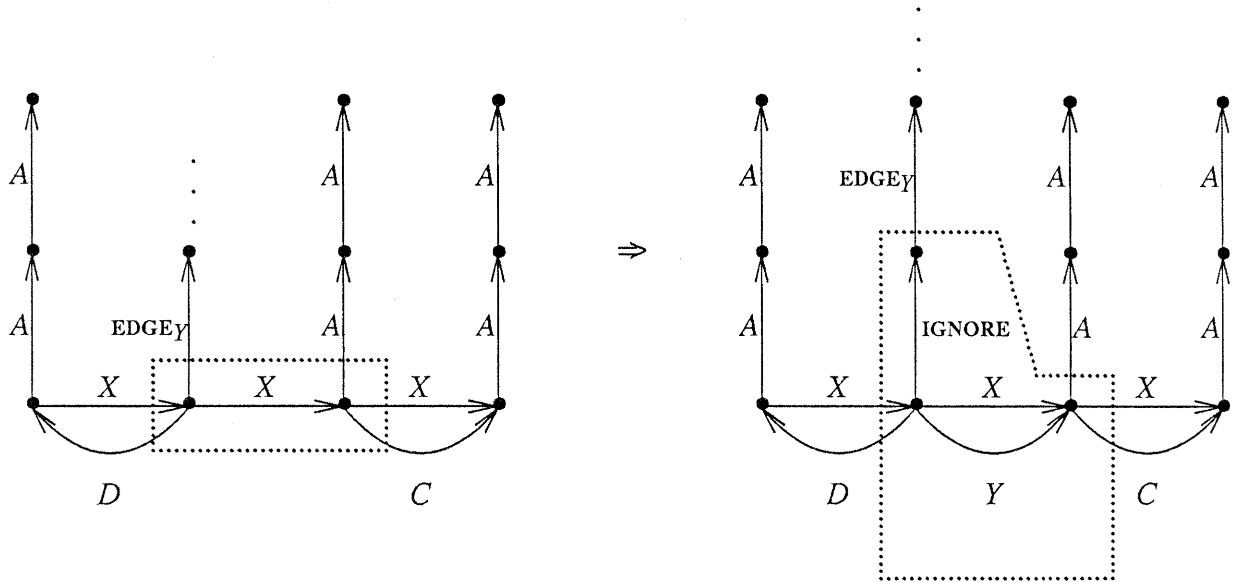
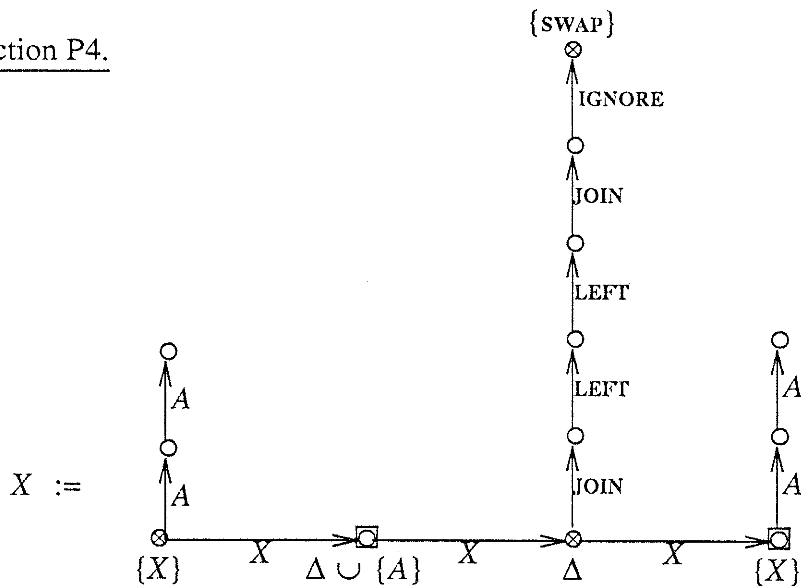


FIG. 5.4. The EDGE$_Y$ instruction.

Production P4.



Example derivation step: The edges C and D in this example are from the graph which is being constructed. The ··· indicates the instruction stick. The mother edge and daughter graph are indicated by dotted boxes.

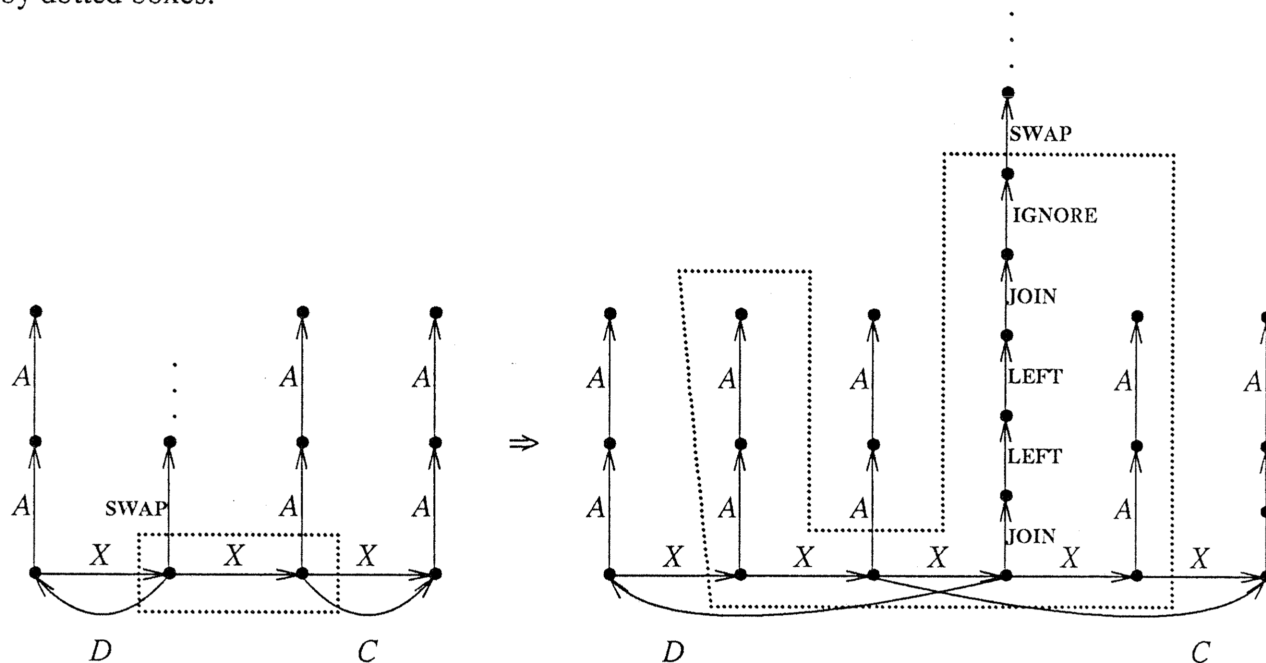
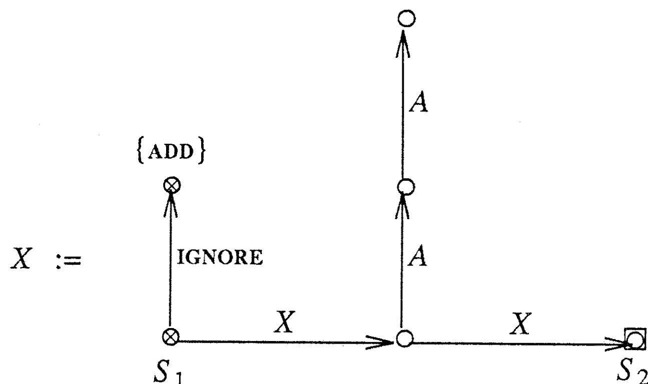


FIG. 5.5. The SWAP instruction.

Production P5.



Example derivation step: The edges C and D in this example are from the graph which is being constructed. The ··· indicates the instruction stick. The mother edge and daughter graph are indicated by dotted boxes.

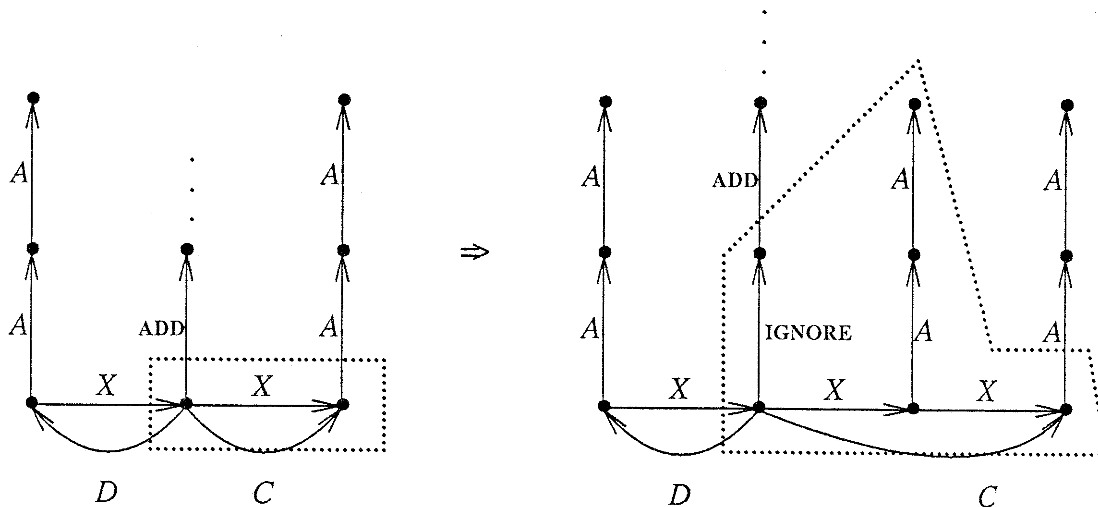
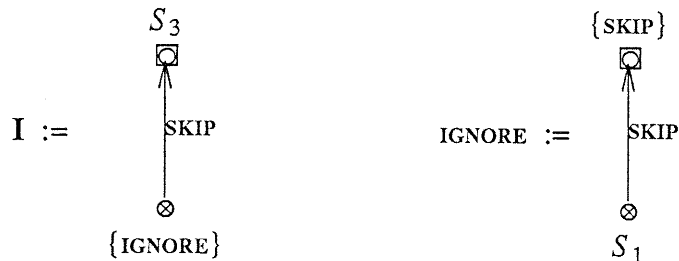


FIG. 5.6. The ADD instruction.

Productions P6 and P7. (For each instruction $\mathbf{I} \neq$ END)



Example derivation step: This shows how Productions P6 and P7 work together to execute an IGNORE instruction which precedes an ADD instruction. The edges C and D in this example are from the graph which is being constructed. The $\cdots$ indicates the instruction stick. The first mother edge is indicated by a dotted box; the second is indicated by a dashed box.

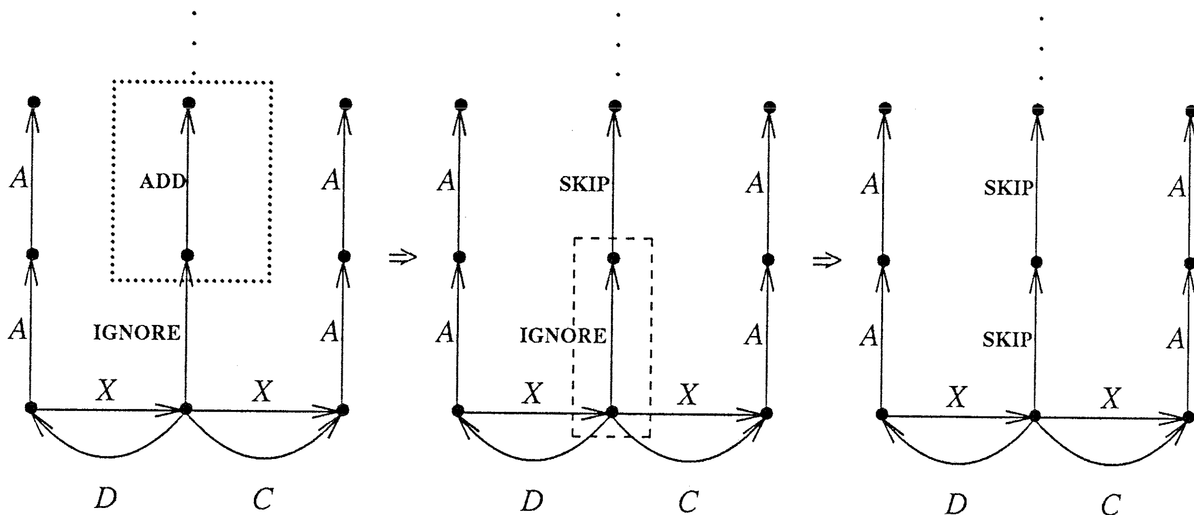
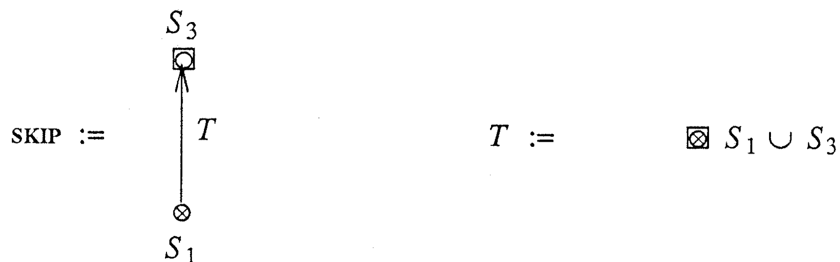


FIG. 5.7. The IGNORE instruction.

Productions P8 and P9.



Example derivation step: This shows how Productions P8 and P9 work together to execute a SKIP instruction. The edges C and D in this example are from the graph which is being constructed. The $\cdots$ indicates the instruction stick. The first mother edge is indicated by a dotted box; the second is indicated by a dashed box.

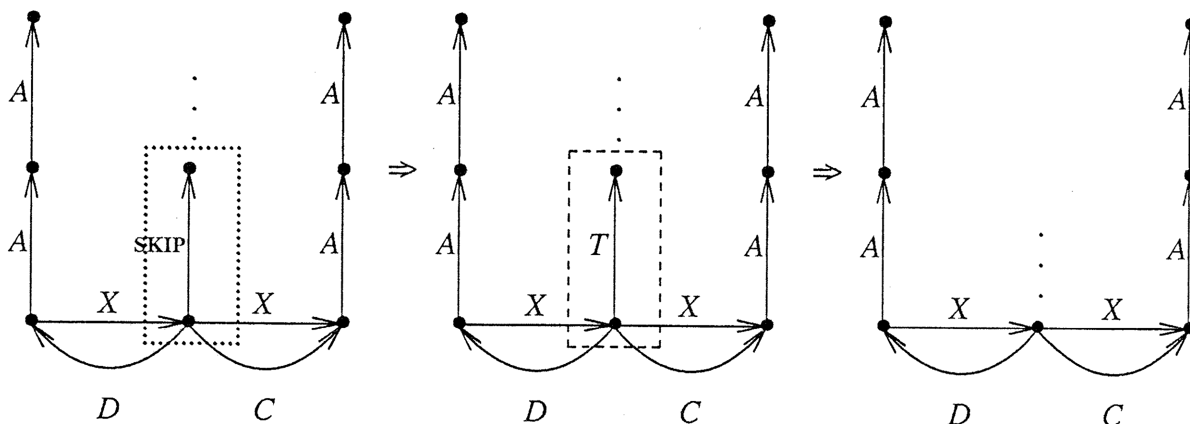
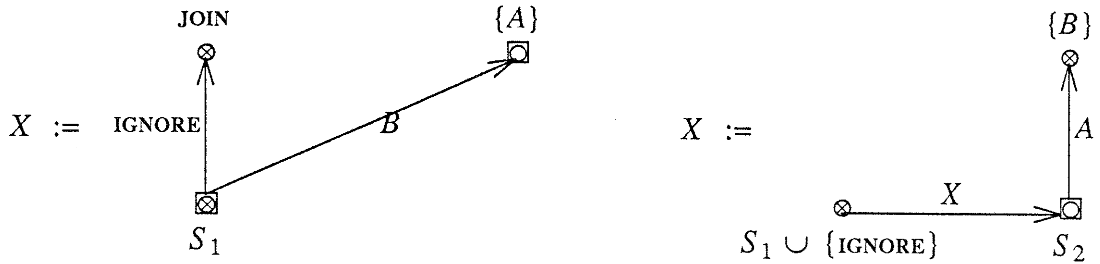


FIG. 5.8. The SKIP instruction.

Productions P10 and P11.



Example derivation step: This shows how Productions P10 and P11 work together to execute a JOIN instruction. Recall that this instruction cannot be used to join the last two nodes in the graph, hence P11 always has a chance to be applied. The edge C in this example is from the graph which is being constructed. The $\cdots$ indicates the instruction stick. The first mother edge is indicated by a dotted box; the second is indicated by a dashed box.

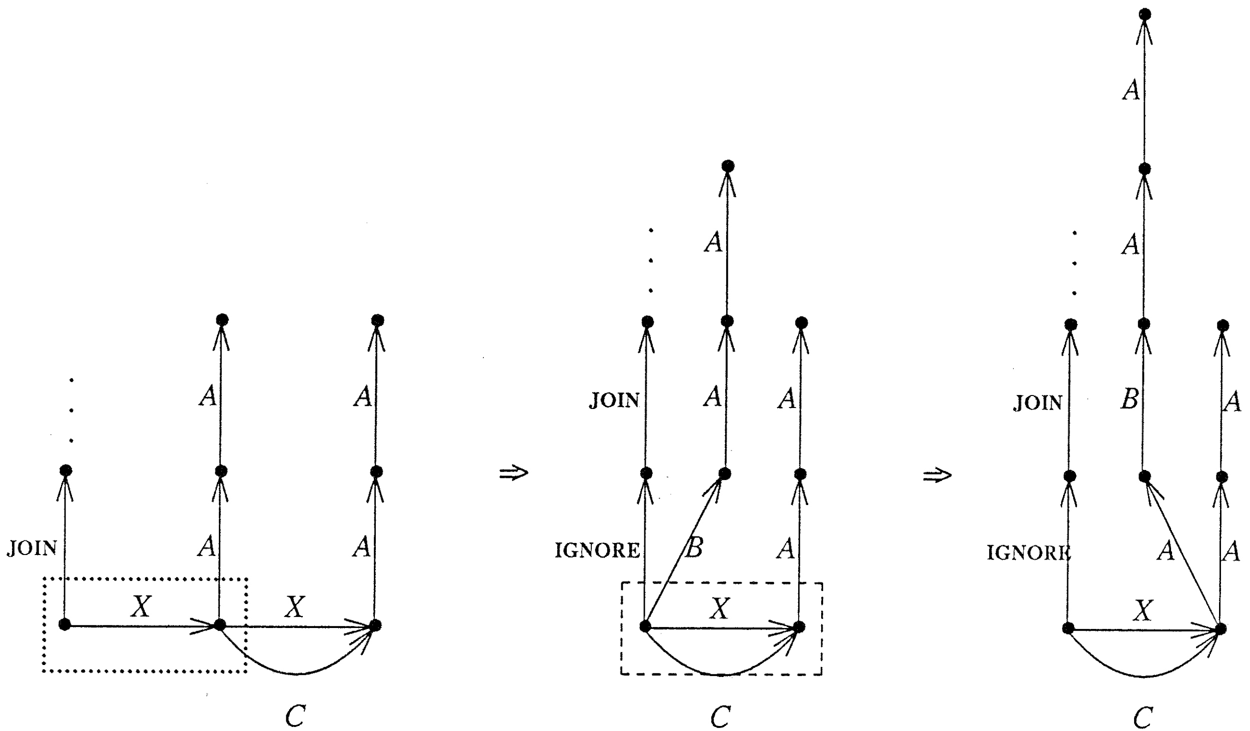
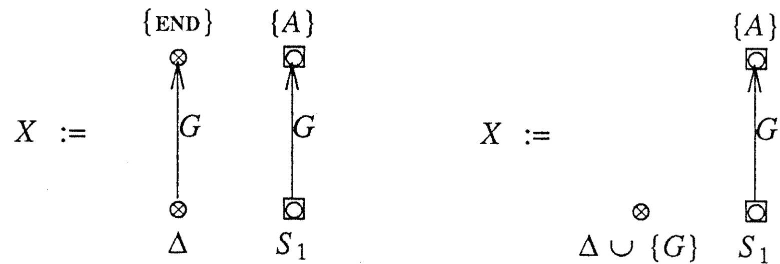


FIG. 5.9. The JOIN instruction.

Productions P12 and P13.



Example derivation step: This shows how Productions P12 and P13 get rid of the X-edges at the end of a derivation. The edges C and D in this example are from the graph which is being constructed. The first mother edge is indicated by a dotted box; the second is indicated by a dashed box.

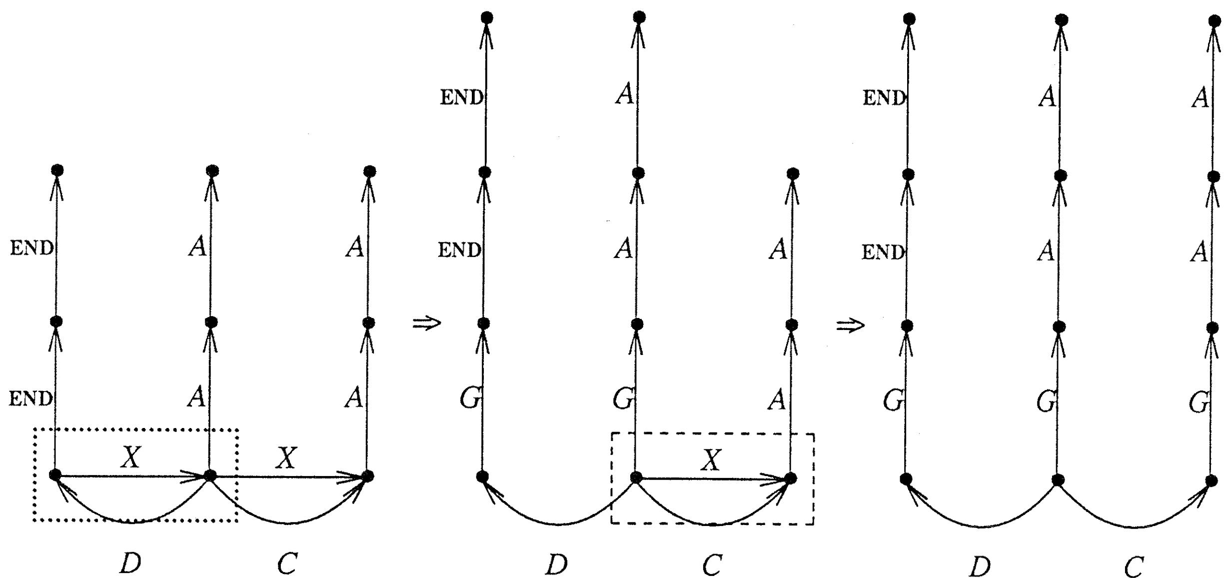
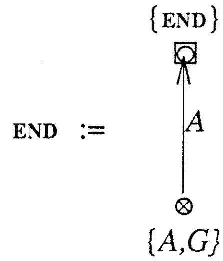


FIG. 5.10. Productions to remove X-edges.

Production P14.



Example derivation step: This productions is used to get rid of the END-edges at the end of a derivation, by changing them to A's. The mother edges are indicated by dotted boxes.

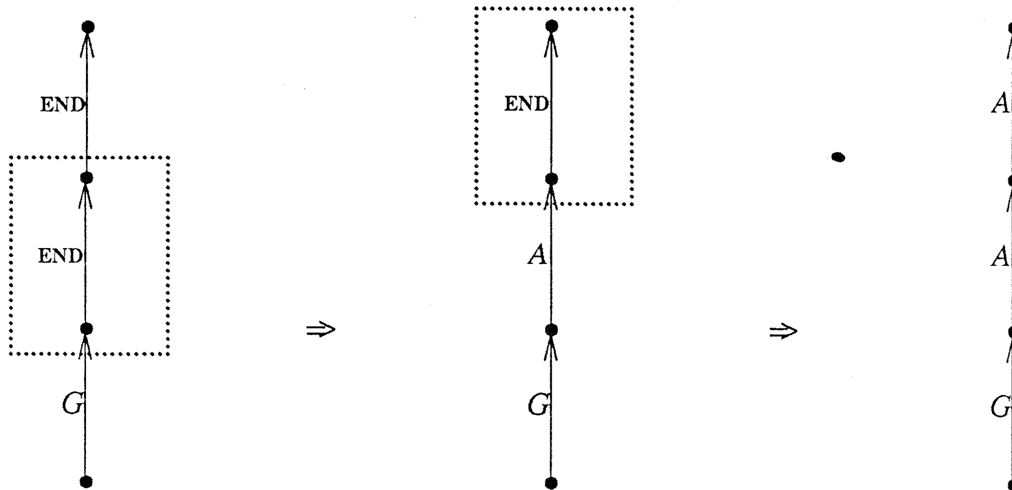
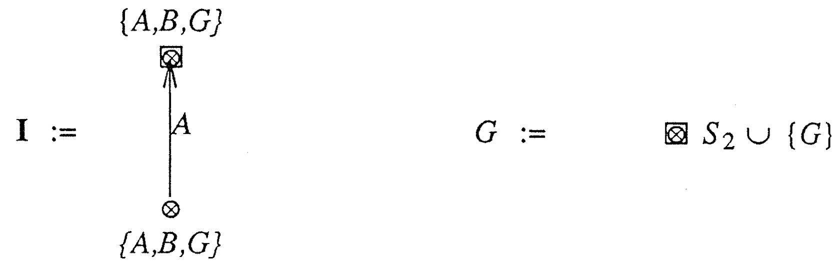


FIG. 5.11. Production to remove END-edges.

Productions P15 and P16 (For **I** = A or B).



Example derivation step: These productions are used to get rid of the A's, B's and G's at the end of a derivation, by collapsing the subgraphs with these edges to a single node. This is a simple example. The mother edges are indicated by dotted boxes.

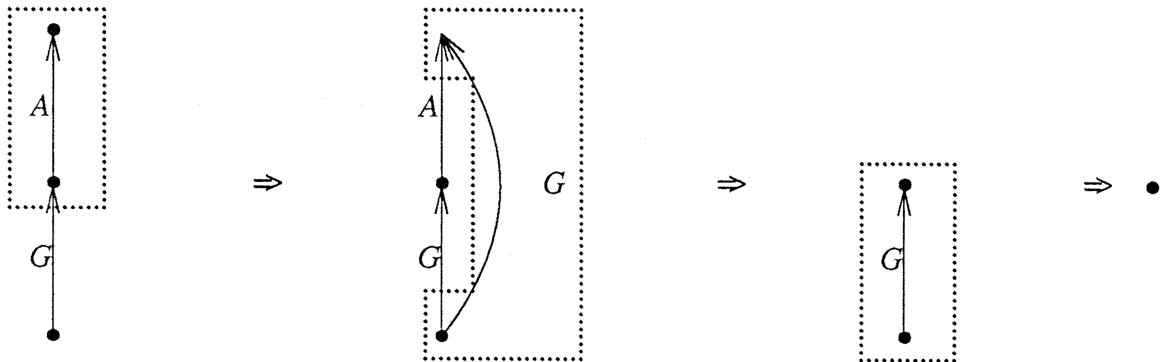


FIG. 5.11. Productions to remove A, B and G-edges.

Here are some comments about the productions:

1. Actually, it is possible to apply a "wrong" production to η_i. For example, the LEFT production might be applied when it should be a RIGHT production. However, the application of a "wrong" production always results in one or more of the "sticks" becoming disconnected from the rest of the graph. Once this happens, the derivation can never finish with a terminal graph. In particular, the only way an A-edge or an END-edge can be removed is if there is a G or X edge in the same connected component of the graph.
2. Now, suppose we start with the graph η_0. From the previous comment, we see that any derivation must "execute" the first instruction, yielding η_1. From here, the derivation must move to η_2, η_3, and so on, until it reaches η_{n-2}. At this point, the graph still contains two END instructions, and $current$=1.
3. Once the derivation reaches η_{n-2}, the derivation can only be finished off by using productions P12 through P16. This leaves us with the terminal graph μ, which the graph program $I_1, \cdots, I_n$ constructed.

The formal result is stated in Lemma 5.1.

Lemma 5.1. *Let* $P = I_1, \cdots I_n$ *be a graph program which constructs some graph* $\mu \in G_\Delta$ *(with* $|\mu| > 1$*). Let* η_i $(0 \le i \le n-2)$ *be one of the auxiliary graphs defined above. Then the only terminal graph generated from* η_i *using the graph scheme* S_Δ *is* μ. □

5.2 The R.E. Theorem

Suppose L is an arbitrary fixed recursively enumerable graph language over an alphabet Δ. This section defines an ELC grammar G_L which generates L. For simplicity, we assume that the one-node graph is not in L. Handling the case where the one-node graph is in L is a a simple extension of the case treated here.

By definition, **program**(L) is a recursively enumerable string language. Thus, there is a phrase-structure grammar GP which generates precisely **program**(L) (see [4]). From normal form results, we can assume the following properties of this grammar:

A. The nonterminal alphabet of GP is disjoint from the alphabet of the program scheme S_Δ.

B. Every production of GP is a context-free production ($Z \rightarrow \alpha$, where Z is a single nonterminal and α is a string of symbols) or a production of the form $UY \rightarrow UZ$ (where U, Y and Z are all nonterminals).*

C. In any derivation of GP, the leftmost symbol remains a nonterminal until after the final production is applied.

Using GP and the grammar scheme S_Δ (of Section 5.1), we can define the ELC-grammar G_L which generates precisely L. As in the previous section, the productions are given in the explicit format, with explicit sets of labels attached to each node of a daughter graph.

Labels, Terminals and Start Symbol of G_L:

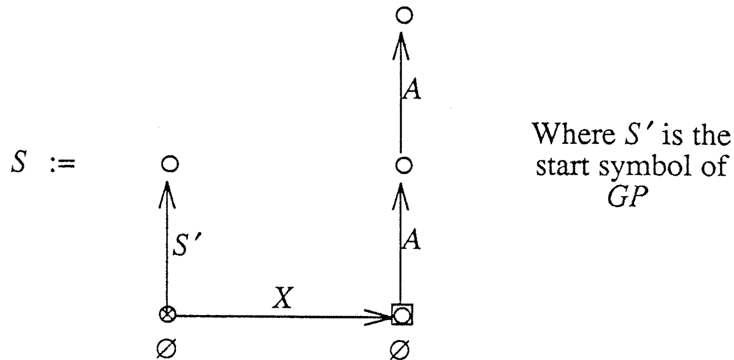
The label set (denoted by Σ) consists of the labels of S_Δ, plus the labels of GP plus a new start symbol S. The terminals are Δ.

Connection relation:

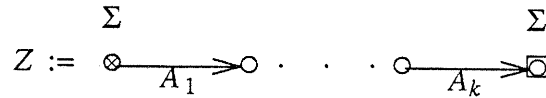
This is all of $\Sigma \times \Sigma$.

*The idea to use this sort of grammar was suggested to us by Hans-Jorg Kreowski. It has been shown to be a normal form by [10]. This results in a much simplier presentation than our original scheme which employed Turing machines.

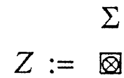
R1. Production for the start symbol (S):



R2. For each context-free production $Z \rightarrow A_1 \cdots A_k$ ($k > 0$) in GP, the graph grammar G_L has this production:



R3. For each context-free production $Z \rightarrow \varepsilon$ in GP, the graph grammar G_L has this production:



R4. For each non-context-free production $UY \rightarrow UZ$ in GP, the graph grammar G_L has this production:

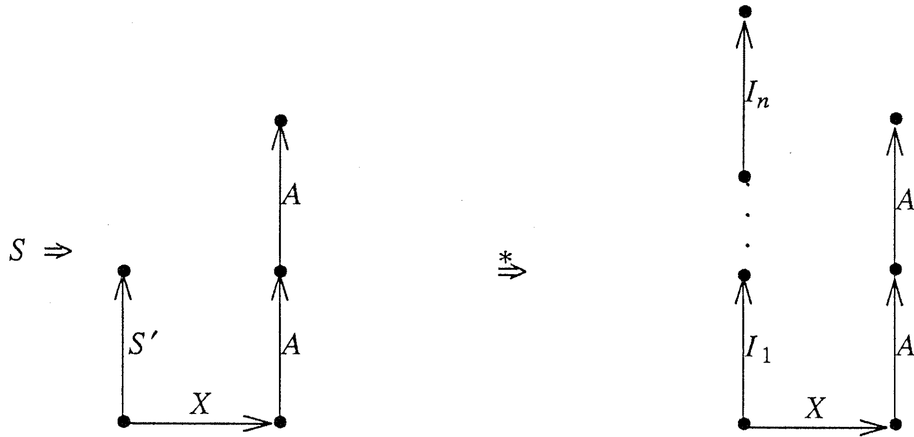
$$Y := \overset{\{U\}}{\otimes} \xrightarrow{Z} \overset{\Sigma}{\boxdot}$$

R5. Productions from S_Δ:

Each of the productions of S_Δ is also a production of G_L.

Lemma 5.2. *If* $\mu \in L$ *then* G_L *generates* μ.

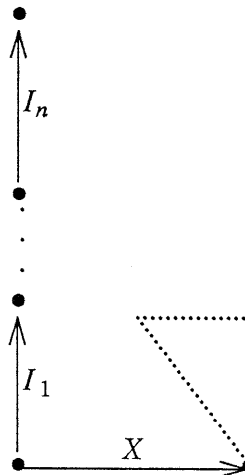
Proof: Let $\mu \in L$ be a graph from the language L, and let **program**$(\mu) = I_1 \cdots I_n$. Using productions R1 through R4, there is a derivation in G_Δ:



Let η_0 be the last graph in this derivation. From Lemma 5.1, there is a derivation $\eta_0 \overset{*}{\Rightarrow} \mu$, using only the rules from the grammar scheme S_Δ (rules R5 above). Therefore, in G_L, there is a derivation $S \overset{*}{\Rightarrow} \eta_0 \overset{*}{\Rightarrow} \mu$. □

Lemma 5.3. *If G_L generates a graph μ, then $\mu \in L$.*

Proof: There are two types of derivations in G_L. In one case, the graphical form eventually contains a disconnected component with an *A,B* or END label, but no *X* or *G* label. Such a derivation cannot be completed to a terminal graph. On the other hand, a derivation can start with production R1, then continue with productions R2-R4, never applying R3 in a situation which would disconnect the graph. In this case, the sentential form will eventually become this:



The sequence $I_1 \cdots I_n$ is a graph program for some graph $\mu \in L$, and the dotted triangle is a connected graph of *A*-edges, with at least one edge that is not adjacent to the node at the bottom. At this point, the only productions which can be applied are those from S_Δ (rules R5 above). By Lemma 5.1, this derivation can only be finished by generating $\mu \in L$. □

Theorem 5.4. *A graph language is recursively enumerable iff if is an ELC language.*

Proof: Given an ELC language L, it is easy to build a Turing machine which enumerates **program**(L) — hence L is recursively enumerable. On the other hand, if L is recursively enumerable (and does not contain the one-node graph), then the previous two lemmas show that the ELC grammar G_L generates L. If L is recursively enumerable and does contain the one-node graph, then an ELC grammar for L is obtained by adding one production (S := ⊠) to an *ELC* grammar for $L - \{\bullet\}$ □

6. A Hierarchy of Language Classes

We have shown that the ELC grammars have the power of recursive enumerability. In fact, this is generally *too* powerful since any nontrivial problem about the language generated by an ELC grammar is undecidable (by an analog of Rice's Theorem [14]). So, the result of the previous section should be taken as a warning: It may be easy to specify languages with an ELC grammar, but all of the problems we may wish to answer about such languages are undecidable.

With this in mind, we want to impose various restrictions on ELC grammars, so that the resulting classes of languages are still "powerful enough", but that they have more manageable properties. With this in mind, we single out three basic features of the "direct derivation step" in an ELC grammar, and then impose restrictions based on these features. In this way we get the following restrictions:

1. Complete Connection Relation.

This requires the connection relation C to be complete — *i.e.*, $C = (\Sigma \cup \{\textit{ISOLATED}\}) \times \Sigma$.

2. Disjoint *source* and *target* sets

This requires α_{source} and α_{target} to be disjoint for each production rule.

3. Singleton *source* and *target* sets

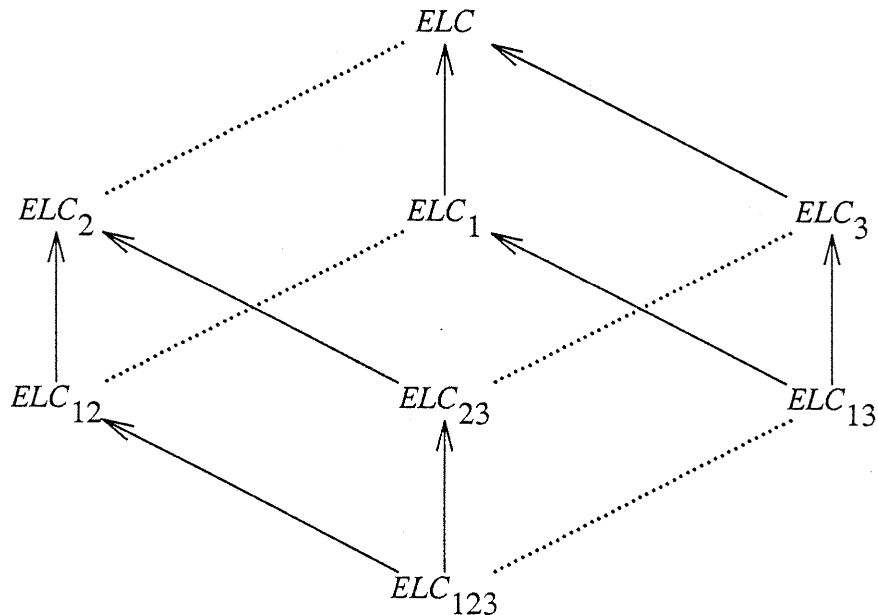
This requires α_{source} and α_{target} to always be singletons in each production rule. However,

they do not need to be disjoint, so that there might be only a single node from α, which is both α_{source} and α_{target}.

When all three of these restrictions are imposed simultaneously, the resulting grammars are the *edge-replacement systems* introduced by Habel and Kreowski [2,3], with one technical change.* The fact that all three restrictions taken together yields this known class of languages gives some justification for our choice of features – especially so since the Habel-Kreowski class has a number of nice properties similar to context-free string grammars. It also has some drawbacks, such as the inability to generate sets of graphs where the number of connected pairs of nodes is nonlinear with respect to the total number of nodes.

On the other hand, when the restrictions are taken in various combinations, the result is the hierarchy of language classes drawn here. (The subscripts indicate which restrictions have been placed on the grammars. The solid arrows indicate proper inclusions of language classes, and the dotted lines are inclusions which may or may not be proper.)

*The systems used by Habel and Kreowski also allow loops (an edge from a node to itself). But, if loops are not allowed, the result is the same as ELC grammars with the three restrictions. There are several sensible ways to handle loops with ELC grammars, but none of these methods correspond to the treatment given by Habel and Kreowski.



The remainder of the paper is a preliminary consideration of some properties of these eight classes of languages. We focus on demonstrating the eight proper inclusions indicated above.

7. Requiring Singleton *source* and *target* Sets

ELC_3 grammars require α_{source} and α_{target} to be singleton sets in each production. Intuitively, this restriction means that a single edge in the neighborhood does not become duplicated when an adjacent edge is replaced. Such an edge remains a single edge, or it may be erased entirely if the connection relation is not complete. The graphs generated by a grammar with this restriction cannot become too highly connected. The formal result is in this lemma:

Lemma 7.1. *Let L be a language generated by an ELC_3 grammar. Then there exists a constant c such that for any graph $\gamma \in L$, the total number of pairs of adjacent nodes in γ is no more than $c\,|\,nodes(\gamma)|$.*

Proof: Let G be the ELC_3 grammar which generates L. The constant c is the maximum number of nodes in a daughter graph of G (with $c > 0$). Actually, we can prove a result which is stronger than the lemma. The result is this: Let γ be any graphical form of G, and suppose the nodes of γ are partitioned into m disjoint sets $S_1, S_2, \cdots, S_m$, where each set S_i is connected. An unordered pair of sets $\{S_i, S_j\}$ is called adjacent if there is an edge from a node of S_i to a node of S_j, or vice versa. We will prove that the total number of pairs of adjacent sets is no more than cm. (This immediately implies the lemma, by taking γ to be a terminal graph and partitioning it into singleton nodes.)

The proof of the stronger result is an induction on the length of the derivation of γ. The base step (γ is $\bullet \xrightarrow{S} \bullet$) is trivial. For the induction step, assume the result holds for graphical forms with derivations of length k, and suppose we now have a graphical form γ with a derivation of length k+1. Let $S_1, \cdots, S_m$ be the partition of γ (as above). Also, let γ' be the next-to-last graphical form in the derivation of γ, and let $X := (\alpha, \alpha_{source}, \alpha_{target})$ be the final production which takes γ' to γ. (Recall that α_{source} and α_{target} must be singleton sets. Because of this we will abuse notation and let α_{source} and α_{target} be the actual single nodes rather than singleton sets of nodes.)

The nodes of γ' are nearly identical to those of g. The only differences are that γ' has two extra nodes (the mother source and mother target) which γ does not have, and γ has the nodes of the daughter graph which γ' does not have. From this, we see that the partition $S_1, \cdots, S_m$ of γ also gives us a partition $S'_1, \cdots, S'_m$ of γ', as follows: If α_{source} is in set S_i, then the mother source node in γ' is in set S'_i; if α_{target} is in set S_i, then the mother target node in γ' is in set S'_i; each other node of γ' is placed in the same set as its corresponding node in γ. Of course, some of the resulting partition sets of γ' may be empty, but this is not forbidden. We will let n denote the

number of such empty sets in the partition of γ'. Also note that if a node v of the daughter graph is in set S_i, then S'_i is either empty, or it contains the mother source or mother target. (Otherwise it is not connected.)

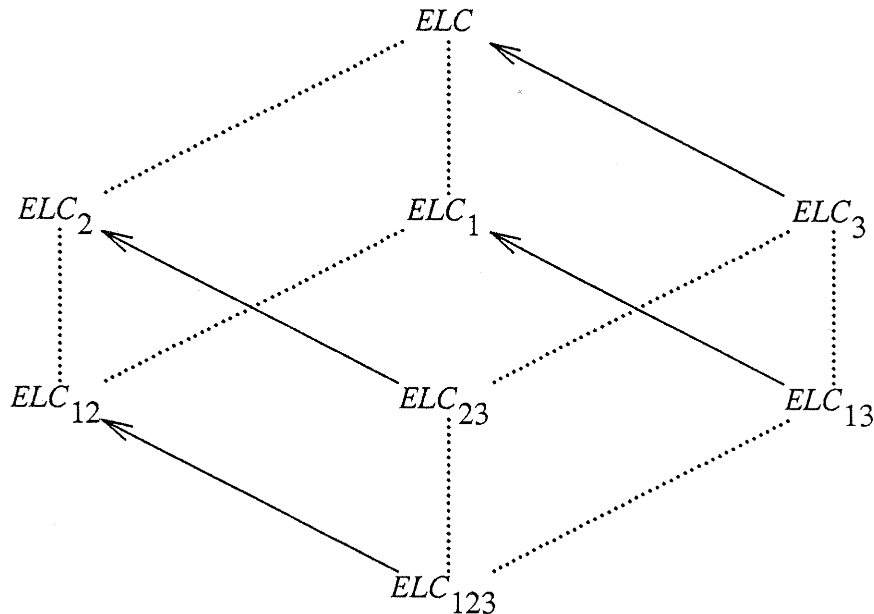
What adjacent pairs can occur in the partition of γ? There are only two sorts of adjacencies that can arise:

1. A pair $\{S_i, S_j\}$ may be adjacent in γ because $\{S'_i, S'_j\}$ is adjacent in γ'. By the induction hypothesis, the maximum number of such adjacencies is $c(m-n)$.
2. Let S_i be one of the sets which contains at least one of the daughter nodes, and let S_j be a set such that S'_j is empty. These might be adjacent in γ, even if they were not adjacent in γ' (since S_j can contain one of the daughter graph nodes). Now, there are no more than c choices for S_i and no more than n choices for S_j. Therefore, the number of adjacencies of this kind is no more than cn.

Hence, the number of pairs of adjacent sets in γ is no more than $c(m-n) + cn \ = cm$, as required. □

One consequence of the above lemma is that the language G_Δ of all graphs over Δ cannot be generated by an ELC_3 grammar (when Δ is nonempty). However, when restriction 3 is removed, it is not difficult to generate G_Δ, even if the other two restrictions are present. The technique is to first generate graphs with lots of edges, then selectively remove edges. This does not work when restriction 3 is present, because graphs with "lots of edges" cannot be generated in the first place. This gives the following corollary:

Theorem 7.2. *The lines indicated by solid arrows in the following diagram are proper inclusions:*



8. Requiring a Complete Connection Relation

ELC_1 grammars require the connection relation to be complete. This means that once an edge is established, it can never be removed, except by rewriting it or by "collapsing" its source and target nodes. We show three results about this restriction. First, we show that certain non-trivial language problems are decidable for ELC_1 grammars. This implies that the ELC_1 languages are a proper subclass of all R.E. languages. The other two results are combinatorial lemmas about the languages generated by ELC_{12} and ELC_{13} grammars. These lemmas are used to show three other proper inclusions of language classes.

8.1. The Reduction from R.E.

For convenience, we assume throughout this subsection that all *ELC* grammars have nonterminal labels chosen from some fixed countably infinite set N, and they have terminal labels chosen from some other fixed countably infinite set T. This does not effect the generative power of the grammars (up to a renaming of terminal labels).

Suppose we have a collection of ELC grammars **H**. For example, **H** might be all of the ELC grammars, or perhaps only those grammars that meet certain specified restrictions (like ELC_1). The class of grammars **H** is called *recursive* provided that it is decidable whether an arbitrary ELC-grammar is in **H**. A *language problem on* **H** is a problem of the form: "*Given a grammar G (from* H*), does L(G) ...?*" The question (indicated by "...?") can be any yes-no question like "*contain a discrete graph*?" or "*contain an infinite number of graphs*?". A language problem on **H** is *nontrivial* if there are grammars in **H** where the answer is "*yes*", and there are other grammars in **H** for which the answer is "*no*". A modification of the proof of Rice's Theorem [14] gives us the following result.

Lemma 8.1. *Let* **H** *be a recursive collection of ELC grammars such that every recursively enumerable graph language is generated by some grammar of* **H**. *Then every non-trivial language problem on* **H** *is undecidable.*

Proof: Let $H_1, H_2, \cdots$ be some effective enumeration of those grammars in **H** which have no terminal symbols. Also, let γ_i denote the discrete graph with i nodes and no edges. A diagonalization argument shows that

$$K = \{\gamma_i \mid \gamma_i \notin H_i\}$$

is not a language generated by a grammar of **H**, hence K is not recursively enumerable.

Next, consider the *membership problem* for **H**: *Given a grammar H of* **H** *and a graph* γ, *determine whether* $\gamma \in L(H)$. If this problem is decidable, then the following would be an R.E. string

language:

$$\{\mathbf{program}(\gamma_i) \mid \gamma_i \notin H_i\}$$

(where **program** is the function from Section 4). But, this would imply that K (from the previous paragraph) is recursively enumerable, which we know is not true. Therefore, the membership problem for **H** is undecidable.

Finally, suppose P is a nontrivial language problem on **H**, and suppose there is an algorithm to decide P. We can proceed as in the proof of Rice's Theorem to use this algorithm to decide the membership problem for **H**. By this contradiction we conclude that P must be undecidable. □

Now, consider ELC_1. This class of grammars is recursive, but the "*one-node*" problem is decidable in this class, as shown in this lemma:

Lemma 8.2. *Let* $G \in ELC_1$. *It is decidable whether the one-node graph is in* $L(G)$.

Proof: First, we define the "destructive" labels of G to be the smallest set of labels such that:

1. If X := ⊠ is a production of G, then X is destructive.
2. Let X := β be a production of G. (We don't care what the *source* and *target* sets are.) Define β' to be the subgraph of β obtained by keeping only destructive edges. If β' is connected, then X is destructive.

Such a smallest set exists since the set of all labels meets these two conditions, and whenever two sets of labels meet the conditions then so does their intersection. In fact, the smallest set of labels to meet these two conditions can be computed by starting with those labels that must be destructive by rule 1, then continue to add any new labels that are forced to be destructive by rule 2 until eventually no new labels need to be added. We claim that $X \overset{*}{\Rightarrow} \bullet$ iff X is destructive. Therefore, $\bullet \in L(G)$ iff the start label is destructive, and this can be determined by computing the set of destructive labels. □

Because the one-node problem is non-trivial, the previous two lemmas imply that ELC_1 does not generate all of the R.E. languages:

Theorem 8.3. *$L(ELC_1)$ is a proper subclass of $L(ELC)$.* □

8.2.* A Result about Restrictions 1 and 2

This section demonstrates that the graphs generated by ELC_{12} grammars always contain a certain simple kind of subgraph. The result needs a few preliminary definitions:

Definition 8.1. Let γ be a graph and S a subset of *nodes*(γ).

a. *neighborhood*(S) = $\{v \in nodes(\gamma) - S \mid$ there exists some node in S which is directly connected to v by an edge of $\gamma\}$.

b. S is a *2-simple* set in γ provided that *neighborhood*(S) can be expressed as a union *neighborhood*(S) $= N_1 \cup N_2$, and for each $v \in S$, either

  (i) v is directly connected to none of *neighborhood*(S), or

  (ii) v is directly connected to all of N_1 and none of $N_1 - N_2$, or

  (iii) v is directly connected to all of N_2 and none of $N_2 - N_1$.

□

Lemma 8.4. *Let L be an ELC_{12} language. Then there exists a constant c such that for any $\alpha \in L$ (with $|nodes(\alpha)| > c$), there exists $\alpha' \in L$ such that:*

(a) $|nodes(\alpha)| = |nodes(\alpha')|$, and

(b) α' has a 2-simple subset S with $3 \leq |S| \leq c$.

* The results of this section were developed with Mark Brissenden. It is related to a similar result about NLC-grammars [5].

Proof: Let G be an ELC_{12} grammar generating L, and let c be the maximum number of nodes in a daughter graph of a production of G. Consider a graph $\alpha \in L$ (with $|nodes(\alpha)| > c$), and a derivation $S \overset{*}{\underset{G}{\Rightarrow}} \beta \underset{G}{\Rightarrow} \gamma \overset{*}{\underset{G}{\Rightarrow}} \alpha$ of α. The graph γ in this derivation is the first point in the derivation such that $|nodes(\gamma)| = |nodes(\alpha)|$. The daughter graphs used in the remainder of the derivation all have precisely two nodes – otherwise the number of nodes in the graphical form grows beyond $|nodes(\alpha)|$, and because of restriction 2, it cannot shrink again. Thus, each production in the derivation $\gamma \overset{*}{\underset{G}{\Rightarrow}} \alpha$ is the replacement of a mother edge by zero or more new edges. In general, α is obtained from γ by replacing each nonterminal edge by zero or more terminal edges. We now modify the derivation $\gamma \overset{*}{\underset{G}{\Rightarrow}} \alpha$ as follows:

1. For each nonterminal label X which appears in γ, choose some arbitrary two node graph δ_X such that $X \overset{*}{\underset{G}{\Rightarrow}} \delta_X$. Such a graph must exist, otherwise the derivation $\gamma \overset{*}{\underset{G}{\Rightarrow}} \alpha$ would not be possible without increasing the number of nodes.

2. Apply the derivation $X \overset{*}{\underset{G}{\Rightarrow}} \delta_X$ to each nonterminal label in γ. Let α' be the resulting terminal graph.

The graph α' is in L and has the same number of nodes as γ and α. In fact, since α' is obtained by making single-edge replacements to γ, we may as well identify the nodes of α' with the nodes of γ. Let S be the set of nodes in α' that were introduced in the daughter graph of the step $\beta \underset{G}{\Rightarrow} \gamma$, and let σ be this daughter graph. We have $3 \leq |S| \leq c$. Moreover, S is 2-simple, which is shown as follows: The division of the neighborhood of S is $N_1 \cup N_2$, where N_1 contains those neighborhood nodes which were adjacent to σ_{source} in γ, and N_2 contains those neighborhood nodes which were adjacent to σ_{target} in γ. Now the nodes of S are precisely the nodes of σ, and there are three cases for such a node:

1. A node which is in neither σ_{source} nor σ_{target}. Such a node is not directly connected to any node

in N_1 or N_2, since it cannot be directly connected to any of the neighborhood of σ in the graphical form γ.

2. A node which is in σ_{source} but not σ_{target}. Because of the complete connection relation, such a node must remain connected to all of N_1. However, it cannot be directly connected to any of $N_1 - N_2$, because it is not connected to any of these nodes in γ.

3. A node which is in σ_{target} but not σ_{source}. Because of the complete connection relation, such a node must remain connected to all of N_2. However, it cannot be directly connected to any of $N_2 - N_1$, because it is not connected to any of these nodes in γ. □

If the connection relation is not complete, then it is possible to generate an infinite language of discrete graphs which does not meet the requirements of Lemma 8.4. In particular, the the "wheel" graphs generated by Example 2.5 do not meet the requirement of the lemma. This is because the only 2-simple subgraphs of the n-node wheel graph have less that 3 nodes, or more than $n-3$ nodes. Therefore, this language cannot be generated by an ELC_{12} grammar. However, Example 2.5 is an ELC_2 grammar, giving the following result:

Theorem 8.5. *$L(ELC_{12})$ is a proper subclass of $L(ELC_2)$.* □

8.3. A Result about Restrictions 1 and 3

The primary result of this section is similar to the "interchange lemma" for context-free string languages [13]. The result applies to ELC_{13} grammars. The current version of the proof needs a number of preliminaries, some of which might be eliminated in a simplier proof. (Although the concepts in these preliminaries may be useful elsewhere too.) Here are the preliminaries:

The follow function:

If we have a derivation step $S:\alpha \Rightarrow \beta$, in an ELC_3 grammar, then the nodes of α can be mapped to the nodes of β by a function $follows: nodes(\alpha) \rightarrow nodes(\beta)$. The function $follows$ maps the mother source node to the source node of the daughter graph, and it maps the mother

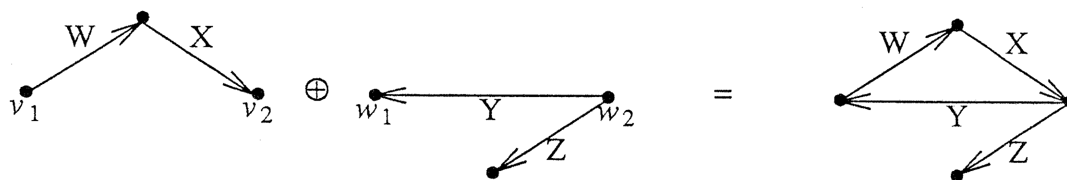
target node to the target node of the daughter graph. All other nodes of α are also nodes of β, so $follows_S$ is the identity function on these nodes. Similarly, if $D:\alpha \overset{*}{\Rightarrow} \beta$ is a derivation of zero or more steps, then we can define $follow_D : nodes(\alpha) \rightarrow nodes(\beta)$, by composing the *follow* functions of the individual derivation steps.

DP-graphs:

A *double-pointed graph* (or *DP-graph*) is a triple (θ, v_1, v_2), where θ is a graph and v_1, v_2 are distinguished nodes of θ. A DP-graph is *proper* if its two distinguished nodes are not the same node. For a label X, we will abuse notation, and let X also stand for the proper two-node DP-graph with a single X-edge from the first node to the second node. We also ambiguously let $\bullet$ denote the one-node DP-graph. We will use capital Greek letters (Θ, Ψ) for DP-graphs, and *DISCRETE* denotes the set of all discrete DP-graphs.

The $\oplus$ operation on DP-graphs:

Let $\Theta = (\theta, v_1, v_2)$ and $\Psi = (\psi, w_1, w_2)$ be two DP-graphs. The graph $\Theta \oplus \Psi$ is obtained by "merging" θ and ψ, and identifying their pairs of distinguished nodes. Here is an example:



We give the formal definition for the case where Ψ is proper or neither of the DP-graphs is proper. (The remaining case, where Θ is proper and Ψ is not proper, is symmetric to the case where Ψ is proper and Θ is not). In this case, the nodes of $\Theta \oplus \Psi$ are

$$(nodes(\theta) \cup nodes(\psi)) - \{w_1, w_2\}.$$

The edges of θ remain the same, as do those edges of ψ which were not incident to its distinguished nodes. An edge whose source was w_k (k=1 or k=2) in ψ has a new source v_k, and

similarly for targets. (The exception is an edge between the two distinguished nodes of ψ, when θ is not proper. Such an edge disappears, since we do not allow self-loops.)

The $\oplus$ operation on sets of DP-graphs:

Let A and B be two sets of DP-graphs. The graph language $A \oplus B$ is defined as $\{\Theta \oplus \Psi \mid \Theta \in A$ and $\Psi \in B\}$.

L_1, L_2, L_3:

Let G be an ELC_3 grammar, and let $\Theta = (\theta, v_1, v_2)$ be a proper DP-graph. We can define three graphs languages. The first language consists of the discrete graphs that can be generated from θ without "collapsing" v_1 and v_2. The second language is the same, except v_1 and v_2 must be collapsed. The third language is needed for some special cases later on.

$$L_1(\Theta) = \{(\psi, follow_D(v_1), follow_D(v_2)) \in DISCRETE \mid \text{For some derivation } D : \theta \stackrel{*}{\Rightarrow} \psi,\ follow_D(v_1) \neq follow_D(v_2)\}$$

$$L_2(\Theta) = \{(\psi, follow_D(v_1), follow_D(v_2)) \in DISCRETE \mid \text{For some derivation } D : \theta \stackrel{*}{\Rightarrow} \psi,\ follow_D(v_1) = follow_D(v_2)\}$$

$$L_3(\Theta) = \{\Psi \in DISCRETE \mid \text{For some derivation } D : \theta \stackrel{*}{\Rightarrow} \psi,\ follow_D(v_1) \neq follow_D(v_2),\ \Psi \text{ is obtained from } (\psi, follow_D(v_1), follow_D(v_2)) \text{ by merging the two points.}\}$$

In the last line, Ψ is obtained by identifying $follow_D(v_1)$ with $follow_D(v_2)$.

Derivations of discrete graphs:

Let G be an ELC_{13} grammar which generates only discrete graphs, and let $(\Theta \oplus \Psi)$ be a graphical form for G. The discrete graphs which can be generated from $(\Theta \oplus \Psi)$ are completely defined by the following:

$$\{\gamma \mid (\Theta \oplus \Psi) \stackrel{*}{\Rightarrow} \gamma\} = L_1(\Theta) \oplus L_1(\Psi) \ \cup\ L_2(\Theta) \oplus L_3(\Psi) \ \cup\ L_3(\Theta) \oplus L_2(\Psi)$$

Lemma 8.6. *Let L be an infinite language of discrete graphs, generated by an ELC_{13} grammar. Then L contains four graphs $\alpha_1, \alpha_2, \beta_1, \beta_2$ such that $\alpha_1 \neq \beta_1$ and $|nodes(\alpha_1)| - |nodes(\alpha_2)| = |nodes(\beta_1)| - |nodes(\beta_2)|$.*

Proof: Assume that L does not contain four graphs as specified in the lemma. From this, we will show that L contains only a finite number of graphs. Let G be the ELC_{13} grammar which generates L. The proof proceeds by assigning a "type" to each derivation in G. There will be only a finite number of derivation types, and each derivation type will generate only a finite number of different graphs. Therefore, G generates only a finite number of graphs.

Let S be the start symbol of the grammar, and consider a derivation $D: S \overset{*}{\Rightarrow} \gamma$. We assign a "type" to this derivation as follows:

1. If $L_1(S) = \{\gamma\}$, then D has type "1". Otherwise, continue to step 2.
2. If $L_2(S) = \{\gamma\}$, then D has type "2". Otherwise, continue to step 3.
3. If neither of the previous two steps has assigned a type to D, then there must be some label X, a proper DP-graph Θ and a pair (i,j) such that:

   A. The derivation D has the form $S \overset{*}{\Rightarrow} (X \oplus \Theta) \overset{*}{\Rightarrow} \gamma$.

   B. (i,j) is (1,1), (2,3) or (3,2).

   C. $|L_i(X)| > 1$.

   D. $\gamma \in (L_i(X) \oplus L_j(\Theta))$.

   For example, we can choose $X = S$, and let Θ be the two-node discrete proper DP-graph, and conditions A-D will all be met for some (i,j). (In particular, condition C is met, because the derivation is not of type 1 or 2.) The graphical form $(X \oplus \Theta)$ in condition A is called an undetermined point in the derivation, because the final graph in the derivation has not yet been determined. There may be many values of X, (i,j) and Θ which meet conditions A-D. We want to choose the values so that the undetermined point $(X \oplus \Theta)$ is as far right as possible. If

there are several choices which are equally far right, then the decision between the choices is arbitrary. Given this choice, the type of the derivation D is "(X,i,j)".

Clearly there are only a finite number of types of derivations. There is at most one graph generated by derivations of type 1, and at most one graph generated by derivations of type 2. It remains to show that for any type (X,i,j), there are only a finite number of different graphs generated by derivations of this type. The proof has three cases, depending on the value of (i,j).

First, consider a type $(X,1,1)$. The set $L_1(X)$ contains at least two discrete DP-graphs (from condition C). Let m_1 and m_2 be the sizes of two different discrete DP-graphs in $L_1(X)$. There must be some fixed integer n_X, which depends only on X, such that whenever $(X \oplus \Theta)$ is a graphical form in G, then $L_1(\Theta)$ contains at most one discrete DP-graph, and this DP-graph has the fixed number of nodes n_X. (If there were two such sizes, say n_1 and n_2, then G would generate four discrete graphs of sizes m_1+n_1-2, m_1+n_2-2, m_2+n_1-2 and m_2+n_2-2. But, we have already assumed that four such graphs do not exist.)

Consider a derivation $S \overset{*}{\Rightarrow} (X \oplus \Theta) \Rightarrow (\Psi \oplus \Theta) \overset{*}{\Rightarrow} \gamma$ of type $(X,1,1)$, where $(X \oplus \Theta)$ is the rightmost undetermined point of the derivation. Note that $L_1(\Psi)$ contains only one graph – otherwise $(X \oplus \Theta)$ would not be the rightmost undetermined point of the derivation. Let n_Ψ be the number of nodes in this one graph. This implies that the final graph γ has exactly $n_\Psi+n_X-2$ nodes. Since there are only a finite number of choices for Ψ (because there are a finite number of production rules), this implies that there are only a finite number of different graphs that can be derived by a derivation of type $(X,1,1)$.

The cases of $(X,2,3)$ and $(X,3,2)$ derivations are treated similarly. □.

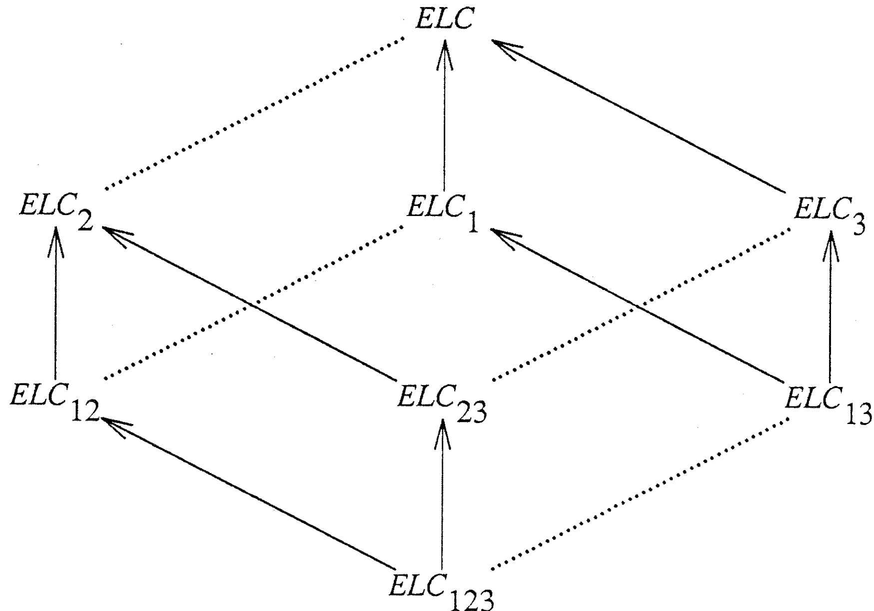
If the connection relation is not complete, then it is possible to generate an infinite language of discrete graphs which does not have four graphs as specified in Lemma 8.6. In particular, the sizes of the discrete graphs generated by Example 2.5 are increasing exponentially. Therefore, this language does not meet the property of Lemma 8.6, and it cannot be generated by an ELC_{13}

grammar. However, Example 2.5 is an ELC_{23} grammar, giving the following result:

Theorem 8.7. *$L(ELC_{13})$ is a proper subclass of $L(ELC_3)$, and $L(ELC_{123})$ is a proper subclass of $L(ELC_{23})$.* □

9. Discussion

We have introduced and studied a new graph grammar model, which is motivated by the previously studied node-label controlled graph grammars and the edge-rewriting systems of Habel and Kreowski. In general, the model has the complete power of recursive enumerability. In order to understand where this power arises, we have studied several obvious restrictions on the model, resulting in the following hierarchy diagram:



The solid arrows indicate proper inclusions of language classes, as shown in Sections 7 and 8. The remaining four dotted lines all correspond to removing restriction 2 (disjoint source and target sets). In one sense, we know that these four dotted lines are also proper inclusions, because it is impossible

to generate the one-node graph in ELC_2. (This is similar to the fact that context-sensitive string grammars cannot generate the empty string without an erasing rule.) However, apart from this special case, we do not currently know whether restriction 2 causes a reduction in power. In particular, if L is an ELC-language, is $L-\{\bullet\}$ always an ELC_2-language? (And similarly for the other three dotted lines). The fact that ELC grammars have the full power of recursive enumerability needs to be taken as a warning. In particular, Rice's theorem implies that every non-trivial question about ELC-languages is undecidable. Hence, in most practical cases, where there is interest in parsing or other questions about the graphs being generated, we will need to work with models that are less powerful than ELC. In our current work we are considering the above three restrictions in this light, as well as the special case where source and target subsets are required to be the entire daughter graph. Here are several other natural research directions that we are continuing to study:

1. Research on the combinatorial properties and decidability properties of the individual subclasses of *ELC* grammars in our diagram.

2. Hopefully the results of the previous questions will help us to single out a central subclass of *ELC* grammars. Once such a central subclass arises, we would like to study this in more depth. A similar plan was followed for node-label control grammars, yielding the tractable subclass of boundary *NLC* grammars [15].

3. We plan to undertake a comparison of *ELC* grammars with other well-known classes of graph grammars. In particular, a careful comparison between *ELC* and *NLC* grammars (also on various sublevels) seems natural, since in both grammar models the replacement mechanism is driven by labels. This should contribute to our understanding of the subtle issue of node-edge duality in the context of graph replacement. Our preliminary research indicates that this is a complex issue which may lead to a consideration of hypergraph grammars.

4. There are also a number of technical issues concerning our model. For example, we do not

currently allow loops in graphs, although loops are allowed in Habel-Kreowski's edge-replacement systems. Is this a critical difference, or can loops be accommodated in the *ELC* model?

References

(1) H. Ehrig, M. Pfender and H.J. Schneider. Graph grammars - an algebraic approach, in: *Proc. Conf. Switch. Automata Theory* (1973), 167-180.

(2) A. Habel and H.-J. Kreowski. On context-free graph languages generated by edge replacement, in: *Graph-Grammars and Their Application to Computer Science*, 2nd International Workshop, (H. Ehrig, M. Nagl and G. Rozenberg, eds), LNCS 153, Springer-Verlag, Berlin (1983), 143-158.

(3) A. Habel and H.-J. Kreowski. Characteristics of graph langages generated by edge replacement, Technical Report, Department of Computer Science, University of Bremen (1985).

(4) M. Harrison, *Introduction to Formal Language Theory*, Addison-Wesley, Reading, MA., (1978).

(5) J. Hoffmann and M.G. Main. Results on NLC grammars with one-letter terminal alphabets, University of Colorado Technical Report Cu-CS-348-86 (September 1986).

(6) D. Janssens and G. Rozenberg. On the structure of node-label controlled graph languages, *Information Sciences* 20 (1980), 191-216.

(7) D. Janssens and G. Rozenberg. Restrictions, extensions and variations of NLC grammars, *Information Sciences* 20 (1980), 217-244.

(8) D. Janssens and G. Rozenberg. Decision problems for node-label controlled graph grammars, *JCSS* 22 (1981), 144-177.

(9) D. Janssens and G. Rozenberg. Graph grammars with neighbourhood controlled embedding, *TCS* 21 (1982), 55-74.

(10) H.C.M. Kleijn, M. Penttonen, G. Rozenberg and K. Salomaa. Direction independent context-sensitive grammars, *Information and Control* 63 (1984), 113-117.

(11) M.G. Main and G. Rozenberg. Handle NLC grammars and languages, University of Colorado Technical Report CU-CS-315-85 (1985), submitted for publication.

(12) M. Nagl. A tutorial and bibliographical survey on graph grammars, in: *Graph-Grammars and Their Application to Computer Science and Biology* (V. Claus, H. Ehrig and G. Rozenberg, eds), LNCS 73, Springer-Verlag, Berlin (1978), 70-126.

(13) W. Ogden, Ross and K. Winklmann, An "Interchange Lemma" for context-free languages, WSU Technical Report CS-81-080, Pullman, WA., (1981).

(14) H.G. Rice. Classes of recursively enumerable sets and their decision problems, *Trans. AMS* 89 (1953), 25-59.

(15) G. Rozenberg and E. Welzl. Boundary NLC graph grammars – basic definitions, normal forms and complexity, *Information and Control* 69 (1986), 136-167.