平成 10 年度

筑波大学第三学群情報学類

卒業研究論文

題目：ビジュアルシステム恵比寿における 視覚的表現を用いた制約の入力

主専攻	情報科学
著者名	藤山 健一郎
指導教員	電子・情報工学系 田中 二郎

要旨

図形言語の文法を定義することによって任意の図形言語を解析する Spatial Parser を生成し、さらに実行することができるビジュアルシステム「恵比寿」が開発されている。

本論文では恵比寿が、２次元的な情報をもつ図形言語を１次元的なテキストで入力するために生じるいくつかの問題点と、視覚的表現を用いることでこの問題を解決できることについて述べる。特に恵比寿の文法定義における「制約」の入力に注目し、この入力方法について考察をおこない、視覚的な表現を利用しインタラクティブな編集を行う、よりわかりやすい入力インターフェイスを提案する。またこの入力インターフェイスを恵比寿に実装したシステム「エビ chu」を作成する。さらにこの「エビ chu」を用いて実際にビジュアルシステムを作成する例をあげ、その作成過程を通してシステムの評価について述べる。

目次

1 序論 **3**

2 準備 **4**
2.1 用語の定義 4
2.2 ビジュアルシステム恵比寿 4
2.2.1 拡張された CMG 4
2.2.2 恵比寿の従来の CMG 入力法について 6

3 CMG 入力法の考察 **9**
3.1 構成要素の入力 9
3.2 構成要素の種類の表示 10
3.3 構成要素の種類の変更 12
3.4 制約の自動生成 13
3.5 制約の編集 14
3.5.1 制約の入力 14
3.5.2 制約の削除 17
3.6 制約の視覚化 18
3.7 識別 ID の整合性 19

4 システム「エビ chu」の実装 **20**
4.1 実装方法 20
4.1.1 使用言語 20
4.1.2 システム構成 20
4.1.3 画面構成 21
4.2 データ構造 22
4.3 考察の反映 23
4.3.1 構成要素の入力 23
4.3.2 構成要素の種類の表示と変更 23
4.3.3 制約の自動生成 24
4.3.4 制約の編集 24
4.3.5 識別 ID の整合性 25

5 システム「エビ chu」の実行例と評価 **27**
5.1 計算の木 27
5.2 ビジュアルシステム：計算の木の作成 29
5.2.1 生成規則 1 の定義 29

5.2.2 生成規則 2 の定義 33
5.3 評価 36

6 結論 40

謝辞 41

参考文献 42

第 1 章

序論

テキストを用いた従来のプログラミング言語では、テキスト表現の制限から逐次的にアルゴリズムを記述するプログラミング方法が適する。しかしながら、現在プログラミングパラダイムが変化するにつれ、テキストによる表現力の限界が指摘されている。

そこで近年、新たなプログラミング手法として、人間が理解しやすい図形などの視覚的な表現を用いたビジュアルプログラミングが注目を集めている。これまでに、いくつかのビジュアルプログラミングシステム [1][2][3][4][5][6] が提供されており、また研究されている。このビジュアルプログラミングシステムの研究では対象となるものが図形を用いた言語（図形言語）であるため、これを実装する際必要となるのが（テキスト言語における 文、単語にあたる図形文、図形単語よりなる）図形言語の解析をおこなう Spatial Parser である。しかし個々のビジュアルプログラミングシステム毎に Spatial Parser を実装するのは困難で時間のかかる作業であった。そこで図形言語の文法を定義することにより、この Spatial Parser を生成し、さまざまな図形言語を解析し、さらに実行することができるビジュアルシステム「恵比寿」 [7][8][9][10] が開発されている。

CMG[11] をベースに、 action の概念を追加することにより、プログラムの実行を可能としたものである拡張 CMG[7] を用いて恵比寿では図形言語の文法を記述し定義する。ここで図形言語を記述するというのは、すなわち図形間の関係を記述することであるが、これは扱う情報が 2 次元上の図形的なものであることを意味する。しかしながら恵比寿ではこれを 1 次元的なテキストで入力するため直感的に理解しにくいという欠点があった。そこで図形間の関係を表す CMG をテキストではなく図形を用いて入力すれば、この欠点を解消できると思われる。

本論文では恵比寿の文法定義における「制約」に特に注目し、この入力方法について考察し、新しく視覚的な表現を利用したインタラクティブな編集を行う よりわかりやすい入力インターフェイスを提案する。またこの入力インターフェイスを恵比寿に実装し、その評価について述べる。

本論文の構成は次のとおりである。まず第 2 章では本論文で用いる用語の定義や、ビジュアルシステム恵比寿についての基本的な知識とその問題点について述べる。第 3 章ではその問題点を解消すべく制約の新しい入力方法についての考察を行う。第 4 章では考察に基づいたインターフェイスを恵比寿上に実装したシステムについて述べ、次の第 5 章でそのシステムを用いて実際にビジュアルシステムを作成し、元の恵比寿と比較することによって評価する。

第 2 章

準備

本章ではまず本論文で用いる用語の定義を行う。次にビジュアルシステム恵比寿について述べ、その問題点について考察する。

2.1　用語の定義

単語を構成するために通常のテキスト言語では文字を一次元に配置していく。これに対してビジュアル言語では円や直線などの図形を主に二次元、もしくはそれ以上の次元に配置する。この円や直線といった基本的な図形を図形文字と定義する。各図形文字は、種類、色、大きさ、位置などといった属性をもつ。図形文字はシステムが提供する最も基本的な図形であり、それ以上分解できないので終端記号（Terminal）とも呼ぶ

テキストではある概念をいくつかの文字からなる文字列である単語として表すが、ビジュアル言語では、いくつかの図形文字を組み合わせたシンボルとして表すのが一般的である。このようなシンボルを図形単語と定義する。図形単語を構成するいくつかの図形を構成要素と定義する。図形単語もまた属性をもつ。図形単語を組み合わせて構成される、テキスト言語の文に相当するものを図形文と定義する。ここで図形文に現れるのは正確には図形単語のインスタンスであることに注意する必要がある。本論文では図形文に現れる図形単語のインスタンスをトークンと定義する。図形単語や図形文は終端記号ではないので、まとめて非終端記号（Non Terminal）とも呼ぶ。

本論文ではビジュアル言語を処理するシステムをビジュアルシステムと定義する。

2.2　ビジュアルシステム恵比寿

ビジュアルシステム恵比寿は図形言語を利用したビジュアルシステムを生成することができるシステムである。つまり、ある文法に基づいて描画された図形を Spatial Pasing することによって解析を行い、処理を行うことができる。（図 2.1）

2.2.1　拡張された CMG

恵比寿では図形言語の定義をするために拡張 CMG[7] を用いて記述する。拡張 CMG とは、図形間の関係を記述する CMG[11] をベースに、生成規則が適用された時に図形の書き換えなどといった action も記述できるようにしたものである。これによって、図形言語の解析だけではなく、その結果を利用してなんらかの処理を行うことも可能としている。また、生成規則の決定性を高めるために not_exist、 all といった構成要素も追加している。

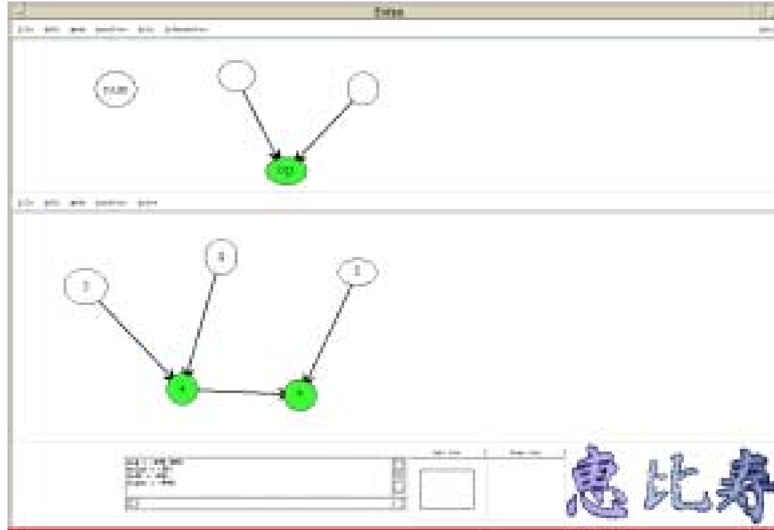

図 2.1: 恵比寿の実行画面

文献 [11] における CMG の生成規則は以下の用になる。

$$
\begin{aligned}
T(\vec{x}) \quad ::= \quad & T_1(\vec{x_1}), \cdots, T_n(\vec{x_n}) \ where \\
& exists \ T'_1(\vec{x'_1}), \cdots, T'_m(\vec{x'_m}) \\
& where \ C \ and \ \vec{x} = F
\end{aligned}
$$

そして拡張 CMG の生成規則は以下のようになる。

$$
\begin{aligned}
T(\vec{x}) \quad ::= \quad & T_1(\vec{x_1}), \cdots, T_n(\vec{x_n}) \ where \\
& exists \ T'_1(\vec{x'_1}), \cdots, T'_m(\vec{x'_m}) \\
& not \ exists \ T''_1(\vec{x''_1}), \cdots, T''_l(\vec{x''_l}) \\
& all \ T'''_1(\vec{x'''_1}), \cdots, T'''_k(\vec{x'''_k}) \\
& where \ C \ and \ \vec{x} = F \ and \ Action
\end{aligned}
$$

ここで $T(\vec{x})$ は生成される図形単語である。また $T_1(\vec{x_1}), ..., T_n(\vec{x_n})$ は n 個の図形単語、もしくは図形文字であり、 normal の構成要素と呼ぶ。実際の非終端記号を構成する部品となる、すなわち書き換えされる対象となるものである。 $T'_1(\vec{x'_1}), ..., T'_m(\vec{x'_m})$ も m 個の図形単語、もしくは図形文字であり、 exist の構成要素と呼ぶ。図のどこかに存在する他のトークンと normal の構成要素との間になりたつ制約を記述するために用いられる。すなわち、このトークンが存在しなければ生成規則は適用されない。 $T''_1(\vec{x''_1}), ..., T''_l(\vec{x''_l})$ は l 個の図形単語、もしくは図形文字であり、 not_exist の構成要素と呼ぶ。 exist と同じく、生成規則の決定性を増すために用いられる。このトークンが存在するときは生成規則は適用されない。 $T'''_1(\vec{x'''_1}), ..., T'''_k(\vec{x'''_k})$ は k 個の図形単語、もしくは図形文字であり、 all の構成要素と呼ぶ。これは図形文の中にある指定された種類の全てのトークンからなるマルチセットを作るために用いられる。つまり同一種類の複数のトークンを指定する際に使用する。 C は属性 $\vec{x_1}, \cdots, \vec{x_n}$、 $\vec{x'_1}, \cdots, \vec{x'_m}$、 $\vec{x'''_1}, \cdots, \vec{x'''_l}$ に関する制約の連接である。制約とは、２つの属性間、もしくは１つの属性と定数の間になんらかの条件を課すことである。 F は属性 $\vec{x_1}, \cdots, \vec{x_n}$、 $\vec{x'_1}, \cdots, \vec{x'_m}$、 $\vec{x'''_1}, \cdots, \vec{x'''_l}$ を引数とする関数であり、生成される非終端記号 $T(\vec{x})$ の属性 $\vec{x}$ を定義している。 C と F に not_exist の属性 $\vec{x''_1}, \cdots, \vec{x''_l}$ が無い

のは、生成規則が適用されるためには、 not_exist の構成要素が在ってはならないからである。 *Action* はアクションを記述したものである。アクションとは「生成規則が適用されたときにスクリプト言語のプログラムとして実行される文字列」と定義される。たとえば値の計算、新たな図形の生成、変更、削除などが可能である。

なお 5.1節に CMG の例が載っているので参照にされたい。

2.2.2 恵比寿の従来の CMG 入力法について

従来の恵比寿で図形言語、すなわち生成規則を定義するには、以下のような手法を用いる。

まず、図 2.2 のように図形を描くことによって大まかな文法と構成要素を与える。同時に恵比寿がその描画された図から、簡単な制約をある程度自動生成する。このあと CMG 入力ウィンドウが開かれるが、ここには入力された構成要素と自動生成された制約が既にテキストで書き出されている。（図 2.3）

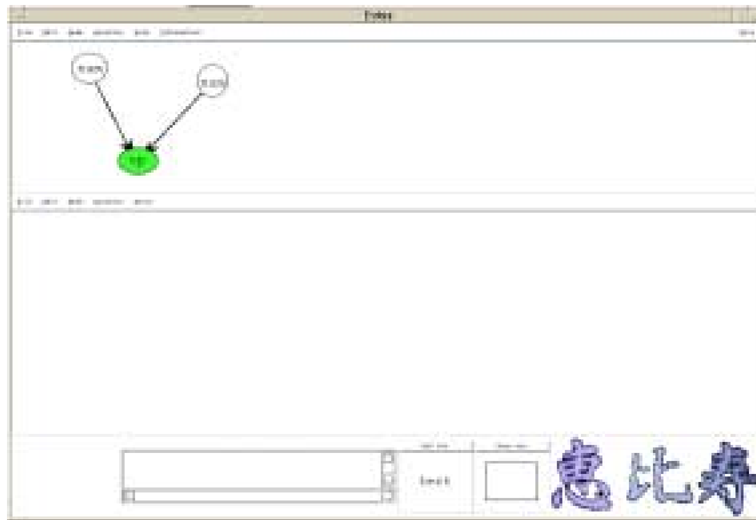

図 2.2: 恵比寿の CMG 入力法１

図 2.4はその CMG ウィンドウを拡大したものだが、上から順に トークン名（$T(\vec{x})$）、属性（$\vec{x} = F$）、アクション（*Action*）、制約（C）、構成要素（normal, exist, not_exist, all）を定義するウィンドウとなっている。「構成要素」ウィンドウはさらに左から normal（$T_1(\vec{x_1}), ..., T_n(\vec{x_n})$）、 exist （$T'_1(\vec{x'_1}), ..., T'_m(\vec{x'_m})$）、 not_exist （$T''_1(\vec{x''_1}), ..., T''_l(\vec{x''_l})$）、 all （$T'''_1(\vec{x'''_1}), ..., T'''_k(\vec{x'''_k})$）を定義するウィンドウに分かれている。ユーザはここで、さらに必要な構成要素、制約、あるいは属性、アクションなどを追加したり、不必要な制約を削除したりとテキストを編集することによって生成規則を定義する。

しかし、この手法では、最初の入力で図形を利用するものの、その後の編集ではこの図が利用されず、結果としてほとんどテキストによって CMG を編集していた。 CMG は図形間の関係を記述するものであるということは既に述べたが、図形関係を扱うには１次元的なテキストの場合と異なり多次元（恵比寿では２次元）な位置関係を扱うため文法の記述が複雑である。それを１次元的なテキストだけで編集するのは次元のギャップがあり直感的にわかりにくく、ユーザに多大な負担をかけることとなる。また描画された図から簡単な制約をシステムが自動生成してはいるが、実際に利用すると不要な制約が多く生成さ

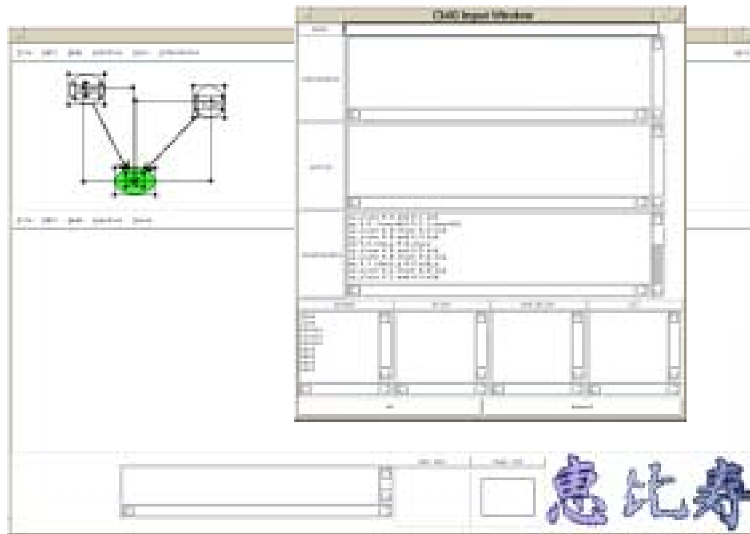

図 2.3: 恵比寿の CMG 入力法 2

れたり、意図していた制約が無い場合があった。そこで本論文ではこれまでの生成規則の定義方法を見直し、特に制約部分の入力方について考察し、最初に描いた図形を利用した視覚的でインタラクティブな編集環境と、よりインテリジェンスな制約自動生成を提供する入力インターフェイスを提案する。

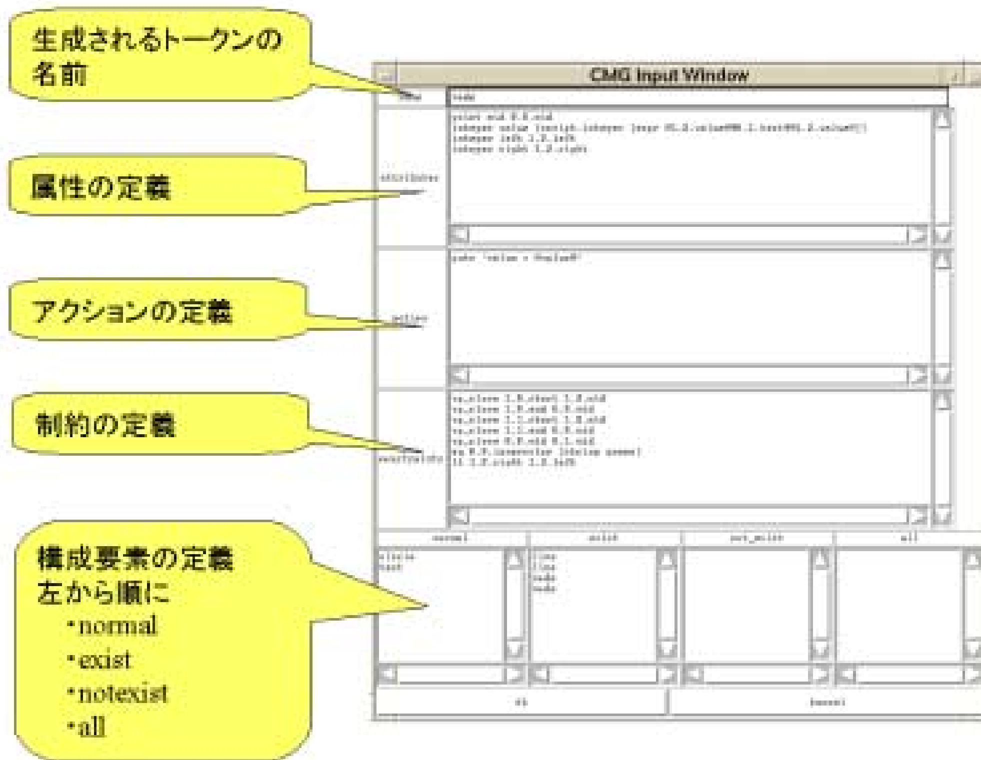

図 2.4: 恵比寿の CMG 入力法３

第 3 章

CMG 入力法の考察

恵比寿で図形言語を定義する、すなわち CMG の生成規則を定義するには図 2.4の CMG ウィンドウに、上から順に

- 生成されるトークンの名前 （name）
- 属性 （attributes）
- アクション （action）
- 制約 （constraints）
- 構成要素 （normal, exist, not_exist, all）

を定義する必要がある。

本章では従来の恵比寿の生成規則のの定義方法の問題点とその解決法について、特に CMG のうち最も基本的な構成要素と制約について考察する。

3.1 構成要素の入力

生成規則を定義する際、その構成要素となる部品についての情報がまず必要となる。どのようなタイプの構成要素が必要となるかは例示的に入力、すなわちキャンバスに実際に描画することによって指定する。これは従来の恵比寿の手法と同じである。ただしここで問題となるのが、構成要素が円や直線といった恵比寿が提供する終端記号だけでなく、他の生成規則によって定義された非終端記号である場合である。この場合、キャンバスに描かれた図形が単なる終端記号の集まりではなく、１つの非終端記号として認識されたほうが後の編集が楽である。たとえばある生成規則で「円とその中央にテキストが書かれたものがノードという非終端記号である」と定義してあるとする。例示的に構成要素を入力するということはキャンバス上に円とテキストを描くが、従来の恵比寿ではこのようにして新しく生成規則を定義しようとすると、構成要素としては終端記号の円とテキストと認識される。そのため、 CMG 入力ウィンドウ上で円とテキストを削除し改めてノードと手動で書き直さなくてはならない。このようなやり方では、構成要素中に非終端記号が占める割合が増加するにつれ、その修正の手間も比例して増えるので、とても大規模なビジュアルシステムを構築することはできない。したがって、この例の場合では、描かれた円とテキストの図形が１つのノードという非終端記号として認識されるべきである。描かれた図形を１つの意味のある非終端記号として認識するためには、他の生成規則を利用してその図形が一度解析される、すなわち Spatial Parsing が行われる必要がある。そのため、生成

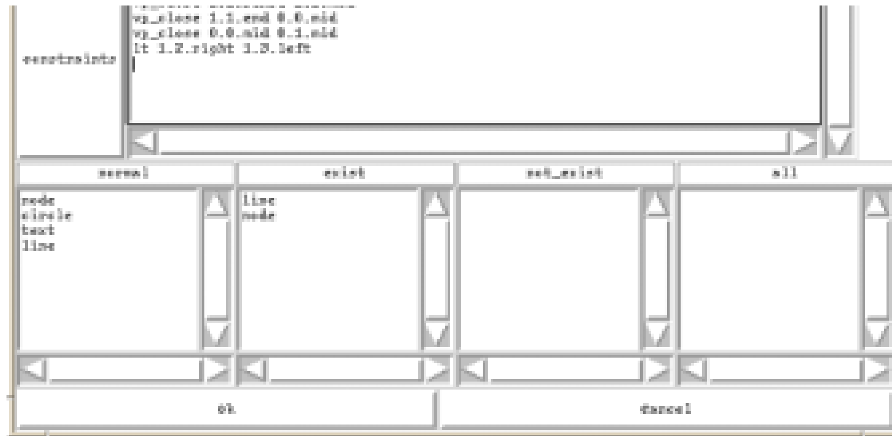

図 3.1: 構成要素表示部

規則の定義の方法としてはトップダウンではなく、ボトムアップに定義する必要がある。
またこれら構成要素について削除や、新規に追加することもやはり例示的に、すなわちキャンバス上に描いたり削除したりすることによって行うことができるべきである。

以上の手法により構成要素を入力することができるが、この時点では構成要素の種類（normal、 exist、 not_exist）が指定されていない。初期の状態ではすべて normal の構成要素として扱っているが、これは必要に応じ exist、 not_exist に変更する。これについては 3.3節で述べる。

3.2 構成要素の種類の表示

構成要素の種類（normal、 exist、 not_exist）は生成規則の定義においてもっとも基本的で、かつ重要なものである。

従来の恵比寿では構成要素の種類の識別は CMG 入力ウィンドウ下部にある構成要素表示部（図 3.1）において、 normal 欄、 exist 欄、 not_exist 欄にそれぞれ構成要素名をテキストで表示し、識別していた。（all は実装されていない）

しかし、このような表示法では構成要素間の図形的位置関係がわかりにくく、全体的な把握が困難である。たとえば図 3.1の例でいえば、 normal 欄、 exisit 欄にそれぞれ「node」という構成要素があるが、最初に構成要素を入力した図形と比較して、どちらの「node」が normal が exist かわかりにくい。

そこでこのテキストによる表示法を改め、最初に構成要素を入力する際に用いた図を利用した視覚的な表現を考える。図形を用いた表示法にも例えば構成要素を指定するとその種類を表示するといった手法（図 3.2）や、構成要素ごとに別々のキャンバスに表示するといった手法（図 3.3）があるが、これらはいずれも図形的位置関係は把握できるが、一度に表示できる種類が限られているため、構成要素全体の把握が困難である。

そこですべての構成要素の種類を同時に表示するための方法として、構成要素の種類ごとに色分けをして種類を明示し、一緒に表示する方法が考えられる。（図 3.4）

この方法ならばすべての構成要素の種類を同一キャンバスに同時に表示できるため、全体的な把握が容易である。また前述の 2 つの手法と同じく、図形間の位置関係を保存したまま表示するため直感的にわかりやすい。

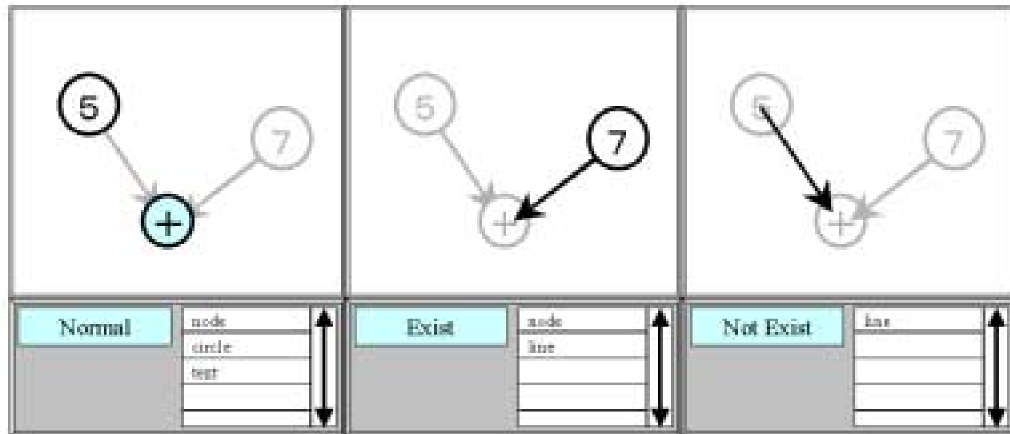

図 3.2: 構成要素の表示法１

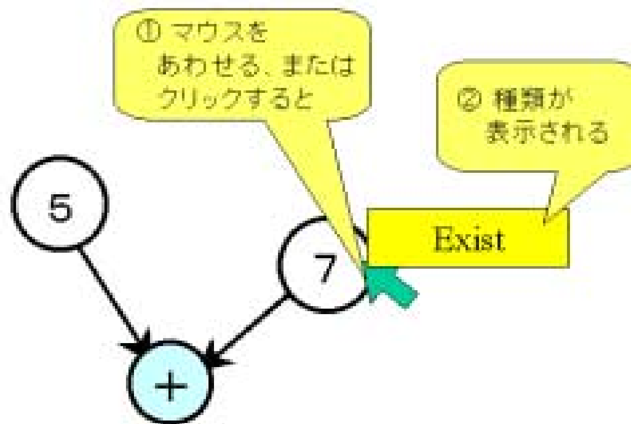

図 3.3: 構成要素の表示法２

ただし、この表示法の場合、構成要素のもともとの色情報が失われてしまうという欠点がある。そこで「種類によって色分けで表示」「構成要素の本来の色で表示」の２種類のモードを瞬時に切り替えれるようにすべきである。

また、構成要素のタイプ（円、直線、 node、 etc...）については描かれた形状から判別できるが、さらに構成要素にポインタを当てることにより、ティップスウィンドウなり、固定のメッセージウィンドウなりにそのタイプなどが表示されればより確実に識別ができると思われる。（図 3.5） なお、今回はメッセージウィンドウに表示する手法を採用した。

ティップスウィンドウは視点移動が少ないというヒューマンインターフェイス上の利点はあるが、構成要素上をポインタが動くたびにティップスウィンドウが次々と飛び出すのは煩わしい上、キャンバス上の情報を隠してしまうという欠点もある。ここで表示される情報はあくまで補助的なものなので固定メッセージウィンドウ表示方式を採用した。

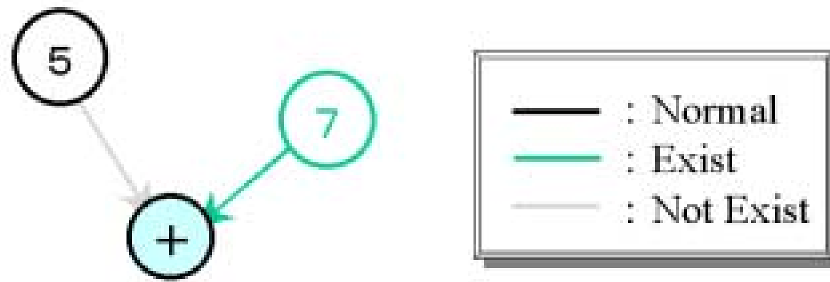

図 3.4: 構成要素の表示法３

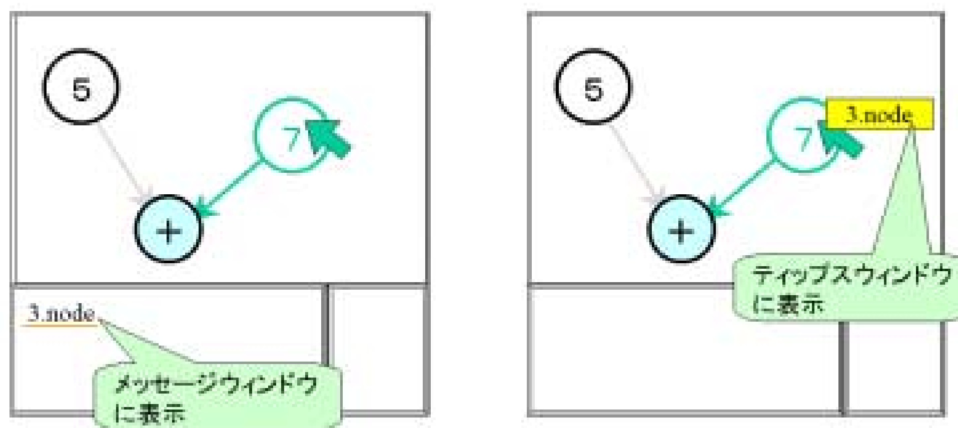

図 3.5: メッセージウィンドウ方式とティップスウィンドウ方式

3.3　構成要素の種類の変更

3.1節で述べたように、入力された時点で構成要素の種類はすべて normal となっている。そこでこれを必要に応じ exist、 not_exist に変更するわけだが、従来の恵比寿では CMG 入力ウィンドウ下部の構成要素欄に書き出された構成要素の名前を編集することで変更していた。たとえば図 3.1の normal の構成要素の「node」を not_exist に変更したい場合、まず normal 欄の 「node」 の名前を削除する。そのあと not_exist 欄に 「node」 と書きこむ。しかしこの方法では変更に手間をがかかるうえ、構成要素の名前が長い場合、名前を間違える恐れもある。またテキストで種類変更をする場合、 CMG の整合性の問題も無視できない。恵比寿では構成要素は構成要素表示部における位置によってナンバリングがされている。構成要素の識別 ID は kind.number という形式を取る。 kind は構成要素の種類で normal なら 0、 exist なら 1、 not_exist なら 3 となる。 number は 各構成要素欄における順序で 0 からはじまる整数である。図 3.1では, たとえば normal 欄の node は 1.0、 exist 欄の node は 1.1 となる。ここで問題となるのは number の部分が順序でつけられているため、ある欄からひとつ構成要素を削除してしまうと、その構成要素以降の構成要素すべての number がずれてしまうということである。図 3.1でいえば exist 欄の line

（1.0）を削除するとそれ以降の構成要素である node は 1.1 から 1.0 となってしまう。種類を変更した構成要素だけでなく、それ以外の識別 ID もずれてしまうと、恵比寿ではこの識別 ID で構成要素を特定して CMG を記述しているため、 CMG 全体を書きなおす必要がでてくる。従来の恵比寿ではこの整合性を保つのもすべてユーザに任せられていた。大規模な生成規則にになるほどその整合性を保つのは人手ではむずかしくなる。したがってこのような識別 ID 管理という単調な作業ははシステム側に一括して任せるべきである。

さて、 3.2節で構成要素の種類の表示法について、入力する際に描いた図を利用する方法を考えたが、ここでもその図を利用する方法が良いと思われる。その理由はやはり図を用いたほうが、その構成要素の図形間の位置関係を確認したまま変更作業が行えるからである。また色を用いて構成要素の種類を表示してあるということは、その種類を変更するとその結果として描かれた構成要素の色が変わるというフィードバックが得られるので、よりインタラクティブな作業ができると考えられるからである。以上のことを考慮すると 図 3.6 のような方法が良いと考えられる。まず変更したい構成要素にポインタをあてる。このときメッセージウィンドウにはこの構成要素のタイプが表示される。構成要素をクリックするとポップアップメニューが表示され、そこで構成要素の種類（normal、 exist、 not_exist）を選択する。選択すると構成要素の色が選択された種類の色に変わる。といった具合である。なお、この際構成要素の識別 ID について意識する必要はない。先ほど述べたように識別 ID の管理はシステム側がすべて行ってくれるからである。識別 ID の管理については 3.7節にて詳しく述べる。

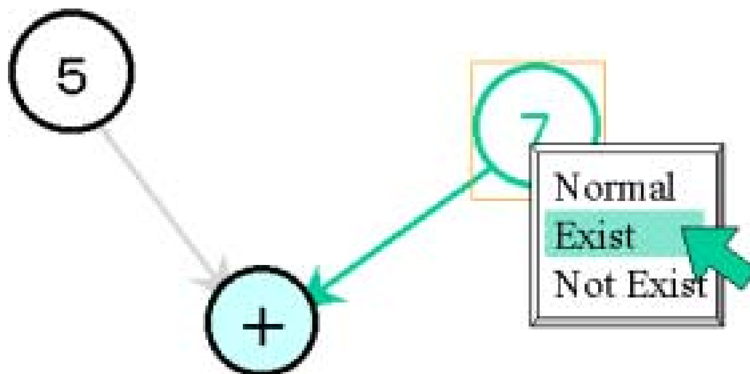

図 3.6: 構成要素の種類の変更

3.4 制約の自動生成

生成規則の適用条件は、制約を満たす構成要素が存在することであり、この制約が生成規則の重要な役割を占めている。制約は構成要素の属性間の関係を示すのであるが、構成要素入力の際、例示的に描いた図から、制約となる構成要素間の関係を合う程度推測することが可能である。システム側がある程度推測して制約を自動生成することによって、ユーザの入力を簡単化することができる。ただ制約として考えられる関係は非常に沢山あるため、必要となる制約から、あまり重要でない制約や、不必要な制約となるものも存在する。そのため従来の恵比寿にもこの制約自動生成機能は実装されていたが、このため実際に利

用すると不要な制約が多く生成されたり、意図していた制約がなかったりしていた。例えば、構成要素の属性として「線の太さ（linewidth）」という情報を持つ図形があるが、例示入力の際キャンバス上に２つの構成要素が同じ太さの線描かれていたとしても、この情報は制約として必要でないといった場合が考えられる。以上のことを考慮して、自動生成を行う前にユーザが必要となる属性を選んで指定することによって不必要な制約を振るいにかけ、ユーザが望む制約を自動生成させるようにするのが良いと思われる。

3.5　制約の編集

自動生成機能によってある程度は制約が生成されるが、一般にこれだけでは生成規則を定義するのに十分とはいえない場合が多い。そこで生成された制約のうち不必要な制約を削除したり、足りない制約を追加したりして生成規則を完成させる。従来の恵比寿では制約を追加、削除するには新たにテキストで記述するしか方法がなかった。しかし単純にテキストで記述する場合、新たに制約を追加するには恵比寿における CMG の文法について正確な知識を持ち合わせていないと記述できず、また文法を知っていてもついつい構文エラーを犯す場合がある。削除する場合にも、恵比寿の CMG 入力ウィンドウには Undo 機能がついていないので、誤って削除してしまった場合、書き直す以外に復元させることはできない。また 3.3節に述べたように識別 ID の整合を保つのも大変である。

そこでテキストによる記述以外の方法で制約の編集を行う方法を考える。

3.5.1　制約の入力

制約には、例えば

- 矩形１の中心点と矢印２の始点の座標が一致している
- 円３の半径が円４の半径より大きい
- イメージ５とイメージ６のイメージ画像が異なる
- 円３を描いている線の太さが２ pt である
- 矩形１の内部の色は白である

とさまざまなものがある。それぞれまったく異なる関係を示すものであるが、すべて

$$op\ Arg1\ Arg2$$

の形式で示すことができる。ここで op は制約の種類、 Arg1、 Arg2 は「構成要素（の識別 ID）. 属性名」、もしくは「{ 型　定数 }」である。例に述べた制約をこの形式にしたがった恵比寿式に表現すると

- eq 1.mid 2.start
- gt 3.radius 4.radius
- neq 1.image 2.image
- eq 3.linewidth {integer 2}
- eq 1.innercolor {string white}

といった具合である。つまり制約を入力するということは制約の種類と属性名もしくは定数を一組にして指定するということである。そこで、図 3.7のような制約生成ウィンドウを設け、その各フィールドに制約の種類、属性名、定数を何らかの方法で入力することによって制約を生成する

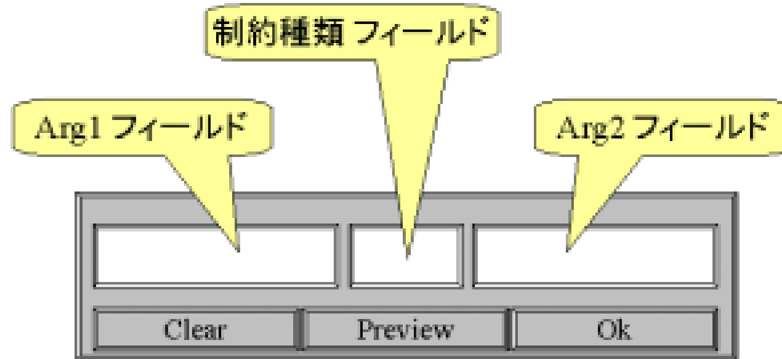

図 3.7: 制約生成ウィンドウ（プロトタイプ）

なお、恵比寿の CMG ウィンドウでは制約は前記法で表示されているが、この制約生成ウィンドウでは中記法で表示するレイアウトとした。恵比寿の CMG ウィンドウにおける表記法とのギャップが気になるが、本論文では最終的にユーザが CMG ウィンドウに記述することなしに制約を定義できるようにすることを目的としているため、恵比寿の表現との整合性を図るより、ユーザの使い勝手のほうを優先させるため中記法を採用する。

まず制約の種類の入力方法について考える。現在恵比寿では制約の種類には

- eq (equal)
- neq (not equal)
- gt (greater than)
- ge (greater or equal)
- lt (less than)
- le (less or equal)
- vp_close
- on_circle

の８つがある。このうち vp_close 制約は eq 制約の亜種、 on_circle 制約は十分に実装されていないので実質６つだけである。したがって選択肢の少ない制約の種類は制約種類入力フィールドにメニューを設け、そこから選ぶようにすれば良いと考えられる。（図 3.7）

また制約の種類は「eq」「gt」といった文字情報で表現するのではなく、一目見て理解できる「＝＝」「＞＝」といった記号で表現した。これも恵比寿の CMG ウィンドウにおける表記法とのギャップが気になるが、同様の理由より、ユーザの使い勝手を優先させる

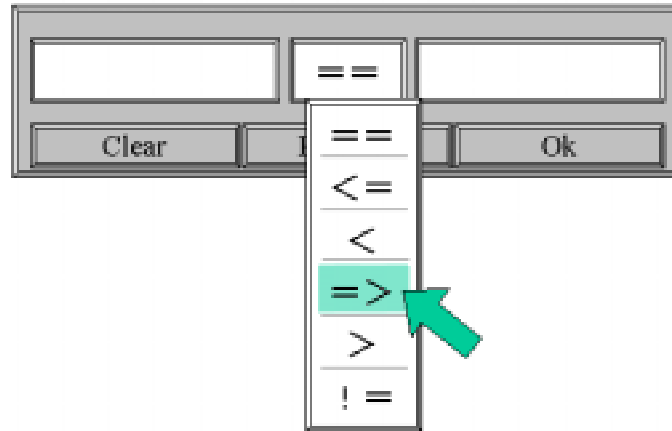

図 3.8: 制約生成ウィンドウ（制約の種類入力）

ため記号による表現法を用いる。

次に「構成要素. 属性名」の入力方法である。順番としては、まず構成要素を選んでから、その構成要素中にある属性名を選択するというのが自然であり、直感的に理解しやすい。そこでまず構成要素の指定法から考えるが、これは構成要素の種類を変更する際に構成要素を選択する手法と同じ状況である。したがってヒューマンインターフェイスの観点から見て、統一的な手法を用いたほうが良いので、同様に例示入力に用いた図をポインタで指し示すことにより指定する方法が良いと思われる。そして属性名の選択であるが、一つの構成要素が持つ属性の個数は一般にさほど多くないので、メニューから選ぶ方式が良いと思われる。すなわち構成要素を指定して何らかの操作 — 一般にはマウスクリックなど — をすると、属性名のリストがメニューとして現れ、その中から選んで指定するという方法である。ところでその属性名のリストメニューだが、これは制約の種類とは異なり、制約生成ウィンドウの Arg フィールドに表示させ選択するのではなく、選んだ構成要素の近くにポップアップメニューとして表示させるのが良いと思われる。なぜなら、構成要素の属性は普段は表示されないので、場合によっては複数の構成要素の属性リストを同時に表示して見比べたりすることもあるからである。ただしリストが複数出現する場合、どのリストがどの構成要素と対応しているのかが明らかでなくてはならない。そこであるリストの上にポインタが乗ると、そのリストに対応した構成要素がセレクト状態 — 例えばバウンディングボックスで囲まれる — になって対応していることを明示すべきである。そしてポップアップメニューに表示された属性名を Cut&Paste の要領で制約生成ウィンドウの Arg フィールド入力する。（図 3.9）

最後に定数の入力についてだが、これは基本的にテキストで入力する以上に格段に優れた入力インターフェイスが思いつかないので、基本的に Arg フィールドにテキストで入力する手法を採用する。ただし定数の型はユーザが意識しなくてもシステム側で自動的に判別してくれるのが望ましい。

ところで、例えば「円１の内部の色が白である」という制約が欲しい場合、例示入力の際にはやはり円の内部の色を白く描くのが普通である。したがって（一般にはテキスト入力となるが）制約に用いる定数が既に例示入力の際使用されていれば、その定数、すなわ

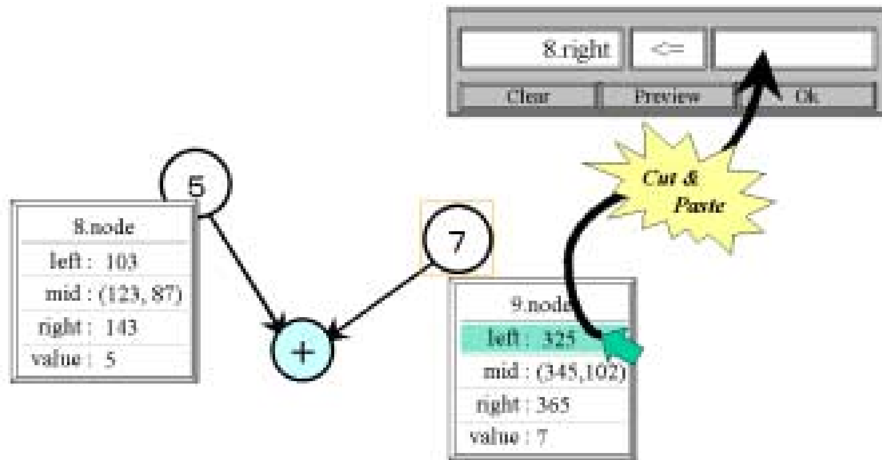

図 3.9: 制約生成ウィンドウ（属性名の入力）

ち属性値を利用して入力することが可能である。先ほどの例の「円 1 の内部の色が白である」というのを恵比寿風に表現すると「eq 1.innercolor {string white}」となる。例示入力の際、円 1 の内部を白く描いてあるということは「1.innercolor」という属性の属性値が「 {string white}」ということである。したがって「eq 1.innercolor {string white}」は「eq 1.innercolor 1.innercolor （の属性値）」と表現できる。よって「構成要素. 属性名」を指定する際に $Arg1 = Arg2$ とした場合、 $Arg2$ は属性名ではなく、その属性値と解釈するようにすればよい。なお、 $Arg1 = Arg2$ としたとき、 $Arg1$、 $Arg2$ ともに属性名と解釈して欲しい場合があるのではという疑問があるかもしれないが、、 $Arg1 = Arg2$ で、 $Arg1$、 $Arg2$ ともに属性名と解釈した場合、これは制約としては意味を成さないのでこの点は問題ない。

3.5.2 制約の削除

さてこれまで制約の新規入力について考察してきたが、制約の削除についても考えてみる。

制約は多すぎると矛盾をきたし解くことができないし、少なすぎると解を一意に定義できない。したがって生成規則を定義する際には割と試行錯誤を繰り返すことになる。よって不必要と思って削除した制約がやはり必要となることもある。従来の恵比寿ではテキストで記述することによって編集していたので、不要とおもわれる制約は削除してしまう以外に方法がなかった。したがって削除においてなんらかの Undo 機能があることが望ましいが、完全な Undo 機能を実装するのは困難である。ところで、なぜ Undo 機能が必要なのかというと、不必要と思われる制約を削除してしまうからである。ということは削除を行わなければ Undo 機能を実装する必要はないということになる。そこで各制約に有効 / 無効を定義し、不必要な制約は無効とすることにより生成規則に反映されないようにすれば良い。必要となればその制約を再び有効とすれば良い。このように削除ではなく有効 / 無効を定義することにより結果的に Undo 機能を実装したのと同様の効果が得られる。

3.6 制約の視覚化

制約の有効 / 無効を選択する際には当然その対象となる制約を選ばなくてはならない。制約を選ぶ方法としては単純にリストに表示してそこから選択するという手法を用いる。制約を完全に視覚化して図から選ぶという手法も考えられるが、定数値や「ある図形単語がある図形単語より右にある」などといった視覚化するのが困難、もしくはかえって分かりにくくなる恐れの制約も存在するので、基本的には前述のとおりリストから選択というほうが良いと思われる。ただ、リストに書かれた制約を見ただけでは、これがどの制約か直観的に理解する事は難しい。そこで選んだ制約がどの制約かを理解するための補助情報として、ある程度の制約の視覚化を考えてみる。

１つの制約は２つの属性間、もしくは１つの属性と定数の関係を記述したものである。したがって制約を理解するのには、制約の対象となっている属性がなんであるかを視覚的に明示するのが有効であると思われる。しかし属性はそれが「円の中の色」とか「線の太さ」などであれば、それぞれ「円の中」「線」などと視覚的に明示することは容易であるが、「計算結果」など内部値はそれを明示することは難しい。そこで属性ではなくその属性を持っている構成要素を明示し、その上で視覚化して分かりやすい属性のみをさらに視覚化して示すという手法が現実的で良いと思われる。視覚化するためには例示入力の際に描いた図を利用する。構成要素は例示入力図に描かれているため、どのような制約でも同じ手法で明示できる。全ての制約をそれぞれ別の方法で完全に視覚化するよりも、このように大まかであるが統一的であるほうがヒューマンインターフェイス的に優れていると思われる。

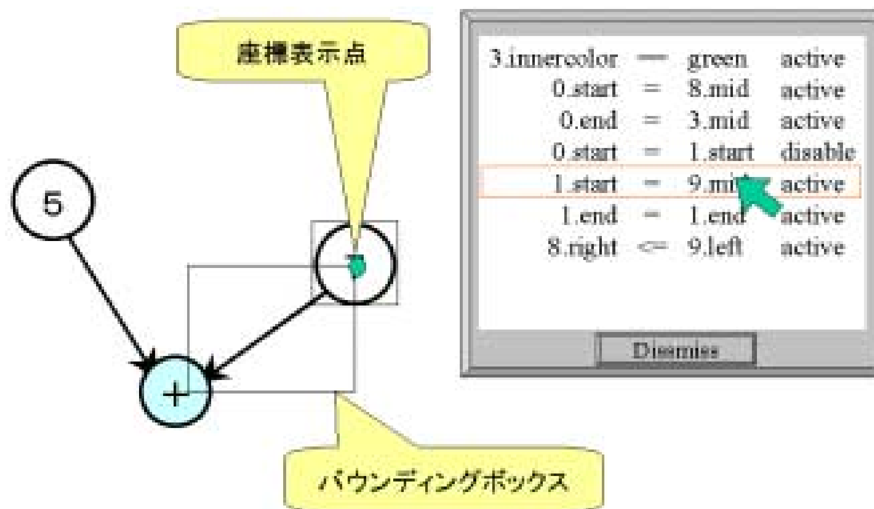

図 3.10: 制約の視覚化

具体的には図 3.10のようになる。

制約が表示されたリストの上にポインタを持ってくると、そのポインタの指す制約の対象となっている属性を持っている構成要素がバウンディングボックスで囲まれる。またその属性が視覚化するのに向いていればさらにそれを視覚的に明示する。図 3.10なら座標情報をその位置に点を表示することにより示している。視覚化に向いている属性としては、座標（点）、図形の色、線の太さなどが考えられる。

3.7 識別 ID の整合性

3.3節でも述べた通り、恵比寿では CMG を記述する際、属性を指定する識別 ID として kind.number という形式を取っている。 kind は構成要素の種類で normal なら 0、 exiast なら 1、 not_exist なら 3 となる。 number は 各構成要素欄における順序で 0 からはじまる整数である。よって構成要素の種類を変更すると、変更した構成要素の kind はもちろん、 number が順序でつけられているため、その構成要素以降の構成要素すべての number がずれてしまう。したがって一つの構成要素の種類を変えただけで多くの構成要素の識別 ID が変わってしまうため、その整合性を保つために CMG 全体を書きなおす必要が生じる場合がある。そもそもこのような問題が生じるのは識別 ID が構成要素の種類、順序という動的なパラメータによって定義されているからである。したがって識別 ID を不変のもの、例えば構成要素に振った通し番号などで定義すればこの問題は解決できる。もともと従来の恵比寿では CMG ウィンドウにおいてテキストで CMG を記述するため、ユーザーが構成要素の識別を容易にできるようにするため kind.number という識別 ID を用いたのだが、本論文では最終的にユーザが CMG ウィンドウに記述することなしに制約を定義できるようにすることを目的としているため、整合性を保つため通し番号で識別 ID を定義する。

第 4 章

システム「エビ chu」の実装

本章では「考察」の章で述べた手法に基づいた CMG 入力インターフェイスを恵比寿上に実装したシステム「エビ chu」について述べる。

4.1 実装方法

4.1.1 使用言語

インターフェイスの実装には恵比寿本体と同じく Tcl/Tk[12] を用いた。

4.1.2 システム構成

なおインターフェイス部は従来の恵比寿のシステムとは基本的に独立している。実際に CMG を受け取り、解釈する部分は恵比寿のシステムを利用するため、入力インターフェイスを用いて入力した CMG を直接処理するのではなく、恵比寿の書式に変換して CMG ウィンドウに書き出す手法を用いた。（図 4.1）

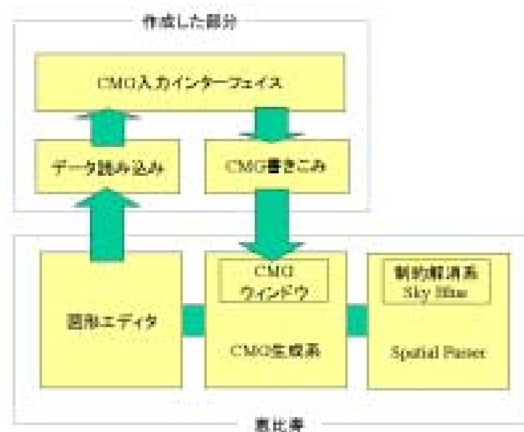

図 4.1: システム構成

この手法の利点は、既存の恵比寿のプログラムを再利用するためコーディングが容易な点と、恵比寿本体と独立しているため恵比寿のシステムを改良する際、インターフェイスの部分を意識無くても良いという点である。また CMG ウィンドウを介しているため、ま

だ実装されていない部分や、属性、アクションについては恵比寿本来のテキストによる記述で補うことができる。

4.1.3 画面構成

エビ chu の実行画面は図 4.2のようになる。

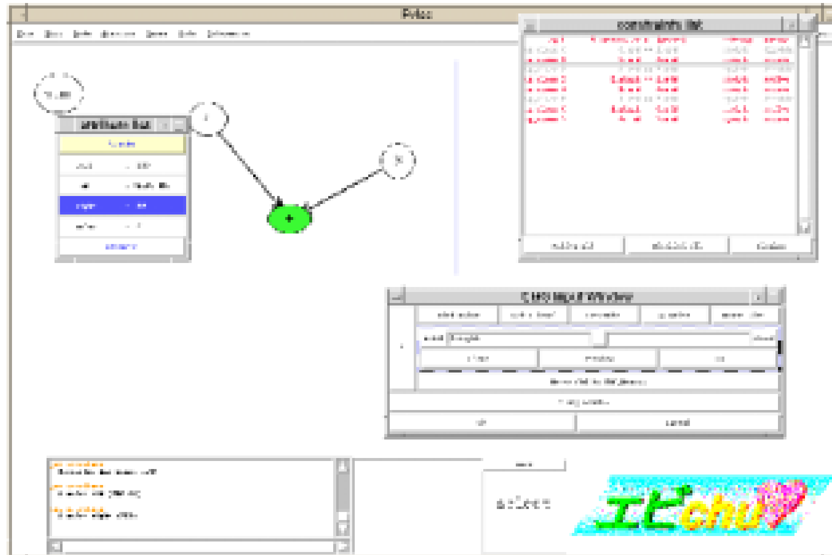

図 4.2: 実行画面

従来の恵比寿は上部に定義ウィンドウ、下部に実行ウィンドウと 2 画面に分割され、実行ウィンドウにのみ Spatial Parsing 機能が付いていた。しかしエビ chu では定義時にも Spatial Parsing が必要なので、定義ウィンドウと実行ウィンドウに差異が無くなったため、ワンウィンドウとした。ワンウィンドウとすることにより、定義と実行の境界がなくなり、よりスムーズな操作を可能としている。

生成規則の定義を行うには、ウィンドウに例示入力図を描き、指定する。するとその図の周辺がボックスで囲まれ、その内部が擬似的に定義ウィンドウとなる。同時に入力インターフェイスとしてのメインメニュー（図 4.3）が出現する。以降、基本的にこのメニュー

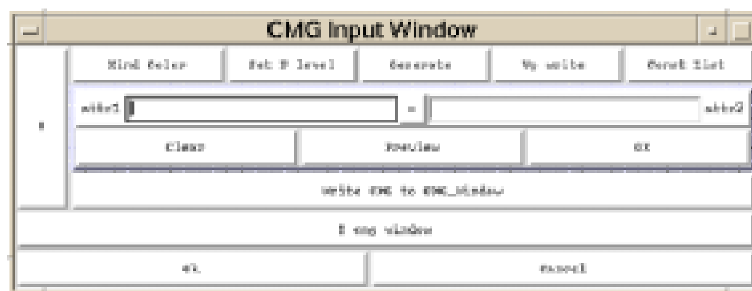

図 4.3: メインメニュー

と例示入力図に対するマウス操作のみで、制約の定義が可能となる。

4.2 データ構造

本節では変数名など実際の文字列はタイプライタ体で、値が変わるものはイタリック体で表記してある。

現在エビ chu では、直接恵比寿内部のデータをいじることはしない。例示入力された図形を Spatial Paring した結果なども全て独自のデータとして持ち、これを参照する。具体的には恵比寿内部での ParseForest のデータをもつ連想配列 `D` からデータ読み込み部を介して必要とするデータを連想配列 `I` に格納している。また入力インターフェイスを実現する上で必要な情報もこの連想配列 `I` に格納している。連想配列 `I` のそれぞれの要素は表 4.1のようになっている。

要素名	値
`I(top)`	トップレベルのトークンの id のリスト
`I(num)`	全トークンの個数
`I(`*t-id*`.type)`	トークンのタイプ
`I(`*t-id*`.attr.`*a_name*`)`	属性名： a_name の値
`I(`*t-id*`.normal)`	構成要素の id，非終端記号のみ
`I(`*t-id*`.oricol)`	トークンの本来の色情報，終端記号のみ
`I(`*t-id*`.kind)`	トークンの種類
`I(`*t-id*`.id)`	恵比寿 CMG ウィンドウにおける識別 ID

表 4.1: 連想配列 `I` の要素

ここで *t-id* はトークンにユニークにつけられた id である。

また定義された制約も CMG ウィンドウに書き出すまではエビ chu 内部の連想配列 `VC` に格納している。連想配列 `VC` のそれぞれの要素は表 4.2のようになっている。

要素名	値
`VC(`*comp*`.num)`	制約の個数
`VC(`*comp*`.`*c-id*`.id1)`	対象となるトークンの id
`VC(`*comp*`.`*c-id*`.id2)`	対象となるトークンの id
`VC(`*comp*`.`*c-id*`.a_name1)`	対象となる属性名
`VC(`*comp*`.`*c-id*`.a_name2)`	対象となる属性名
`VC(`*comp*`.`*c-id*`.type)`	属性のタイプ
`VC(`*comp*`.`*c-id*`.valid)`	この制約の有効 / 無効を示す
以下 *comp*=`vp_close` のみ	
`VC(vp_close.`*c-id*`.x)`	vp_close している x 座標
`VC(vp_close.`*c-id*`.y)`	vp_close している y 座標
`VC(vp_close.`*c-id*`.c_id)`	vp_close ポイントを示す円のキャンバス id

表 4.2: 連想配列 `I` の要素

なお、 *comp* は制約の種類（eq, vp_close, lt, le, gt, ge, neq）、 *c_id* は各制約の種類内での制約の id である。

4.3 考察の反映

第 3章で述べた手法をどのように実装したかについて述べる

4.3.1 構成要素の入力

構成要素の例示入力には従来の恵比寿が持っている図形エディタとしての機能をそのまま用いた。また入力された終端記号を一度 Spatial Parsing するのにも恵比寿の Spatial Parser としての機能を利用し、実装した。ただし、入力した構成要素の削除、および新規追加の機能は実装していない。

4.3.2 構成要素の種類の表示と変更

構成要素の種類は

- normal ··· 青
- exist ··· 赤
- not_exist ··· 灰色

で表現した。その理由は normal と exist の色は恵比寿におけるトークンリスト表示の際の表示色と整合性をとるためであり、 not_exist はネガティブなイメージを持つためである。all は恵比寿自体に実装されていないため、扱っていない

変更する際のポップアップメニューには本来の色で表示する「Original Color」というメニューアイテムを追加した。「種類で色分けして表示」と「構成要素本来の色で表示」を切り替える機能も実装している。メインメニュー上の「Original/Kind Color」ボタンを押すと交互に切り替えられる。また、このポップアップメニューでは現在の構成要素の種類のメニューアイテムのみ色付きで表示される。（図 4.4）

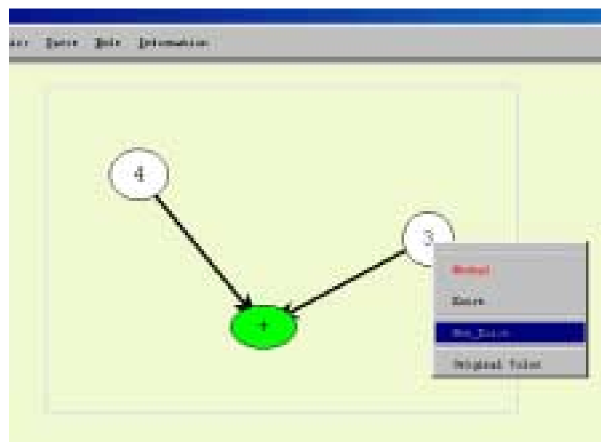

図 4.4: 構成要素の種類の変更

たとえば、ある構成要素の種類が exist なら、ポップアップメニューの exist メニューアイテムの文字の色のみが exist を示す赤で表示される。これによって「構成要素本来の色で表示する」モードでも構成要素の種類を変更する際には現在の構成要素の種類が確認できる。

さらに構成要素のタイプの表示であるが、基本的には図形から視覚的に判断するので、補助情報として本来のシステムメッセージウィンドウの横に専用のインフォメーションウィンドウを設け、そこに表示するようにした。（図 4.5）この時その構成要素が非終端記号の

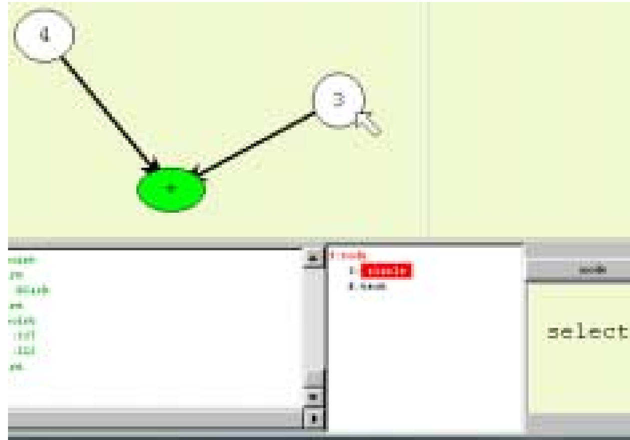

図 4.5: 構成要素のタイプの表示

場合、さらにその構成要素が終端記号まで同時に表示される。そして現在ポインタが指し示している終端記号の名前が反転表示される。なお表示される文字も構成要素の種類の色で表示され、ここでも構成要素の種類を確認することができる。

4.3.3 制約の自動生成

自動生成を行う際、ユーザは必要となる属性を選択できるが、その選択できる属性は現在表 4.3の通りである。

なお、従来の恵比寿と異なり、自動生成は必要とする属性を変えて何度でもやり直すことができる。

4.3.4 制約の編集

制約の入力、削除方法については考察で述べた通りである。

ただ座標の vp_close 制約については、例示入力図に視覚的に表現し、かつそこで制約の有効 / 無効を選択できるようにする手法を試験的に実装してみた。具体的にはメインメニュー上の「制約表示ボタン」を押すと、 vp_close 制約がかかっている点に円が表示される。その円にポインタを合わせクリックするとメニューが表示され（図 4.6）そこでその vp_close 制約を無効にするかを選ぶ。また vp_close 制約が一つの座標で複数かかっている場合、制約を示す点が重なって表示され、下の制約が確認できない場合があるので、制約を無効にしないで、その表示円だけを消すこともできる。このときインフォメーションウィンドウには vp_close 制約がかかっている 2 つのトークンが表示される。

Point	座標
Size(x)	横方向の大きさ
Size(y)	縦方向の大きさ
InnerColor	図形の内部の色
LineColor	線の色
Color	テキストなどの色
Text	文字列
FontType	フォントのタイプ
FontSize	フォントのサイズ
Image	イメージ画像
LineWidth	線の太さ
Arrow	線の矢印の型

表 4.3: 自動生成で選択される属性

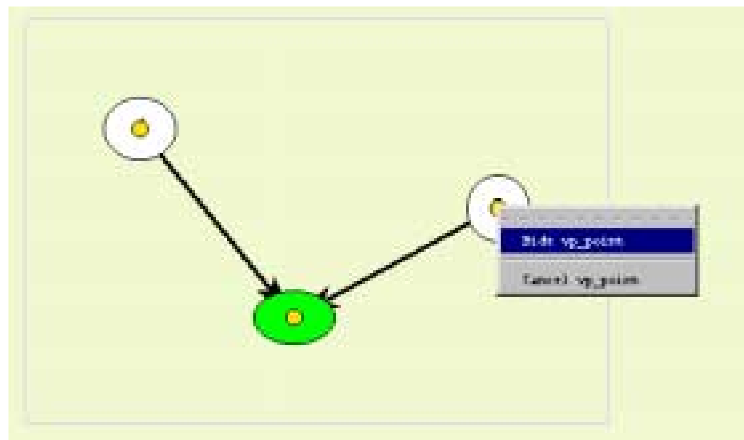

図 4.6: vp_close 制約の視覚的選択

4.3.5 識別 ID の整合性

考察で述べた通り、入力インターフェイス上では全てのトークンにユニークな通し番号をつけてそれを識別 ID としている。しかし最終的には制約を恵比寿の CMG ウィンドウに書き出すので、恵比寿式の識別 ID に変換しなくてはならない。その方法は以下の通りである。

1. 構成要素を CMG ウィンドウの構成要素表示部に書き出す際、その構成要素の種類の順序を連想配列 `I(`$t\text{-}id$`.id)` に格納する。

2. 制約を CMG ウィンドウに書き出す際、識別 ID を $kind$.I(t-id.id) とする。ここで $kind$ は

$$kind = \begin{cases} normal & \cdots \mathtt{I}(t - id.\mathtt{kind}) = blue & : 0 \\ exist & \cdots \mathtt{I}(t - id.\mathtt{kind}) = red & : 1 \\ not_exist & \cdots \mathtt{I}(t - id.\mathtt{kind}) = gray & : 2 \end{cases}$$

である。

第 5 章

システム「エビ chu」の実行例と評価

本章ではエビ chu を用いて実際にビジュアルシステム「計算の木」を作成する例をあげる。またその作成過程を通してエビ chu の評価を行う。

5.1　計算の木

計算の木とは、たとえば

$$(3+4)\times 5$$

という式を図 5.1 のように表現し、その結果として式の答えである 35 という結果を得られるビジュアルシステムである。

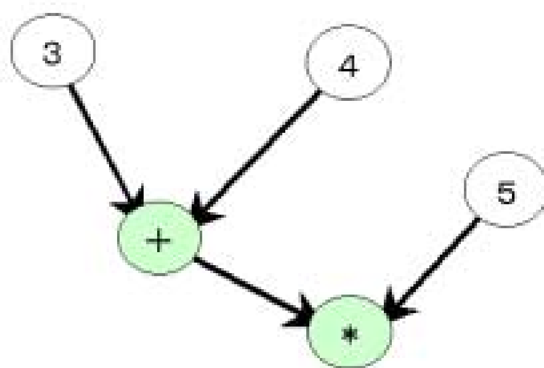

図 5.1: 計算の木

計算の木は以下の 2 つの生成規則を定義することによって再帰的に定義できる。

1. ノードは円の中に数字が描いてあるものである。
2. ノードは円の中に演算子が描かれ、それに 2 つのノードが矢印によってつながれている。

これをより厳密に拡張 CMG の生成規則に則って記述すると以下のようになる

```
1:    node(point mid, integer value, integer left, integer right) ::=
2:       C:circle, T:text
3:       where (
4:          C.mid == T.mid
5:          C.innercolor = white
6:       )
7:       and {
8:          mid = C.mid;
9:          value = {script.integer{set dummy @T.text@};
10:         left = C.lu_x;
11:         right = C.rl_x;
12:      } and {
13:         display(value = @value@);
14:      }
15:
16:   node(point mid, integer value, integer left, integer right) ::=
17:      C:circle, T:text
18:      exists N1:node, N2:node, L1:line, L2:line
19:      where (
20:         N1.mid == L1.start &&
21:         N2.mid == L2.start &&
22:         C.mid == L1.end &&
23:         C.mid == L2.end &&
24:         C.mid == T.mid &&
25:         C.innercolor = green &&
26:         N1.right < N2.left
27:      )
28:      and {
29:         mid = C.mid;
30:         value = {script.integer {@N1.value@@t.text@@N2.value@}}
31:         left = C.lu_x;
32:         right = C.rl_x;
33:      } and {
34:         display(value = @value@)
35:      }
```

1 行目から 13 行目までが生成規則 1 の定義である。 1 行目は図形単語の名前と属性を定義している。名前が node であり、 mid, value, left, right という属性値を持ち、それぞれの型は mid が point 型、それ以外が integer 型である。 2 行目では図形単語がどのような構成要素からできているかを定義している。ここでは全て normal 要素で、一つの円（circle） C と文字列（text） T から成るということを示している。 4 行目 5 行目は制約「Constraits」を定義している。 4 行目では円 C の中心座標： mid という属性と文字列 T の中心座標： mid という属性がほぼ一致しているという制約を定義し、 5 行目では円 C の内部の色： innercolor という属性が白であるという制約を定義している。 7 行目以下は生成規則が適用されたとき実行される部分である。 8 行目から 11 行目までは定義中の図形単

語にどのような属性値を与えるかという「Attributes」を定義している部分である。8 行目は mid という属性と円 C の中心座標：mid という属性が常に等しいことを、10,11 行目は left, right という属性がそれぞれ円 C の左上の x 座標：lu_x、右下の x 座標：rl_x と一致させるということを示している。9 行目は value という属性の値を定義しているが、ここで script 指令を用いている。script 指令について詳しくは生成規則 2 で説明する。ここではテキスト T の文字列：text を整数として扱うことを示している。13 行目は「Action」を定義している。ここで display が表示させるコマンドだとすると、この生成規則が適用されたとき

```
value = 3
```

などと表示される。なお @ マークで囲まれた属性は、その属性値に変換される。

16 行からが生成規則 2 の定義である。生成規則 1 とほとんど同じであるので異なる点のみを説明する。まず 18 行目であるが、これは構成要素として exist の構成要素を必要とすることを示している。25 行目では円 C の内部の色：innercolor が緑色であるという制約を示している。この制約は演算子の描かれた円と数字の描かれた円を区別するためにあるまた 26 行目の制約はノード N1 の右の x 座標：right がノード N2 の左の x 座標：left より小さい、すなわちノード N1 がノード N2 より左にあるということを示す制約であるこの制約は左右のノードを区別するためにある。この制約が無ければ、例えば「5 − 2」なのか「2 − 5」なのか区別することができないといった状況が起きる恐れがある。30 行目ではまた script 指令を用いている。script 指令とは書かれた文字列をスクリプト処理系に渡し実行させ、図形単語の属性値を生成するのに用いられる。script 指令は次のような形式である。

$$\{\texttt{script}.type\ \ \{script\}\}$$

ただし *type* は得られた結果をどのような型として扱うかを示し、*script* はスクリプト処理系に解釈させる文字列である。なお *script* 内では @ マークに囲まれた属性は、その属性値に変換されて処理系に渡される。script 指令が無いとすると CMG の解析系があらかじめ持っている関数を用いて値を生成するしかないが、script 指令を用いることによって、スクリプト言語によってより高級な演算が可能となる。

5.2 ビジュアルシステム：計算の木の作成

5.2.1 生成規則１の定義

生成規則 1 を定義する過程を述べる。

1. 構成要素の例示入力を行う。

1.1 キャンバス上に円を描きその中心付近に適当なテキスト文字列（図 5.2）を描く。

図 5.2: 例示入力図

1.2 それらの図形をまとめて選択し、 Rule メニューから VCW を選択する。

2. 上記の操作を行うと例示入力図周辺がボックスで囲まれ、メインメニューが現れる。（図 5.3）

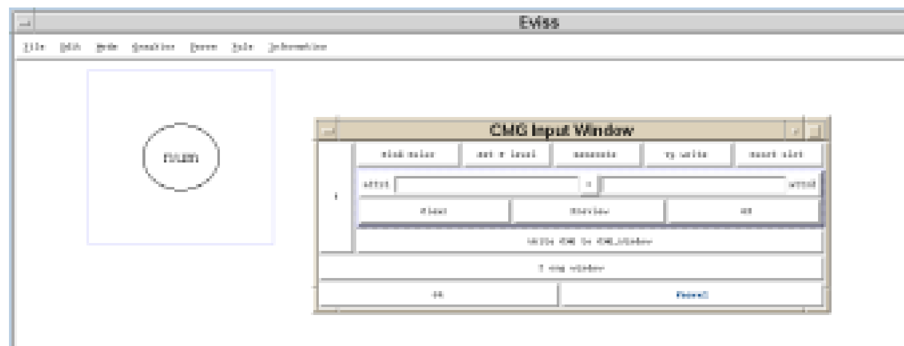

図 5.3: 作業画面

3. 例示入力図から制約を自動生成してみる。

3.1 メインメニューの「Set P level」ボタンを押し、自動生成したい制約を選ぶ。ここでは座標制約だけを生成して欲しいので、 Point の項をチェックする（図 5.4）

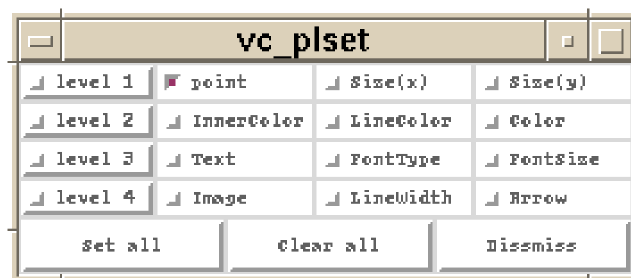

図 5.4: 自動生成設定パネル

3.2 そしてメインメニューの「Generate」ボタンを押すと制約が自動生成される。

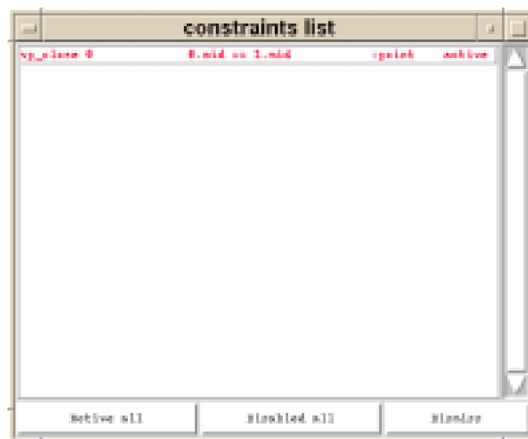

図 5.5: 制約リスト

3.3 どんな制約が生成されたかは、メインメニューの「Const List」ボタンを押すと制約リストが表示され、確認できる。（図 5.5 ）

ここではこの制約が必要なのでそのままにする。

4. 次に足りない制約「円の内部の色が白」を手動で入力する。

4.1 円を中ダブルクリックすると属性リストが現れる。（図 5.6）

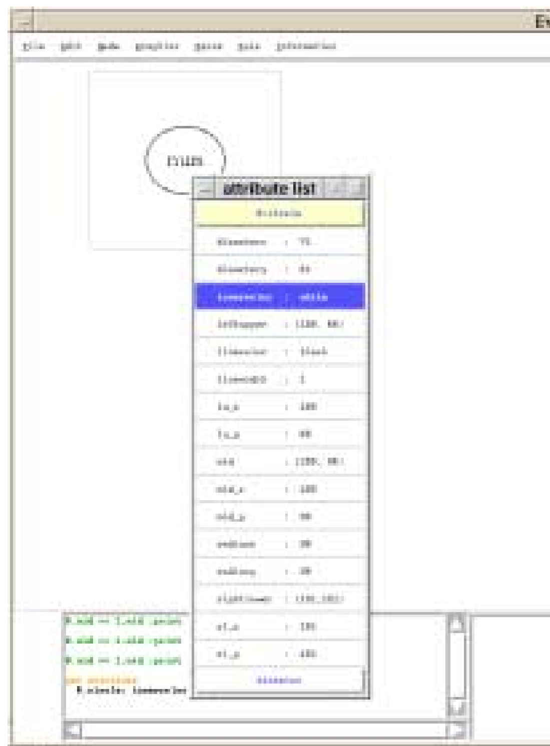

図 5.6: 属性リスト

4.2 リスト中から「円の内部の色」を示す属性： innercolor を選んでクリック。円はあらかじめ白で書かれているので、 innercolor の属性値は「white」である。

4.3 ここでは二属性関係制約ではなく、１属性と定数の制約を入力するので制約生成ウィンドウの両 Arg フィールドに innercolor をダブルクリックで入力。（図 5.7）

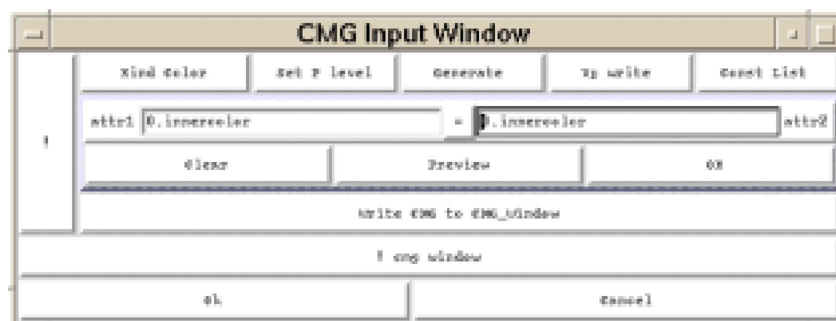

図 5.7: 制約の入力

4.4 制約の種類は「=」として、最後に「OK」ボタンを押すと制約が入力される。図 5.8 は「Cont List」で確認したところ。

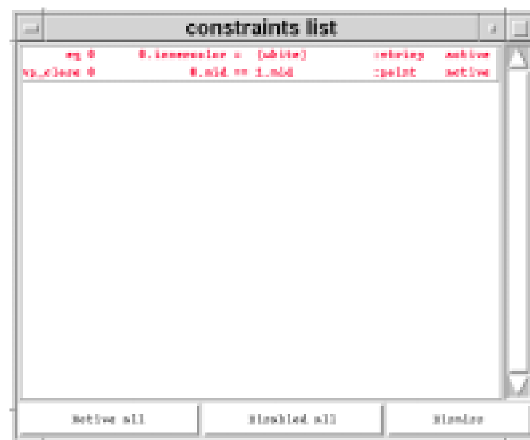

図 5.8: 制約リスト

5. すべて終わったら最後に「Write CMG to CMG_Window」ボタンを押して CMG ウィンドウに書き出す。図 5.9 はメインメニューより CMG ウィンドウを開いて書き出した所である。

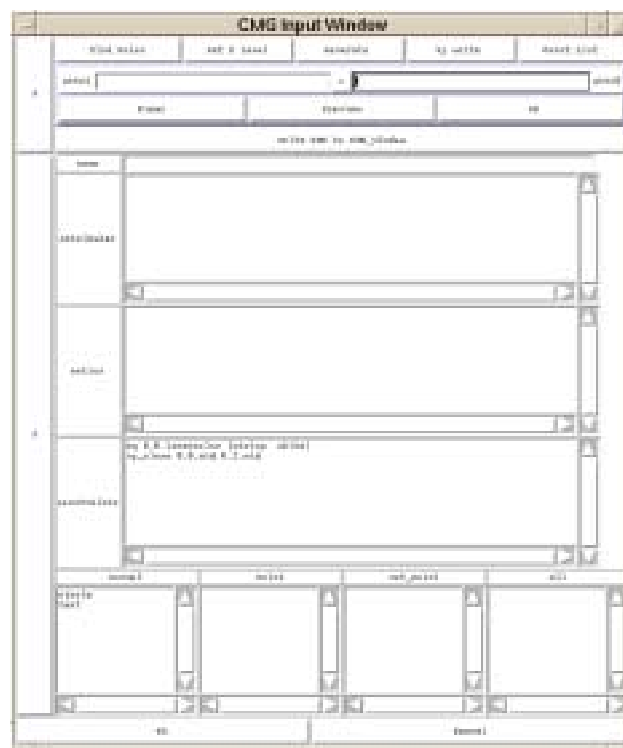

図 5.9: CMG ウィンドウ

あとは属性、アクションを定義して完成。

以上で生成規則 1 の定義は終了である。なお、生成規則 1 では全ての構成要素が normal のため、構成要素の種類の変更は行なわなかった。

5.2.2　生成規則２の定義

生成規則 2 を定義する過程を述べる。

1. キャンバス上に図 5.10 のような図形を描き、その図形をまとめて選択し、 Rule メニューから VCW を選択する。

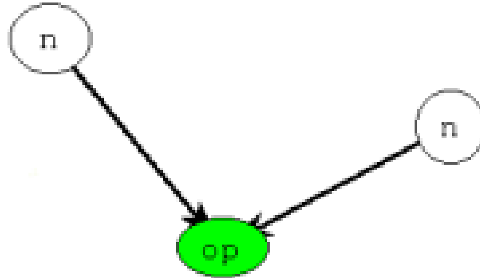

図 5.10:　例示入力図

2. まず構成要素の種類を変更する。

2.1 とりあえずメインメニューの「Kind Color」ボタンを押してみる。すると構成要素が種類別に色分けされて表示される。この段階ではすべて normal なので青色で表示される。また元の色に戻したいときは「Kind Color」ボタン（このときラベルは「Original Color」に変わっている）を再び押す。

2.2 最初に右ノードの種類を normal から exist に変える。ポインタを右の円の上に持ってくるとインフォメーションウィンドウに、これが単なる円ではなくノードと認識されていることが表示される。（図 5.11）

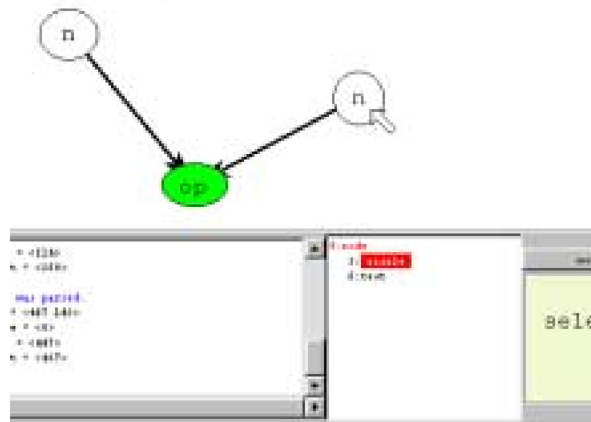

図 5.11:　非終端記号の認識

実際、図 5.12 のように円をドラッグして動かすと、内部の文字もついてくる。

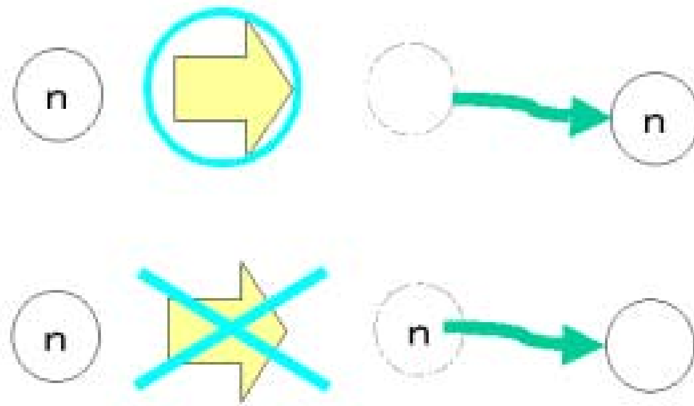

図 5.12: 非終端記号の認識

そこでそのままマウスの中ボタンをプレスするとポップアップメニューが表示される。そこから exist を選ぶ。（図 5.13）

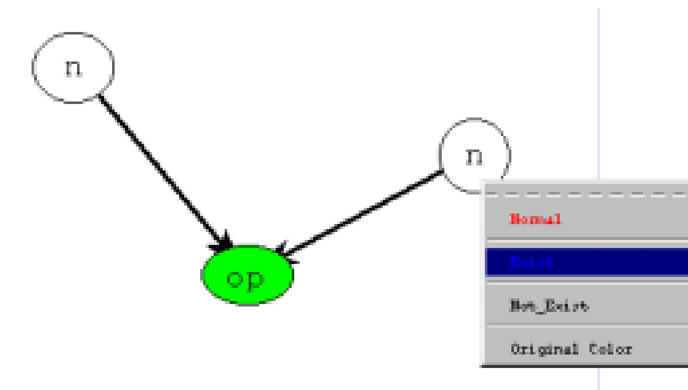

図 5.13: 構成要素の種類の変更

これで右ノードの種類の変更は完了である。

2.3 同様に左ノードと、中央のサークルと左右のノードをつないでいる 2 つのラインの種類も exist に変更する。

3. 次に制約を自動生成する。

3.1 生成規則 1 を定義したときと同じように座標制約だけを自動生成する。

3.2 制約リストで不必要な制約を無効にする。この際、現在ポインタが指しているリスト上の制約の対象となっている構成要素がバウンディングボックスに囲まれ例示入力図上に表示されるので、これを参照しながら行うと楽である。

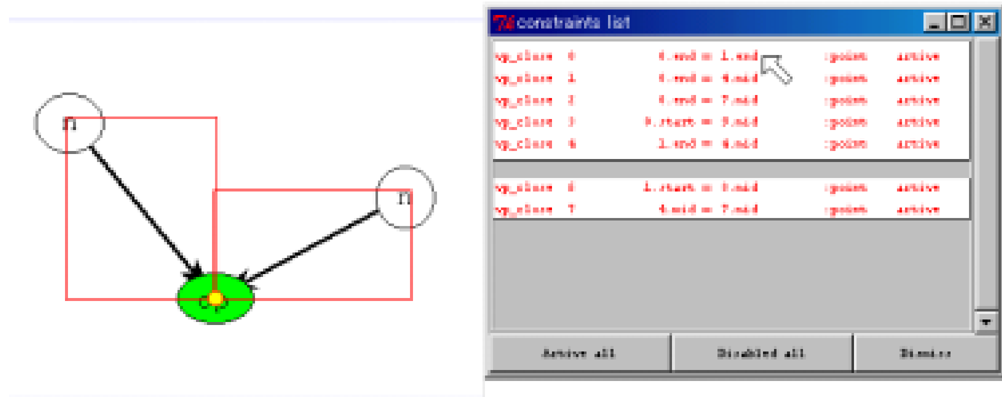

図 5.14: 制約の選択

図 5.14 は左右のラインの終点が一致しているという制約を表している。（左右のラインがボックスで囲まれ、終点に座標一致を示す円が表示されている）

4. 不足している制約を手動で入力する。

4.1 「円の内部が緑」という制約を生成規則 1 と同様に定義し入力する。

4.2 「右ノードの right が 左ノードの left より小さい」という制約を定義する。

4.2.1 まず左右のノードの属性リストを表示させる（図 5.15）

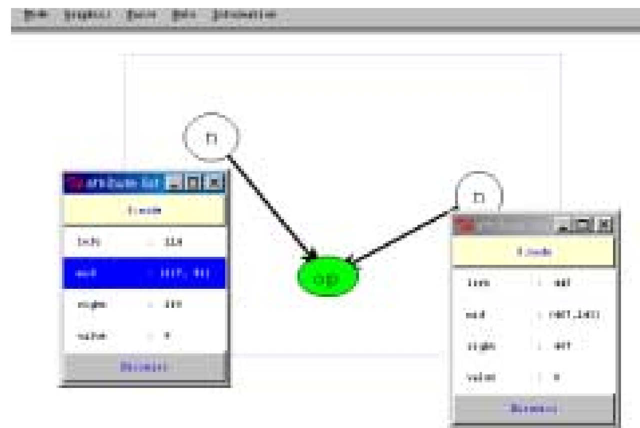

図 5.15: 属性リスト

4.2.2 左ノードの属性リストから「right」を選び、制約生成ウィンドウの Arg 1 フィールドにダブルクリックで入力。

4.2.3 右ノードの属性リストから「left」を選び、制約生成ウィンドウの Arg 2 フィールドにダブルクリックで入力。

4.2.4 そして制約の種類は「＜」を選び、「OK」ボタンを押して制約を入力。（図 5.16）

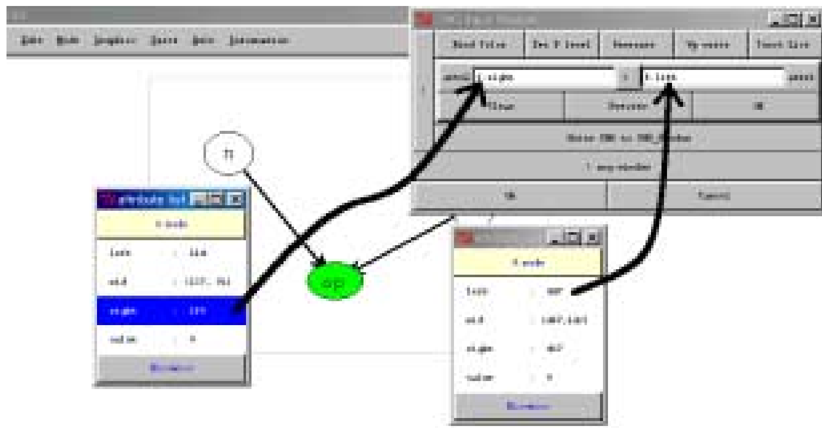

図 5.16: 制約入力

5. 最後に CMG ウィンドウに書き出して制約定義終了。あとは属性、アクションを定義して完成。

以上で生成規則 2 の定義は終了である。

これで計算の木の定義は完成である。

5.3 評価

恵比寿とエビ chu を比較して評価するために、より複雑な生成規則 2 に注目してみる。生成規則 2 の「制約」を定義するには CMG ウィンドウに表 5.1のように書かれている必要がある。

欄	内容
constraints	`vp_close 1.0.start 1.3.mid`
	`vp_close 1.1.start 1.2.mid`
	`vp_close 1.0.end 0.0.mid`
	`vp_close 1.1.end 0.0.mid`
	`vp_close 0.0.mid 0.1.mid`
	`eq 0.0.innercolor { string green }`
	`lt 1.2.right 1.3.right`
normal	`circle`
	`text`
exist	`line`
	`line`
	`node`
	`node`

表 5.1: 生成規則 2

ところが、従来の恵比寿では図形の例示入力を行って CMG ウィンドウを開くと表 5.2 のように描かれている

欄	内容
constraints	eq vp_close 0.0.end 0.1.end eq 0.0.linewidth 0.1.linewidth vp_close 0.0.start 0.3.mid vp_close 0.0.end 0.4.mid eq 0.0.end_y 0.5.mid_y vp_close 0.0.end 0.5.mid eq 0.0.start_y 0.7.mid_y vp_close 0.0.start 0.7.mid eq 0.1.start_y 0.2.mid_y vp_close 0.1.start 0.2.mid vp_close 0.1.end 0.4.mid vp_close 0.1.end 0.5.mid vp_close 0.1.start 0.6.mid eq 0.2.diametery 0.3.diameterx eq 0.2.radiusy 0.3.radiusx eq 0.2.linewidth 0.3.linewidth eq 0.2.linewidth 0.4.linewidth vp_close 0.2.mid 0.6.mid eq 0.3.linewidth 0.4.linewidth vp_close 0.3.mid 0.7.mid vp_close 0.4.mid 0.5.mid
normal	line line circle circle circle text text text

表 5.2: 従来の恵比寿

まず制約を見てみると、 linewidth などといった、実際には必要としない制約が非常に数多く自動生成されいている。次に構成要素をみると、 node という構成要素が書かれていない。このため circle と text を一組消して、新たに node と構成要素欄に書きこむ必要がある。また、構成要素の種類を変えるために line と新たに作った node を normal 欄から削除して、 exist 欄に書き移さなくてはならない。構成要素を書き換えると、その構成要素の識別 ID はそのほとんどが変わってしまうため、自動生成された制約はほとんど意味を成さない。結局白紙からテキストで書き起こすのとほとんど変わらない労力を要する。またテキストで記述するため、生成規則全体のイメージが直感的に理解できないうえ、誤字脱字や文法上のミスなども犯すおそれがある。

一方エビ chu は

1. ユーザ指定による制約の自動生成

2. 非終端記号の認識

3. 構成要素の識別 ID の自動管理

4. 制約や構成要素の種類の視覚的表現

5. その視覚的表現を用いた入力

といった機能を持つことにより、

1. 無駄な制約を削除する手間を省略

2. 構成要素入力の簡易化

3. ユーザが意識して識別 ID の整合性を保つ必要が無い

4. 構成要素の図形的位置関係や制約の直感的な把握が容易

5. 直感的理解とフルマウスオペレーションによる快適な操作

といった特徴をもつため、ユーザの負荷を大幅に軽減できると思われる。

そこで実験として実際に何人かの人に恵比寿とエビ chu のそれぞれで生成規則 2 の「制約」の定義をおこなってもらった。なお被験者には図 5.17 に示すような生成規則 2 の概要を示したメモを手渡し、既に生成規則 1 が定義され例示入力図が描かれた状態の恵比寿で実験を行なった。

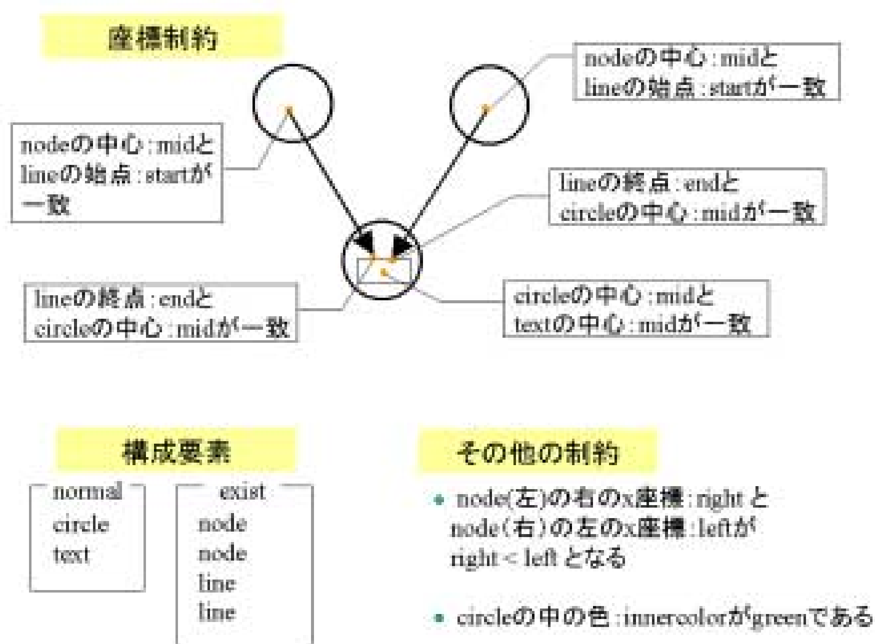

図 5.17: メモ

その所要時間を図 5.18 に示す

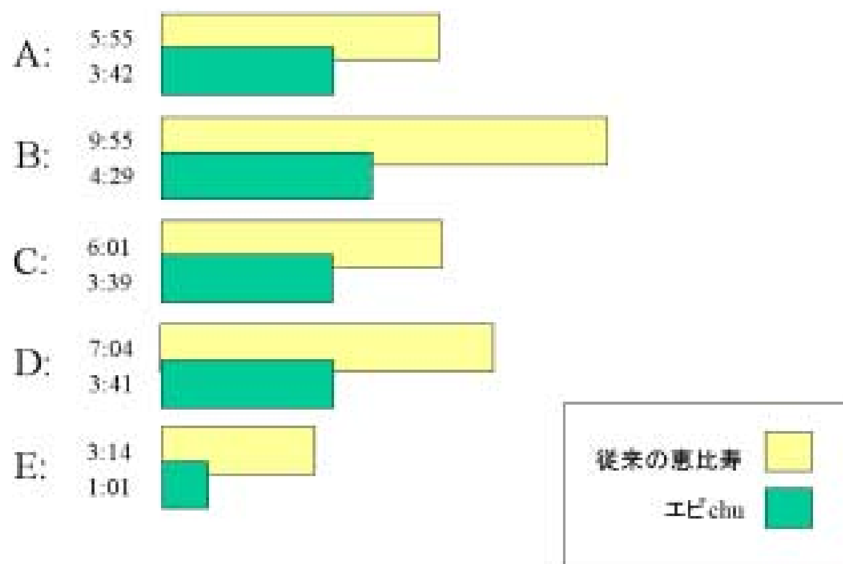

図 5.18: 作業所要時間

被験者 A ～ D については恵比寿は何度も使用したことがあるが、エビ chu は初めてという者達である。にもかかわらず、どの被験者も慣れているはずの従来の恵比寿でテキストで定義するよりも、初めて扱うエビ chu で定義した場合のほうが、半分程度の時間しかかかっていない。被験者 E のようにエビ chu の扱いに慣れた者であれば、作業時間を $\frac{1}{3}$ 以下にまで短縮することも可能である。

どの場合にしろ、恵比寿よりエビ chu のほうが作業効率が 2 倍ほどアップしていることが分かる。

これはエビ chu の使いやすさを実証している。

第 6 章

結論

本論文ではビジュアルシステム恵比寿が、2 次元的な情報をもつ図形言語を 1 次元的なテキストで入力するために生じるいくつかの問題点と、視覚的表現を用いることでこの問題を解決できることについて述べた。特に生成規則定義における「制約」の入力に注目し、この入力法について考察をおこない、視覚的な表現を利用したインタラクティブな編集を行う、より分かりやすい入力インターフェイスを提案した。また、この入力インターフェイスを恵比寿上に実装したシステム「エビ chu」を作成した。さらに「エビ chu」を用いて実際にビジュアルシステム「計算の木」を作成する例をあげ、その作成過程を通して従来の恵比寿と比較することにより「エビ chu」の評価を行った。

今後は「制約」の入力だけではなく、一度定義した制約を再表示して、やはり視覚的表現を用いて編集できるようしたいと考えている。そのためには制約から図形を再描画する必要があるが、それには TRIP2[13] のような方法が考えられる。また「制約」だけではなく、「属性」「アクション」の入力方法についても考察し、最終的により使いやすいビジュアルシステムを目指すために、さまざまな手法を採用し研究を進めていく予定である。

謝辞

本研究を進めるにあたり、終始ご指導下さった指導教官の田中二郎教授に心から感謝いたします。また研究全般においてなにかと助けていただいた恵比寿プロジェクトグループリーダーの飯塚和久さん、ならびに忙しいなかインターフェイスの評価を行ってくれたプロジェクトメンバーの丁錫泰さん、西名毅さん、奥村穂高さんにも感謝したいと思います。そして田中研究室の皆さんにはシステム作成にあたり多くの貴重な助言をいただきました。ここに感謝の意を表します。

参考文献

[1] Jiro Tanaka. PP:Visual Programming System for Parallel Logic Programming Language GHC. *Parallel and Distributied Computing and Networks '97*, pages 188-193, 1997.

[2] Nobuo Kawaguchi, Toshiki Sakabe, and Yasuyoshi Inagaki. TERSE:Term rewriting support enviroment. In *Workshop on ML and its Apprication,* pages 91-100, 1994.

[3] Yasunori Harada, Kenji Miyamoto, and Rikio Onai. VISPATCH: Graphical rule-based language controlled by user event. In *Proceedings of the 1997 IEEE Symposium on Visual Languages,* 1997.

[4] I. Yoshimoto, N. Monden, M. Hirakawa, M, Tanaka, and T. Ichikawa. Interactive Iconic Programming Facility in HI-VISUAL. In *Proceedings of the 1986 Workshop on Visual Languages,* pages 34-41, 1986

[5] M. Hirakawa, Y. Nishimura, M. kado, and T.Ichikawa. Interpretation of Icon Overlapping in Iconic Programming. In *Proceedings of the 1991 IEEE Workshop on Visual Languages,* pages 254-259, 1991.

[6] 松岡聡, 宮下健. 例示による GUI プログラミング. ビジュアルインタフェース, pages 79-97, 共立出版, 1996.

[7] 馬場昭宏. Spatial Parser Generator を持ったビジュアルシステム. 筑波大学大学院博士課程工学研科修士論文, 1998.

[8] 馬場昭宏, 田中二郎. Spatial Parser Generator を持ったビジュアルシステム. 情報処理学会論文誌 Vol.39, No.5, 1998.

[9] 馬場昭宏, 田中二郎. Spatial Parser Generator の Tcl/Tk を用いた実装. 情報処理学会シンポジウムインタラクション '97 論文集, pages 71-78, 1997.

[10] 馬場昭宏, 田中二郎. GUI を記述するためのビジュアル言語. インタラクティブソフトウェア V, 日本ソフトウェア学会 WISS'98, pages 135-140, 近代科学社, 1997.

[11] Kim Marriot. Constraint Multiset Grammars. In *Proceedings of the 1994 IEEE Symposium on Visual Languages,* pages 118-125, 1994

[12] 宮田重明・芳賀敏彦共著, Tcl/Tk プログラミング入門. オーム社, 1995.

[13] Satoshi Matsuoka, Shin Takahasi, Tomihisa Kamada, and Akinori Yonezawa. A General Framework for Bidirectinal Translation brtween Abstract and Pictorial Data. *ACM Transaction on Information Systems* Vol.10, No.4, pages 408-437, 1992.