Handle NLC Grammars and R. E. Languages

MICHAEL G. MAIN

Department of Computer Science, University of Colorado,
Boulder, Colorado 80309

AND

GRZEGORZ ROZENBERG

Institute of Applied Mathematics and Computer Science, University of Leiden,
Leiden, The Netherlands;
and Department of Computer Science, University of Colorado,
Boulder, Colorado 80309

Received October 3, 1985

A graph grammar is a mechanism for generating sets of graphs (called graph languages). This paper examines the generating power of a simple extension of the well studied node-label controlled graph grammars. We show that the extension, called handle NLC grammars, gives the ability to generate any recursively enumerable graph language. The proof proceeds in three steps. First we show how a handle NLC grammar can simulate a phrase-structure string grammar, where the strings that the phrase-structure grammar works on are considered to be graphs of a special form. Then we demonstrate a way of encoding graphs as strings. The final step shows how a handle NLC grammar can convert a string encoding of a graph into the graph itself. © 1987 Academic Press, Inc.

1. INTRODUCTION

Over the past 20 years, several models of "graph grammars" have been introduced as mechanisms for generating sets of graphs (e.g., [2, 5, 6, 16]). Numerous applications in computer science and biology have been studied (e.g., [3, 14, 15]). One particular model is the node-label controlled (NLC) graph grammars ([4, 9, 10, 11, 13], for example). They provide a production-based mechanism for generating graph languages (sets of graphs). In NLC grammars, the left side of each production is required to be a single labeled node. This paper examines the extension of these graph grammars where the left side of each production is a graph of the form \[ A \longrightarrow B \]. The main result shows that the resulting class of graph languages is exactly the recursively enumerable graph languages.

* G. Rozenberg has been supported in part by National Science Foundation Grant MCS-79-03838. M. Main has been supported in part by National Science Foundation Grant DCR-84-02341.

0022-0000/87 $3.00
Copyright © 1987 by Academic Press, Inc.
All rights of reproduction in any form reserved.
Terminology and notation. Throughout the paper, the term graph refers to an
undirected, node-labeled, finite graph with no self-loops or multiple edges, and we
do not distinguish between isomorphic graphs. For a finite alphabet $A$, the set of all
graphs with labels chosen from $A$ is denoted $G_{A}$. If a node $v$ in a graph has a label
$A$, then we call $v$ an $A$-node. The empty set is denoted $\emptyset$ and the empty graph is
written $e$. A handle is a graph with exactly two labeled nodes, and one edge connecting
the nodes.

We assume that the reader is familiar with the rudiments of computability theory
and rewriting systems (such as string grammars).

2. Handle Node-Label Controlled Graph Grammars

The exact extension of NLC grammars which we use is defined here.

**Definition.** A Handle NLC graph grammar (HNLC grammar) is a 5-tuple
$(\Sigma, \Lambda, P, \sigma, C)$, where

- $\Sigma$ is a finite set of node labels;
- $\Lambda \subset \Sigma$ is the set of terminal labels; elements of $\Sigma - \Lambda$ are nonterminal labels;
- $P$ is a finite set of productions; each production has the form

$$A \xrightarrow{\eta} B := \alpha,$$

where $A, B \in \Sigma$ (at least one nonterminal) and $\alpha \in G_{\Sigma}$;

- $\sigma \in G_{\Sigma}$ is the start graph;
- $C \subseteq \Sigma \times \Sigma$ is the connection relation.

The way that an HNLC grammar generates a set of graphs (its graph language)
is based on the mechanism of ordinary NLC grammars. Here is a somewhat informal
description of this generation process. Let $G = (\Sigma, \Lambda, P, \sigma, C)$ be an HNLC
grammar. A production $A \xrightarrow{\eta} B := \alpha$ of $P$ transforms a graph in the following way.

**Step 1.** Start with a graph $\mu$; within $\mu$, find a specific occurrence of the handle
$A \xrightarrow{\eta} B$, which is called the mother handle. The set of nodes which are directly connected
to the mother handle is called the neighborhood.

**Step 2.** Delete the mother handle (and its incident edges) from the graph $\mu$.
Call the resulting graph $\mu'$.

**Step 3.** Add to $\mu'$ a disjoint copy of $\alpha$. The new occurrence of $\alpha$ is called the
daughter graph.

**Step 4.** For each pair $(Y, Z)$ in the connection relation $C$, connect every
$Y$-node in the daughter graph with every $Z$-node in the neighborhood. Call the
resulting graph $\eta$. 

We write $\mu \Rightarrow G \eta$ to denote the relation "$\eta$ is directly derived from $\mu$ in $G$." If there exists a finite sequence of transformations

$$\mu_0 \Rightarrow G \mu_1 \Rightarrow G \cdots \Rightarrow G \mu_m$$

then we write $\mu_0 \Rightarrow^* G \mu_m$ and say that $\mu_m$ is derived from $\mu_0$ in $G$; the finite sequence is called a derivation of length $m$. The language generated by the grammar $G$, also called an HNLC language, is the set of all graphs in $G_d$ which can be derived from the graph $\sigma$; that is $L(G) = \{ \mu \in G_d \mid \sigma \Rightarrow G \mu \}$. When the grammar $G$ is understood, the notation will be simplified to $\Rightarrow$ or $\Rightarrow^*$.

**Example.** Suppose we have a production $\chi = a \rightarrow X \chi$ in an HNLC grammar with connection relation $C = \{(a, a)\}$. We can apply this production to the graph $\bullet \overrightarrow{a} \bullet \overrightarrow{X} \bullet$ in two ways. If the mother handle includes the leftmost $Z$, then the result is

$$\bullet \overrightarrow{a} \bullet \overrightarrow{X} \bullet \overrightarrow{Y} \bullet$$

with the new daughter graph as indicated. On the other hand, if we choose the rightmost $Z$, then the result is

$$\bullet \overrightarrow{a} \bullet \overrightarrow{Y} \bullet \overrightarrow{X} \bullet \overrightarrow{Z}$$

**Example.** Consider $G = (\Sigma, A, P, \sigma, C)$, where $\Sigma = \{ W, X, Y, Z, a \}$, $A = \{ a \}$, $C = \{(a, a), (Y, X), (W, X), (W, a)\}$, the start graph $\sigma$ is

$$\bullet \overrightarrow{a} \bullet \overrightarrow{X} \bullet \overrightarrow{Y} \bullet \overrightarrow{Z}$$

and $P$ has these three productions:

$$\chi = a \rightarrow X \chi, \quad \chi = Z \rightarrow W, \quad \chi = W \rightarrow a$$

The grammar generates "circles" of the form

$$\bullet \overrightarrow{a} \bullet \overrightarrow{X} \bullet \overrightarrow{Y} \bullet \overrightarrow{Z}$$

with three or more nodes. (This cannot be done by an NLC grammar; see [4].)

A useful extension of HNLC grammars is to allow each production to have an individual connection relation, rather than providing only one global connection relation, which all productions must use. This extension does not increase the language-generating capability of HNLC grammars. That is, an HNLC grammar with individual connection relations can be converted to a usual HNLC grammar,
without affecting the language that is generated. The algorithm for this conversion is identical to the method given for the same conversion on NLC grammars [10], and hence not given here.

3. RELATIONSHIP BETWEEN HNLC LANGUAGES AND STRING LANGUAGES

One way to represent a string $x = x_1 x_2 \cdots x_n$ as a graph is by

$$\bullet \quad \cdots \quad \cdots \quad \bullet$$

The $\bullet$ is a special "endmarker" symbol which does not appear in the alphabet of $x$. A graph of this sort is called a string-graph and is denoted by $x^s$ (see [12] for a similar method). If $L \subseteq A^*$, then the graph language associated with $L$ is $L^s = \{ x^s \mid x \in L \}$. Clearly, if $L^s$ is an HNLC language, then $L$ is recursively enumerable. The following theorem shows that the converse also holds.

**Theorem.** A string language $L$ is recursively enumerable if and only if $L^s$ is an HNLC language.

**Proof.** It is clear that $L$ is recursively enumerable whenever $L^s$ is an HNLC graph language. For the other half of the proof, suppose that $L \subseteq A^*$ is a recursively enumerable string language. (With $\varepsilon$ not in $L$; the case where $\varepsilon$ is in $L$ is a simple extension.) Then there is a phrase-structure string grammar $G$ which generates $L$, and the rules of $G$ all have the form $AB := w$ (with $w$ nonempty) [8]. Let $H$ be the following HNLC grammar: the labels of $H$ are $\Gamma \cup \bar{\Gamma} \cup {\{L, \&}, S}$, where $\bar{\Gamma}$ and $\bar{\Gamma}$ are new copies of $\Gamma$ with barred and double-barred symbols. (We assume all of these sets are disjoint.) The terminal labels of $H$ are the terminal labels of $G$ plus $\bullet$. The start graph for $H$ is $\bullet \quad \cdots \quad \cdots \quad \bullet$, where $S_0 S_1 \cdots S_{n-1} S_n$ is the start string of $G$. The productions of $H$ have individual connection relations, as follows:

1. For each production $AB := Y_1 \cdots Y_k$ of $G$, there is an $H$-production $\cdot \quad \cdots \quad \cdots \quad \bullet$, with connection relation $L \times (\Gamma \cup \{L\})$ and $\bar{\Gamma} \times (\bar{\Gamma} \cup \{\&\})$.

2. For each pair of symbols $X, Y \in \Gamma$, there are two $H$-productions $\cdot \quad \cdots \quad \cdots \quad \bullet$ (with connection relation $\bar{\Gamma} \times \Gamma \cup \{L\}$ and $\bar{\Gamma} \times \bar{\Gamma} \cup \{\&\}$), and $\cdot \quad \cdots \quad \cdots \quad \bullet$ (with connection relation $\Gamma \times \Gamma \cup \{L\}$ and $\bar{\Gamma} \times \bar{\Gamma} \cup \{\&\}$).

3. For each $X \in \Gamma$ there is an $H$-production $\cdot \quad \cdots \quad \cdots \quad \bullet$, with all possible pairs in the connection relation.

4. For each symbol $X \in \Gamma$ there is an $H$-production $\cdot \quad \cdots \quad \cdots \quad \bullet$, with connection relation $\Gamma \times \Gamma$.

Note that if $x \Rightarrow_G y$ is a derivation step in the string grammar $G$, then $\cdot \quad \cdots \quad \cdots \quad \bullet \quad \cdots \quad \bullet \quad \cdots \quad \bullet$ (where $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_m$). This implies that $L^s$ is a subset of $L(H)$. The proof that $L(H)$ is also a subset of $L^s$.
is an induction on the length of a derivation in $H$. The induction hypotheses is: if $\sigma \Rightarrow_H \gamma$ is a derivation of length $j$ then $\gamma$ is a graph of the form

$$x_1 \rightarrow \cdots \rightarrow x_n,$$

where the string $x = x_1 x_2 \cdots x_n$ has these properties:

1. The last character is either $\$ \text{ or } \&, and these two characters do not appear elsewhere in the string.
2. There is at most one barred character, which appears after every $L$ and unbarred character.
3. Every double barred character appears after the barred character.
4. If all the characters are made unbarred, and the characters $L$, $\$, and $\&$, are removed, then the result is a sentential form from $G$.

The induction on $j$ is a case-analysis of the productions of $H$. Therefore, $L(H) = L^\$, as required for the theorem.

4. Encoding and Decoding Graphs

Let $A$ be some fixed alphabet of node labels. This section gives one way to encode a graph $\alpha \in G_A$ as a string. The key point of the encoding is that “decoding” can be carried out by an HNLC grammar, also given in this section.

To encode a graph $\alpha$, we first fix some linear ordering on $\alpha$'s nodes, so we can talk about the first, the last, or the $i$th node of $\alpha$. Then we give two strings.


terms(\alpha) = X_1 X_2 \cdots X_n, \text{ where } n \text{ is the number of nodes in } \alpha \text{ and } X_i \text{ is the label of the } i \text{th node of } \alpha.

instructions(\alpha): \text{ this is a string of "instructions" telling how to connect the nodes of } \alpha. \text{ Of course, there are many different methods for giving such instructions. The method described below uses the alphabet } \{L, R, S, E, \!\} \text{ (which is disjoint from the label alphabet of } \alpha).\n
The instruction string assumes that we have been given a linear ordering of the labeled nodes of $\alpha$, referred to as the node sequence. Associated with the node sequence is a node pointer, which at the beginning points to the last node of the sequence. The instruction string can be "executed" by reading its characters one at a time, from left-to-right. Each time a character is read, some action is carried out, such as rearranging the node sequence, moving the node pointer, or establishing edges between the nodes in the node sequence. The actions for each instruction character are the following.

$L$ — Move the node pointer left one spot in the sequence.
$R$ — Move the node pointer right one spot in the sequence.
$E$ — Establish an edge between the node which is currently pointed at and the node which follows it in the sequence.
$S$ — This causes the node sequence to change. Specifically, the node which is currently pointed to by the node pointer is swapped with the node which follows the node pointer in the current ordering. The node pointer continues to point to the same position in the sequence (which is now a new node).

$!$ — This character is not actually an instruction like the other characters. Instead, it is an end-of-string marker which will only appear at the end of the instruction string, to indicate that “processing” is completed. The purpose of this will become clear later.

**Example 4.1.** Let $\alpha$ be the graph

![Graph](image)

One way to encode $\alpha$ is the following:

$$\text{labels}(\alpha) = AABC,$$

and

$$\text{instructions}(\alpha) = LLERSLELi!$$

These instructions would be followed, step-by-step:

<table>
<thead>
<tr>
<th>Node Sequence with its pointer</th>
<th>Remaining Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Start:</strong> $ABC$</td>
<td>$LLERSLELi!$</td>
</tr>
<tr>
<td>After executing $LL$: $ABC$</td>
<td>$ERSLELi!$</td>
</tr>
<tr>
<td>After executing $E$: $ABC$</td>
<td>$RSLELi!$</td>
</tr>
<tr>
<td>After executing $E$: $ABC$</td>
<td>$SLELi!$</td>
</tr>
<tr>
<td>After executing $S$: $ABC$</td>
<td>$LELi!$</td>
</tr>
<tr>
<td>After executing $L$: $ABC$</td>
<td>$ELi!$</td>
</tr>
<tr>
<td>After executing $E$: $ABC$</td>
<td>$Li!$</td>
</tr>
<tr>
<td>After executing $L$: $ABC$</td>
<td>$i!$</td>
</tr>
</tbody>
</table>
After the last instruction is executed, the remaining graph is exactly \( \alpha \); thus the pair of strings \( \langle AABC, LLERS\overline{SE}EL \rangle \) encodes the graph \( \alpha \).

Notice how this example leaves the "pointer" at the first node in the current linear ordering; in general, we will assume that \( \text{instructions}(\alpha) \) always does this (which will simplify things later). For the special case of the empty graph, we define \( \text{labels}(\alpha) \) to be the empty string and \( \text{instructions}(\alpha) \) to be the string \( \dagger \).

Our reason for giving this method of encoding graphs is that it is fairly simple to design an HINLC grammar, denoted \( D_{\alpha} \), which can "decode" this encoding. Specifically, for any graph \( \alpha \in G_{\alpha} \) the grammar \( D_{\alpha} \) can derive \( \alpha \) (and no other graph) from the starting string-graph \( \langle \text{labels}(\alpha) \quad \text{instructions}(\alpha) \rangle^S \). For example, \( D_{\alpha} \) will take the graph

\[
A \quad A \quad B \quad C \quad E \quad E \quad E \quad \bar{R} \quad S \quad L \quad E \quad E \quad \dagger \quad S
\]

and produce exactly one terminal graph—the graph \( \alpha \) from Example 4.1. The label set used by \( D_{\alpha} \) is the union of sets

\[
\{ S, Z, Q, L_1, R_1, S_1, S_2, S_3 \}, \quad I = \{ L, R, S, E \},
\]

\[
I = \{ \bar{L}, \bar{R}, \bar{S}, \bar{E} \}, \quad \Lambda, \bar{\Lambda}, \bar{\bar{\Lambda}}, \bar{\bar{\bar{\Lambda}}},
\]

where \( \bar{\bar{\bar{\Lambda}}} \) contains a new symbol \( \bar{\bar{\bar{\Lambda}}} \) for each \( X \in \Lambda \) (and similarly for \( \bar{\Lambda}, \bar{\bar{\Lambda}}, \) and \( \bar{\bar{\bar{\Lambda}}} \)). As usual, all these symbols are distinct. The following steps are the basic mechanism that \( D_{\alpha} \) employs to derive a graph \( \alpha \) from its encoding.

(1) The right end of the starting string-graph looks like the following: \( \ldots \longrightarrow \dagger \). The \( \dagger \) marks the end of \( \text{instructions}(\alpha) \), and the \( S \) is provided by the mechanism which converts a string to a graph. At the beginning of the derivation, the \( S \) will move to the left through the barred instruction symbols, removing the bars from each of them. When the last bar is removed, the \( S \) is also removed, leaving a graph of the form

\[
\begin{array}{c}
\text{labels}(\alpha) \\
\downarrow \text{instructions}'(\alpha)
\end{array}
\]

where \( \text{instructions}'(\alpha) \) results from \( \text{instructions}(\alpha) \) by removing bars. Thus, if \( \alpha \) is the graph from Example 4.1, then we get

\[
A \quad A \quad B \quad C \\
L \quad E \quad E \quad \bar{R} \quad S \quad L \quad E \quad E \quad \dagger
\]
(The purpose of this step is to ensure that no derivation can start unless a \( S \) appears at the end of the graph; this is required later.)

(2) The first instruction from the instruction string is now "executed." Movements of the "pointer" are accomplished by moving the edge which connects \( \text{labels} \ (z) \) with the rest of the graph. Continuing with Example 4.1, the derivation will reach this graph after executing the first \( L \) instruction:

\[
\begin{array}{c}
A \quad A \quad B \quad C \\
L \quad E \quad R \quad S \quad L \quad E \quad L \\
\end{array}
\]

Notice how the nodes to the right of the pointer change to \( \tilde{A} \) symbols in order to keep the orientation of \( \text{labels} \ (z) \). Also, during execution of the \( L \)-instruction, the corresponding \( L \)-node was erased from the graph. The exact mechanism for executing this instruction and the others will be defined in a moment.

(3) The next instruction is executed, and so on, until eventually \( ! \) is reached. For our example, when \( ! \) is reached, the graph looks like the following:

\[
\begin{array}{c}
A \quad A \quad C \quad B \\
\end{array}
\]

The \( Z \)-nodes in the graph are a mechanism for marking where final edges occur. Specifically, when the derivation finally ends, any two nodes which are mutually adjacent to a \( Z \)-node at this point will be connected; other nodes will not be connected. This placement of edges is instigated by the \( ! \)-instruction. The final result is the graph \( z \):

\[
\begin{array}{c}
A \quad A \quad C \\
\end{array}
\]

During this last stage the labels \( \tilde{A} \) and \( Q \) will be used. Most of the other symbols are used as temporaries while executing instructions.

Table I lists the productions of the HNLC grammar we have been describing. Each production has an individual connection relation, as described at the end of Section 2. In the connection relations, \( \Sigma \) is the entire set of labels for the grammar.
TABLE I
An Example Derivation, Starting with the Encoding of the Graph 2

<table>
<thead>
<tr>
<th>Production</th>
<th>Connection relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1. Productions to remove the bars from the instructions, and to delete the $</td>
<td></td>
</tr>
<tr>
<td>A. For each $X \in I$:</td>
<td>${X} \times I$ $\cup$ ${S} \times I$ $\cup$ ${S} \times A$</td>
</tr>
<tr>
<td>B. For each $A \in A$:</td>
<td>${A} \times \Sigma$</td>
</tr>
<tr>
<td>P2. Productions to execute $L$ instruction</td>
<td></td>
</tr>
<tr>
<td>A. For each $A \in A$:</td>
<td>${A} \times (A \cup \bar{A} \cup {Z}) \cup {L_1} \times (I \cup A)$</td>
</tr>
<tr>
<td>B. For each $B \in A$:</td>
<td>${B} \times \Sigma$</td>
</tr>
<tr>
<td>P3. Productions to execute $R$ instruction</td>
<td></td>
</tr>
<tr>
<td>A. For each $A \in A$:</td>
<td>${A} \times (A \cup \bar{A} \cup {Z}) \cup {R_1} \times (I \cup \bar{A})$</td>
</tr>
<tr>
<td>B. For each $B \in A$:</td>
<td>${B} \times \Sigma$</td>
</tr>
<tr>
<td>P4. Productions to execute $E$ instruction</td>
<td></td>
</tr>
<tr>
<td>For each $A \in A$:</td>
<td>${A} \times (A \cup \bar{A} \cup {Z} \cup I) \cup {Z} \times \bar{A}$</td>
</tr>
</tbody>
</table>

(Table continued)
### TABLE I—Continued

<table>
<thead>
<tr>
<th>Production</th>
<th>Connection relation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>P5. Productions to execute $S$ instruction</strong></td>
<td></td>
</tr>
</tbody>
</table>

A. For each $A \in A$:

```
   A
 \rightarrow S
```

\[
\{ \bar{A} \} \times (\bar{A} \cup \{Z\}) \cup \{ S_1 \} \times (I \cup A \times \bar{A})
\]

B. For each $B \in A$:

```
   B
 \rightarrow S
```

\[
\{ \bar{B} \} \times (\bar{A} \cup \{Z\}) \cup \{ S_2 \} \times (I \cup A \cup \bar{A} \cup \bar{A})
\]

C. For each $A \in A$:

```
   A
 \rightarrow S
```

\[
\{ \bar{A} \} \times (\bar{A} \cup \bar{A} \cup \{Z\}) \cup \{ S_3 \} \cup \{ S_4 \} \times (I \cup A \cup \bar{A})
\]

D. For each $B \in A$:

```
   B
 \rightarrow S
```

\[
\{ B \} \times (\bar{B} \cup \bar{A} \cup I \cup \{Z\})
\]

<table>
<thead>
<tr>
<th><strong>P6. Production to handle special case of the empty graph</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T \rightarrow S := e$</td>
</tr>
<tr>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>P7. Productions to “clean up” at the end of the derivation</strong></th>
</tr>
</thead>
</table>

A. For each $A \in A$:

```
   A or \bar{A}
 \rightarrow S
```

\[
\{ \bar{A} \} \times (A \cup \bar{A} \cup \{Z\}) \cup \{!\} \times \bar{A}
\]

B. For each $A \in A$:

```
   A or \bar{A}
 \rightarrow S
```

\[
\{ \bar{A} \} \times (A \cup \bar{A} \cup \{Z\})
\]

C. For each $A \in A$:

```
   \bar{A}
 \rightarrow S
```

\[
\{ \bar{A} \} \times \Sigma
\]

D. For each $A \in A$:

```
   \bar{A}
 \rightarrow S
```

\[
\{ A \} \times \Sigma
\]
Figure 4.1 shows an example derivation, starting with the encoding of the graph \( \alpha \) from Example 4.1. Notice that the final result is the graph \( \alpha \).

Start graph:
\[
A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G \rightarrow H \rightarrow I \rightarrow J
\]

\( \Rightarrow \) via P1-A:
\[
A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow S \rightarrow L \rightarrow E \rightarrow R \rightarrow S \rightarrow L \rightarrow E \rightarrow L
\]

\( \Rightarrow \) via P1-A:
\[
A \rightarrow B \rightarrow C \rightarrow S \rightarrow L \rightarrow E \rightarrow R \rightarrow S \rightarrow L \rightarrow E \rightarrow L
\]

\( \Rightarrow \) via P1-B:
\[
A \rightarrow B \rightarrow C \rightarrow S \rightarrow L \rightarrow E \rightarrow R \rightarrow S \rightarrow L \rightarrow E \rightarrow L
\]

\( \Rightarrow \) via P2-A:
\[
A \rightarrow B \rightarrow C
\]

\( \Rightarrow \) via P2-B:
\[
A \rightarrow B \rightarrow C
\]

\( \Rightarrow \) via P2-A and P2-B:
\[
A \rightarrow B \rightarrow C
\]

\( \Rightarrow \) via P3-A and P3-B:
\[
A \rightarrow B \rightarrow C
\]

\( \Rightarrow \) via P5-A:
\[
A \rightarrow B \rightarrow C
\]

\( \Rightarrow \) via P5-B:
\[
A \rightarrow B \rightarrow C
\]

\( \Rightarrow \) via P5-C:
\[
A \rightarrow B \rightarrow C
\]

\( \Rightarrow \) via P5-D:
\[
A \rightarrow B \rightarrow C
\]

\( \Rightarrow \) via P7-A:
\[
A \rightarrow C \rightarrow B
\]

\( \Rightarrow \) via P7-A:
\[
A \rightarrow C \rightarrow B
\]

\( \Rightarrow \) via P7-A:
\[
A \rightarrow C \rightarrow B
\]
For any graph $\alpha$, we define $\text{encode}(\alpha)$ to be the string resulting from the concatenation of $\text{labels}(\alpha)$ and $\text{instructions}(\alpha)$. The following theorem, which follows from the definition of $D_\alpha$, indicates how the grammar $D_\alpha$ can "decode" this encoding.

**Theorem.** Let $\alpha$ be any graph from $G_\alpha$. Then the only terminal graph generated by $D_\alpha$ starting with $(\text{encode}(\alpha))^S$ is $\alpha$.

5. **RECURSIVELY ENUMERABLE GRAPH LANGUAGES**

For any graph $\alpha$, let $\text{encode}(\alpha)$ be the encoding of $\alpha$ as the string $\text{labels}(\alpha)\text{instructions}(\alpha)$, as indicated in the previous section. A graph language $L$ is called recursively enumerable if and only if the string language $\text{encode}(L) = \{\text{encode}(\alpha) \mid \alpha \in L\}$ is a recursively enumerable string language.

An application of Church's thesis tells us that each HNLC graph language is recursively enumerable. The constructions of the previous two sections imply that the converse also holds, which will be shown now.

**Theorem.** A graph language is recursively enumerable if and only if it is an HNLC language.

**Proof.** Given an HNLC language $L$, it is easy to build a Turing machine which enumerates $\text{encode}(L)$. For the other half of the proof, let $L$ be a recursively enumerable graph language over an alphabet $\Delta$. By definition, $\text{encode}(L)$ is a recursively enumerable string language. Thus, using the technique of Section 3, we can construct a graph grammar $H$ which generates exactly $(\text{encode}(L))^S = \{(\text{encode}(\alpha))^S \mid \alpha \in L\}$. If we add $H$ to the productions of $D_\alpha$ (from Section 4), then for each $\alpha \in L$, there is a derivation

$$(\text{Start graph of } H) \xrightarrow{H} (\text{encode}(\alpha))^S \xrightarrow{D_\alpha} \alpha.$$ 

Notice that these are the only derivations of graphs over the terminal alphabet $\Delta$. This follows because the $D_\alpha$ productions cannot begin until the $S$ label appears (and this is always the last step of a derivation by $H$), hence the productions of $D_\alpha$ cannot interfere with those of $H$. Thus, the combined grammar, with rules from $H$ and $D_\alpha$ (and using $H$'s start graph and terminal alphabet $\Delta$) generates exactly $L$. $\blacksquare$

**Note.** The definition of recursively enumerable which we have used makes reference to the method of the previous section for encoding graphs. However, any other effective method for encoding graphs gives the same class of recursively enumerable graph languages. In particular, any encoding that can be computed from our encoding would not change the definition of a recursively enumerable graph language.
6. Discussion

Within the theory of graph grammars, many graph generating mechanisms have been studied. In general, a production in a graph grammar is of the form $\alpha := \beta$, where $\alpha$ and $\beta$ are graphs. A specific case where $\alpha$ is a one-node graph (hence as simple as possible) has been studied thoroughly within the framework of NLC grammars.

The present paper extends the mechanism of NLC grammars to the rewriting of handles rather than single nodes only. We have shown that this simple extension yields "too much" generative power, i.e., all recursively enumerable graph languages. Two other graph generating mechanisms have been shown to have similar power [1,17]; the advantage of HNLC grammars is that it is a simple extension of the well-studied NLC grammars.

A graph rewriting mechanism which is related to HNLC grammars is the edge-replacement system studied by Habel and Kreowski [7]. This model does not yield the power of recursive enumerability, because it uses a very restrictive mechanism to embed a daughter graph. A natural follow-up of our research is to bridge the Habel–Kreowski approach with our own. Some questions in this direction are:

1. What restrictions on an HNLC grammar can one impose to get a "reasonable" generative power (e.g., restrictions on the connection relation, restrictions on the allowable production)?

2. What happens if we vary the method of embedding the daughter graph by distinguishing different parts of the neighborhood? (This is methodologically close to the approaches from [7, 16].)

We are currently working in this direction and hope to report on our research in the future.

Acknowledgments

We thank H. J. Kreowski and one of the referees who both pointed out how to simplify the proof in Section 3. Our original proof, based on Turing machines, was longer and did not show the strong connection with phrase-structure grammars.

References