筑波大学大学院博士課程

工学研究科修士論文

Spatial Parser Generator を持った ビジュアルシステム

電子・情報工学専攻

著者氏名　　馬場 昭宏

指導教官　　田中 二郎　印

平成 10 年 1 月

平成 9 年度 工学研究科修士論文概要

Spatial Parser Generator を持った ビジュアルシステム

指導教官　田中 二郎　印

電子・情報工学専攻　学籍番号 965366　馬場昭宏

ビジュアルプログラミングシステムや，特定用途の図形エディタでは対象となる図が図形を用いた言語（ビジュアル言語）であるためにビジュアル言語の解析をおこなう部分（Spatial Parser）を実装する必要があった．しかし個々のアプリケーションごとに Spatial Parser を実装するのは困難で時間のかかる仕事であった．また，これまでの Spatial Parsing アルゴリズムは解析をおこなうことに主眼がおかれ，その結果に基づいて何らかのコードを実行するというものは少なかった．

本研究では Spatial Parsing アルゴリズムの一つである CMG にアクションの概念を追加することで実際におこなった解析結果を利用した動作をおこなえるようにした．アクションはスクリプト言語が解釈可能な任意の文字列であると定義し，われわれはアクション中で図形の生成，削除，属性の変更などをスクリプトとして記述することによって図を描き換える方法を提案した．また，制約解消系と Spatial Parser Generator を持つことにより，ビジュアル言語の文法を動作を与えることで様々なビジュアル言語に対応することができるシステム「恵比寿」を作成した．恵比寿ではまずはじめに図形を用いて入力し，それから CMG を半自動的に生成するため，直感的に図形言語の文法と動作を定義できる．さらに恵比寿では一度解析が終了すると文法に基づいて図形間に制約が課せられる．恵比寿のインタフェースおよび Spatial Parser Generator は Tcl/Tk により実装され，制約解消系としては SkyBlue の C 言語による実装を用いている．本論文では恵比寿上でアプリケーションを作成する五つの例（計算の木，計算の木を描き換えるもの， GUI を作成するもの， VISPATCH， HI-VISUAL）を示した．最後に恵比寿上でのプログラミングとオブジェクト指向プログラミングおよび論理プログラミングにおける概念との対応についての考察をおこなった．

目 次

第 1 章

はじめに

現在様々な分野においてビジュアル言語が用いられている．ビジュアル言語というとビジュアルプログラミング言語を連想する場合が多いと思われるが，ビジュアルプログラミング言語はビジュアル言語のうちとくにプログラミングすることを目的としたものである．ビジュアル言語はこの他にも ER ダイアグラム [1]， OMT のオブジェクト図 [2]，回路図，状態遷移図，楽譜，数式，表，ベン図などからテレビドラマの登場人物の関係などを表わす図まで様々なものがある．回路図や ER ダイアグラム， OMT のオブジェクト図など，特定の用途の図は専用のエディタを用いて描くことが多かった．汎用のエディタ（ドローツール）を用いて描くと，エディタがその図の意味を知らないために使用者が多くの操作をしなくてはならないためである．たとえば円の中にラベルとして文字が書いてあるものをノード，それらのノード間をつなぐ直線をエッジとしてグラフを表現するようなものを考えると，ノードを動かすことを意図して円を動かしても，文字と直線はその円の動きに追随しない．これはビジュアル言語である回路図などを構造を持たない図として描こうとしているために生じる問題である．したがってこのような専用のエディタもビジュアル言語を処理するためのシステムであるとみなすことができる．それぞれのビジュアル言語を処理するためのシステムがあるとは限らない．ユーザが自分でビジュアル言語を定義し，その処理系を作成できることが望ましい．

プログラミングの初期の過程においてデータ構造やデータの流れ，処理の流れなどを表すのに図を用いることが多い．ビジュアルプログラミングシステムはこのような図をそのまま使ってプログラミングをしようというものである．しかし，既存のビジュアルプログラミングシステムは特定のビジュアルプログラミング言語の仕様に固定されており，変更は困難で，かつ時間のかかる仕事であった．

ここでテキスト言語でのプログラミングについて考えてみる．テキスト言語でプログラムを作るときには，まず「言語」によって記述し，「処理系」が翻訳または解釈することによって実行可能となる．言語は（処理系に特化した拡張は考えられるが）一般には処理系に依存していない．つまり，一つの言語に対して複数の処理系が存在し得る．ビジュアル言語でも同様に「言語」と「処理系」を切り離して考えれば，「処理系」に依存しない「言語」

を定義できる．このように処理系から切り離したビジュアル言語による文は二次元ならば一枚の紙の上に，三次元ならば模型として描く（作る）ことができるはずである．テキスト言語では「言語」と「処理系」の分離によって Parser Generator を作ることが可能となった．ビジュアル言語の処理系でもこれまでのようなアドホックな方法ではなく，ビジュアル言語の文法に従ってビジュアル言語を解析する部分（Spatial Parser）を作るという考え方に基づけば，より普遍的な方法でビジュアル言語を解析することができ，このことを利用してビジュアル言語の文法を記述することにより Spatial Parser を生成する Spatial Parser Generator を実装することができる．

しかし実際のビジュアル言語の処理系ではビジュアル言語の解析のみならず，解析結果に応じた動作をおこない，図形間に幾何制約を課すことによって意味的関係を保存し，さらには描いた図形へのフィードバックをおこなうことが必要とされる．われわれは Spatial Parser Generator と制約解消系を持つことでこれらの要求を満たすようなシステム「**恵比寿**」[3] を作成した．恵比寿ではビジュアル言語の文法とその動作を与えることでビジュアルシステムを記述できる．

本論文の構成は以下の通りである．2 章では本論文で必要となる知識を準備する．3 章では CMG へのアクションの導入について述べる．4 章で実装したシステム「恵比寿」について述べ，5 章ではその実装について述べる．次に 6 章で恵比寿を用いて作成したビジュアルシステムの例を挙げる．7 章では恵比寿上のプログラミングパラダイムに関する考察をおこなう．8 章では関連研究について述べる．

第 2 章

準備

本章ではまず本論文中で使用する用語を定義し、次に要素技術とその動向について述べる．

2.1 用語の定義

ビジュアル言語に関する用語をテキスト言語と比較しながら定義する．

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．これに対して円や直線などの図形を二次元，もしくはそれ以上の次元に配置するものを**ビジュアル言語**と呼ぶ．ビジュアル言語におけるこれらの基本的な図形のことを**図形文字**と呼ぶことにする．各図形文字は，種類，色，大きさ，位置などの属性を持つ．

テキスト言語ではある概念を単語として表すが，同様にビジュアル言語にも単語に相当するものがあると考えられる．ビジュアル言語ではある概念をいくつかの図形を組み合わせて作った意味のある一つの「もの」を表現するためのシンボルとして表すのが一般的である．本論文ではビジュアル言語におけるこのようなシンボルを**図形単語**と呼ぶことにする．図形単語も属性を持つ．一つの図形単語を作るためのいくつかの図形を構成要素と呼ぶことにする．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**と呼ぶ．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呼ぶことにする．

ビジュアル言語のうちとくにプログラミングをするための言語をビジュアルプログラミング言語と呼ぶ．

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**と呼ぶ．一口に「処理する」と言っても多くの意味が考えられる．たとえば回路図を描くとそのシミュレーションをする，楽譜を描くとそれを演奏する，数式を描くと計算をする，もしくはその数式を表わす LaTeX のソースコードを生成する，などである．このような処理の内容に関してはここでは定義せず，任意のものであるとする．

2.2 要素技術

本節では 1章で述べたようなシステムの作成に際してどのような要素技術が必要であるかということについて例を通して考察する．例として，計算の木を実行するようなシステムを作成することを考える．これはもっとも大まかな言い方をすると「図 2.1のような図を与えると結果として 35 を返す」というものである．より詳細な仕様を述べる．入力となる図は円の中に数字または演算子が書かれたもの（ノード）とそれらを結ぶ矢印（エッジ）から構成される．数字のノードに入るエッジはなく，演算子のノードに入るエッジはちょうど二本である．各ノードから出ていくエッジはないか，もしくは一本である．ノードから出ていくエッジがない場合はそこで計算が終了する．エッジがある場合はそのエッジがつながっているもう一方のノードへの引数として用いられる．各ノードは値を持つ．数字のノードの値は書かれた数字の値である．演算子のノードの値は，入ってくる二つのエッジによって結ばれているノードが持つ値を引数として演算子のノードに書かれた演算子によって計算した値である．このとき左にあるノードを第一引数，右にあるノードを第二引数とする．各図形は円，テキスト文字列，矢印などの単位で入力するが，一度描いたノードは一つのものとして移動できるものとする．つまり，ノードの一部である円を移動させると，そのノードのもう一つの構成要素であるテキスト文字列もそれに追随して移動し，またそのノードにつながっているエッジも同時に追随するものとする．

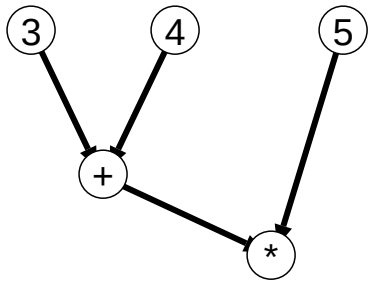


図 2.1: 計算の木

このようなシステムは，

1. 「つながっている」「中にある」「左にある」といった図形の位置関係を調べてそれら複数の図形から構成される新しい単位（図形単語）として扱い，
2. そのような関係を保存する

という二つの部分に大きく分けることができる．

1は Spatial Parsing と呼ばれ，テキスト言語の構文解析に対応するものである． 2のためには各図形の間に成り立っている制約が常に成り立つように維持する**制約解消系**が必要である．

ビジュアル言語の文法を記述する方法とそれに基づいて Spatial Parsing をおこなうアルゴリズムとしては Picture Layout Grammars（PLG）[4]， Positional Grammars（PG）[5]， Relation Grammars（RG）[6]， Constraint Multiset Grammars（CMG）[7] などが知られている．以下の節ではとくに PLG， RG， CMG をとりあげ，これらの方法でどのようにして文法を記述するのかについて述べる．

2.3 Relation Grammars（RG）

Relation Grammar における図形文 w の定義は以下の通りである [6].

図形文字の集合 V_T と関係記号（制約を表す述語）の集合 V_R を用いて V_T と V_R 上の文 w は組 $< \sigma, \rho >$ として定義される． σ は s- アイテム（本論文ではトークン）(v, i) の集合である．ただし $v \in V_T$， $i \in N$ であり， N は自然数の集合とする． σ 中の s- アイテム (v, i) は v^i と略記する． i は同じ図形単語を区別するのに用いられる． ρ は r- アイテムの集合であり， $r(t_1, t_2)$ の形をしている．ここで t_1， $t_2 \in \sigma$ であり， $r \in V_R$ は r が t_1 と t_2 の間に成り立つことを示している．

図 2.1は $V_T = \{\bigcirc, \nearrow, text\}$ と $V_R = \{cont, cae, cas\}$ 上の文 $w =< \sigma, \rho >$ として図 2.2のように表わすことができる．

$$
\begin{aligned}
\sigma &= \{\bigcirc^1, \bigcirc^2, \bigcirc^3, \bigcirc^4, \bigcirc^5, text^1, text^2, text^3, text^4, text^5, \nearrow^1, \nearrow^2, \nearrow^3, \nearrow^4\} \\
\rho &= \{cont(\bigcirc^1, text^1), cont(\bigcirc^2, text^2), cont(\bigcirc^3, text^4), cont(\bigcirc^4, text^4), \\
&\quad cont(\bigcirc^5, text^5), cas(\nearrow^1, \bigcirc^1), cas(\nearrow^2, \bigcirc^2), cas(\nearrow^3, \bigcirc^3), cas(\nearrow^4, \bigcirc^4), \\
&\quad cae(\nearrow^1, \bigcirc^3), cae(\nearrow^2, \bigcirc^3), cae(\nearrow^3, \bigcirc^5), cae(\nearrow^4, \bigcirc^5)\}
\end{aligned}
$$

図 2.2: 図 2.1を表わす Relation Grammar の文

図 2.2において r- アイテム $cont(t_1, t_2)$ は t_1 が t_2 を包含する（contain）ことを， $cas(t_1, t_2)$ は t_1 の始点と t_2 の中点が一致している（Connected At Start point）ことを， $cas(t_1, t_2)$ は t_1 の終点と t_2 の中点が一致している（Connected At End point）ことをそれぞれ示している．この例によってわかるとおり， Relation Grammar においては図形文字や図形単語に属性はない．そのため値による制約，たとえば円の半径がある一定の大きさ以上であるとか，文字列の値が 3 であるといったことを書くことができない．

Relation Grammar は次のような六つ組み G として定義される [6].

$$G = (V_N, V_T, V_R, S, P, R)$$

ここで， V_N は図形単語の有限集合， V_T は図形文字の有限集合， V_R は関係記号の有限集合， S は開始記号， P は s- 生成規則の有限集合， R は r- 生成規則の有限集合である．

s- 生成規則は $A ::= \sigma, \rho$ の形をしている．ここで，$A \in V_N$ であり，σ は s- アイテム (v, i) の集合である．ただし $v \in V_N \cup V_T$，$i \in N$ である．ρ は r- アイテムの集合である．r- アイテムの中では σ 中の s- アイテムが使われる．s- 生成規則はインデックスによってラベルが付けられている．r- 生成規則は $rel(x_1, x_2) ::= [p, j], v$ の形をしている．p は x_j（ただし $j \in \{1, 2\}$）に対して適用される s- 生成規則のインデックスである．v は r- 生成規則が適用されたときに r- アイテム $rel(x_1, x_2)$ を置き換える r- アイテムの集合である．

図形単語の s- アイテムは s- 生成規則によって生成される．このとき，図形単語の r- アイテムが r- 生成規則によって生成される．ここで，その r- 生成規則は p としてその s- 生成規則を持つものである．

例として RG $CompTree_{RG} = (\{Node\}, \{\bigcirc, \nearrow, text\}, \{cont, cas, cae\}, S, P, R)$ を定義する．P は次のような s- 生成規則を含む．

$$
\begin{aligned}
[1]Node &::= \{\bigcirc^1, text^1\}\{cont(\bigcirc^1, text^1)\} \\
[2]Node &::= \{Node^1, Node^2, text^1, \bigcirc^1, \nearrow^1, \nearrow^2\} \\
&\quad \{cas(\nearrow^1, Node^1), cas(\nearrow^2, Node^2), cae(\nearrow^1, \bigcirc^1), cae(\nearrow^2, \bigcirc^1)\}
\end{aligned}
$$

R は次のような r- 生成規則を含む．

$$
\begin{aligned}
cas(\nearrow, Node) &::= [1, 2], \{cas(\nearrow, \bigcirc^1)\} \\
cas(\nearrow, Node) &::= [2, 2], \{cas(\nearrow, \bigcirc^1)\}
\end{aligned}
$$

この文法 $CompTree_{RG}$ を用いて図 2.1を解析していく過程を示す．初期状態は図 2.2に示したとおりである．まず s- 生成規則 [1] を $\bigcirc^1$，$text^1$，$cont(\bigcirc^1, text^1)$ に対して適用する．このとき，一番目の r- 生成規則によって $cont(\nearrow^1, \bigcirc^1)$ は，$cont(\nearrow^1, Node^1)$ に書き換えられる．この時点で

$$
\begin{aligned}
\sigma &= \{\bigcirc^2, \bigcirc^3, \bigcirc^4, \bigcirc^5, text^2, text^3, text^4, text^5, \nearrow^1, \nearrow^2, \nearrow^3, \nearrow^4, Node^1\} \\
\rho &= \{cont(\bigcirc^1, text^1), cont(\bigcirc^2, text^2), cont(\bigcirc^3, text^4), cont(\bigcirc^4, text^4), \\
&\quad cont(\bigcirc^5, text^5), cas(\nearrow^2, \bigcirc^2), cas(\nearrow^3, \bigcirc^3), cas(\nearrow^4, \bigcirc^4), cae(\nearrow^1, \bigcirc^3), \\
&\quad cae(\nearrow^2, \bigcirc^3), cae(\nearrow^3, \bigcirc^5), cae(\nearrow^4, \bigcirc^5), cas(\nearrow^1, Node^1)\}
\end{aligned}
$$

となる．さらに同様に生成規則を適用していくことによって図 2.3に示すような解析木を得る．

効率的な解析のためには決定性が重要である．つまり，後戻りすることなく解析がおこなわれるべきである．しかし RG では $CompTree_{RG}$ を決定性のある文法として書くことはできず，演算子のノードが数字のノードとして解析されてしまう可能性がある．

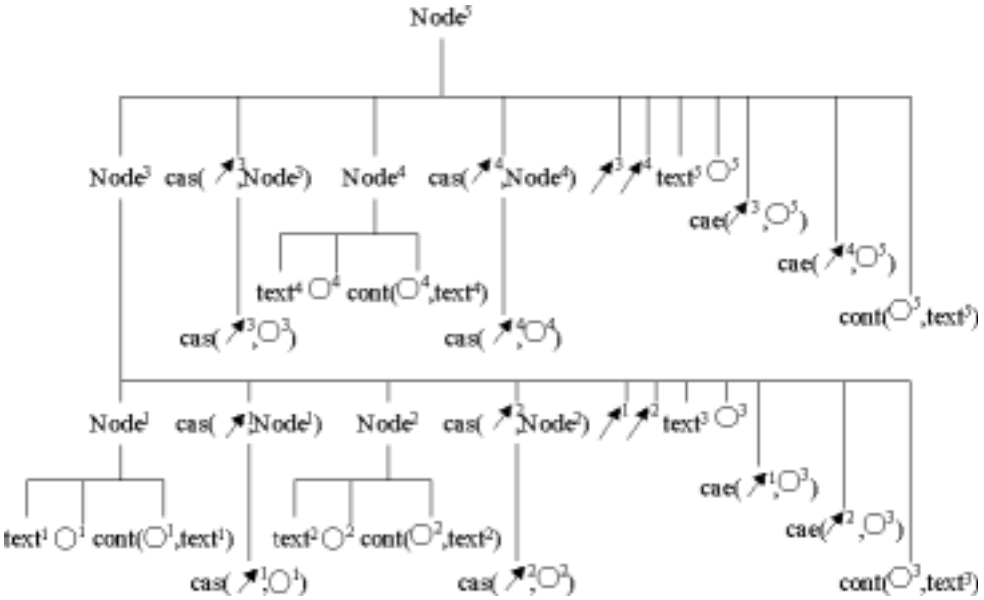


図 2.3: 図 2.2に対する解析木

2.4 Picture Layout Grammars（PLG）

PLG における図形文はトークンのマルチセットである．すべての図形単語及び図形文字は必ずそれに外接する長方形の左上の点の x 座標および y 座標と右下の点の x 座標および y 座標を属性として持つ．またその他に図形単語もしくは図形文字独自の属性を持つこともある．図 2.1は PLG の図形文として図 2.4のように表される．

PLG は Attributed Multiset Grammars（AMG）に基づいている． AMG は六つ組 $G = (N, \Sigma, s, I, \mathcal{D}, P)$ である．ここで， N は図形単語の有限集合， Σ は図形文字の有限集合， $s \in N$ は開始シンボル， I は属性の集合， $\mathcal{D}$ は属性の定義域の集合， P は生成規則の集合を表す．一つの生成規則は三つ組 (R, SF, C) である．ここで， R は $A \rightarrow M_1/\gamma$ の形をした書き換え規則であり， A は図形単語， M_1 は図形単語もしくは図形文字のマルチセット， γ は生成規則の文脈を形成する図形文字のマルチセットである． SF は右辺の図形単語もしくは図形文字の属性から左辺の図形単語の属性の値を決めるための関数， C は右辺の属性の間に成り立つべき制約である． PLG は生成規則によって図を構成するような AMG である．

例として次のような PLG$CompTree_{PLG} = (N, \Sigma, s, I, \mathcal{D}, P)$ を定義する．

$N = Node$， $\Sigma = \{circle, text, line\}$， $s = Node$， $I = \{lx, ty, rx, by, text, value\}$， $\mathcal{D} = \{\{lx, ty, rx, by \in integer\}, \{text \in string\}\}$ で， P は次に示すような生成規則から構成される．

$$Node \quad \rightarrow \quad \{circle, text\}/\{\}$$

$\{circle[lx = 10, ty = 10, rx = 30, by = 30], circle[lx = 50, ty = 10, rx = 70, by = 30],$

$circle[lx = 110, ty = 10, rx = 130, by = 30], circle[lx = 30, ty = 50, rx = 50, by = 70],$

$circle[lx = 90, ty = 70, rx = 110, by = 90],$

$text[lx = 17, ty = 13, rx = 23, by = 27, text = \text{'3'}],$

$text[lx = 57, ty = 13, rx = 63, by = 27, text = \text{'4'}],$

$text[lx = 117, ty = 13, rx = 123, by = 27, text = \text{'5'}],$

$text[lx = 37, ty = 53, rx = 43, by = 67, text = \text{'+'}],$

$text[lx = 97, ty = 73, rx = 103, by = 87, text = \text{'*'}],$

$line[lx = 20, ty = 20, rx = 40, by = 60], line[lx = 60, ty = 20, rx = 40, by = 60],$

$line[lx = 40, ty = 60, rx = 100, by = 80], line[lx = 120, ty = 20, rx = 100, by = 80]\}$

図 2.4: 図 2.1を表す Picture Layout Grammar の図形文

$Node.value = text.text, Node.lx = circle.lx, Node.ty = circle.ty$

$Node.rx = circle.rx, Node.by = circle.by$

$where$

$contains(circle, text)$

$Node \rightarrow \{Node_1, Node_2, line_1, line_2, circle, text\}/\{\}$

$Node.value = Node_1.value + Node_2.value, Node.lx = circle.lx,$

$Node.ty = circle.ty, Node.rx = circle.rx, Node.by = circle.by$

$where$

$text.text == \text{'+'}, contains(circle, text)$

$line_1.lx == (Node_1.lx + Node_1.rx)/2, line_1.ty == (Node_1.ty + Node_1.by)/2$

$line_2.lx == (Node_2.lx + Node_2.rx)/2, line_2.ty == (Node_2.ty + Node_2.by)/2$

$line_1.rx == (circle.rx + circle.rx)/2, line_1.by == (circle.ty + circle.by)/2$

$line_2.rx == (circle.rx + circle.rx)/2, line_2.by == (circle.ty + circle.by)/2$

$Node \rightarrow \{Node_1, Node_2, line_1, line_2, circle, text\}/\{\}$

$Node.value = Node_1.value * Node_2.value, Node.lx = circle.lx,$

$Node.ty = circle.ty, Node.rx = circle.rx, Node.by = circle.by$

$where$

$$text.text == \text{'*'}, contains(circle, text)$$

$$line_1.lx == (Node_1.lx + Node_1.rx)/2, line_1.ty == (Node_1.ty + Node_1.by)/2$$

$$line_2.lx == (Node_2.lx + Node_2.rx)/2, line_2.ty == (Node_2.ty + Node_2.by)/2$$

$$line_1.rx == (circle.rx + circle.rx)/2, line_1.by == (circle.ty + circle.by)/2$$

$$line_2.rx == (circle.rx + circle.rx)/2, line_2.by == (circle.ty + circle.by)/2$$

$CompTree_{PLG}$ を用いて図 2.4を解析していく過程を図 2.5に示す.

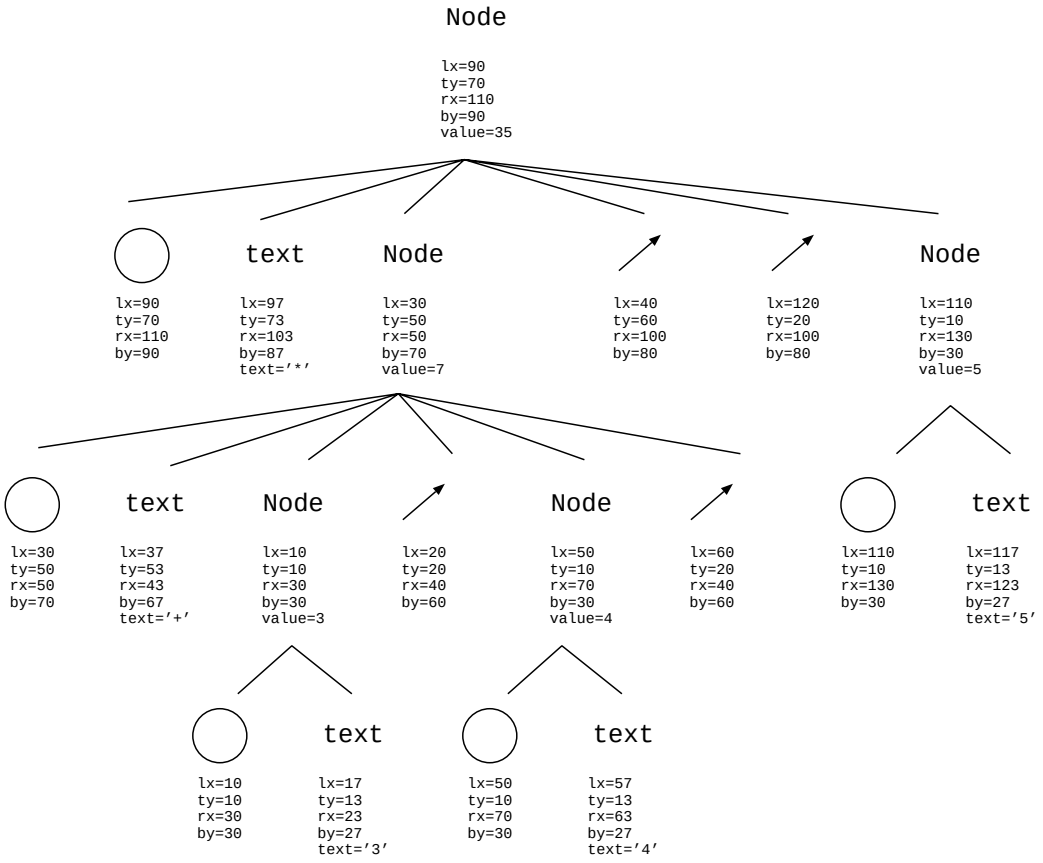


図 2.5: 図 2.4に対する解析木

この $CompTree_{PLG}$ も $CompTree_{RG}$ と同様に決定性はない．円の属性として線の太さや塗り潰す色などを考慮することによって $CompTree_{PLG}$ に決定性を持たせることは可能ではあるが，これは図形文を描くときの自由度を下げることになるのであまりいいアイデアではない.

2.5 Constraint Multiset Grammars（CMG）

トークンは$T(\vec{\theta})$と書かれる．Tは型記号であり，$\vec{\theta}$はTの属性への代入である．Tが終端シンボル（図形文字），非終端シンボル（図形単語），開始シンボルであるとき，そのトークンはそれぞれ終端トークン，非終端トークン，開始トークンと呼ばれる．図形文は，トークンのマルチセットである．終端トークンのみを含む図形文を終端文と呼ぶ．図 2.1はCMG の図形文として図 2.6のように表される．

$$
\begin{aligned}
&\{circle(mid=(20,20),radius=10), circle(mid=(60,20),radius=10),\\
&circle(mid=(120,20),radius=10), circle(mid=(40,60),radius=10),\\
&circle(mid=(100,80),radius=10), text(mid=(20,20),text=\text{'3'}),\\
&text(mid=(60,20),text=\text{'4'}), text(mid=(120,20),text=\text{'5'}),\\
&text(mid=(40,60),text=\text{'+'}), text(mid=(100,80),text=\text{'*'}),\\
&line(start=(20,20),end=(40,60)), line(start=(60,20),end=(40,60)),\\
&line(start=(40,60),end=(100,80)), line(start=(120,20),end=(100,80))\}
\end{aligned}
$$

図 2.6: 図 2.1を表す Constraint Multiset Grammar の図形文

CMG は図形文字の集合T_T，図形単語の集合T_{NT}，開始記号$S_T \in T_{NT}$，生成規則の集合Pから構成される．すべての図形単語及び図形文字$t \in T_T \cup T_{NT}$は属性を持っている．開始記号は生成規則の左辺にしか現れない．生成規則は次の形をしている．

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

ここで，Tは図形単語，T_1，$\cdots$，T_nは$n \geq 1$であるような図形単語もしくは図形文字，T'_1，$\cdots$，T'_mは$m \geq 0$であるような図形単語もしくは図形文字，$\vec{x}$，$\vec{x_i}$，$\vec{x'_i}$は異なる変数，Cは$\vec{x_1}, \cdots, \vec{x_n}, \vec{x'_1}, \cdots, \vec{x'_m}$の間の制約の連接，$F$は$\vec{x_1}, \cdots, \vec{x_n}, \vec{x'_1}, \cdots, \vec{x'_m}$を引数とする関数である．RG ではトークン間に制約が課せられていたが，CMG および PLG ではトークンの属性の間に制約が課せられていることに注意する．$m = 0$のとき，その生成規則はローカルであるという．ある文法のすべての生成規則がローカルであるとき，その文法はローカルであるという．

例として次のような CMG$CompTree_{CMG} = (T_T, T_{NT}, P)$を定義する．
$T_T = \{circle, text, line\}$，$T_{NT} = \{Node\}$であり，$P$は次のような生成規則から構成される．

$$
\begin{aligned}
&Node(N_{mid}, N_{value}) ::= \\
&\quad circle(C_{mid}, C_{radius}),\ text(T_{mid}, T_{text})\ where \\
&\quad C_{mid} = T_{mid}\ and \\
&\quad N_{mid} = C_{mid},\ N_{value} = T_{text}. \\
&Node(N_{mid}, N_{value}) ::= \\
&\quad Node(O_{mid}, O_{value}),\ Node(P_{mid}, P_{value}),\ circle(C_{mid}, C_{radius}), \\
&\quad text(T_{mid}, T_{text}),\ line(L_{start}, L_{end}),\ line(M_{start}, M_{end})\ where \\
&\quad O_{mid} = L_{start},\ P_{mid} = M_{start},\ C_{mid} = L_{end}, \\
&\quad C_{mid} = M_{end},\ T_{text} = \text{'+'},\ C_{mid} = T_{mid}\ and \\
&\quad N_{mid} = C_{mid},\ N_{value} = O_{value} + P_{value}. \\
&Node(N_{mid}, N_{value}) ::= \\
&\quad Node(O_{mid}, O_{value}),\ Node(P_{mid}, P_{value}),\ circle(C_{mid}, C_{radius}), \\
&\quad text(T_{mid}, T_{text}),\ line(L_{start}, L_{end}),\ line(M_{start}, M_{end})\ where \\
&\quad O_{mid} = L_{start},\ P_{mid} = M_{start},\ C_{mid} = L_{end}, \\
&\quad C_{mid} = M_{end},\ T_{text} = \text{'*'},\ C_{mid} = T_{mid}\ and \\
&\quad N_{mid} = C_{mid},\ N_{value} = O_{value} * P_{value}.
\end{aligned}
$$

生成規則は次のように記述することもある.

$$
\begin{aligned}
&Node(mid, value) ::= \\
&\quad circle : C(mid, radius),\ text : T(mid, Ttext)\ where \\
&\quad C.mid = T.mid\ and \\
&\quad mid = C.mid,\ value = T.text.
\end{aligned}
$$

$CompTree_{CMG}$ を用いて図 2.6を解析していく過程を図 2.7に示す.

G を CMG とする. 次に示すような変数を含まない G の生成規則のインスタンスを持つとき, 図形文 S が図形文 S' を生成するといい, $S \Rightarrow_G S'$ と書く.

$$
\begin{aligned}
T(\vec{\theta}) \quad ::= \quad & T_1(\vec{\theta_1}), \cdots, T_n(\vec{\theta_n})\ where \\
& exists\ T'_1(\vec{\theta'_1}), \cdots, T'_m(\vec{\theta'_m}) \\
& where\ C\ and\ \vec{\theta}\ =\ F.
\end{aligned}
$$

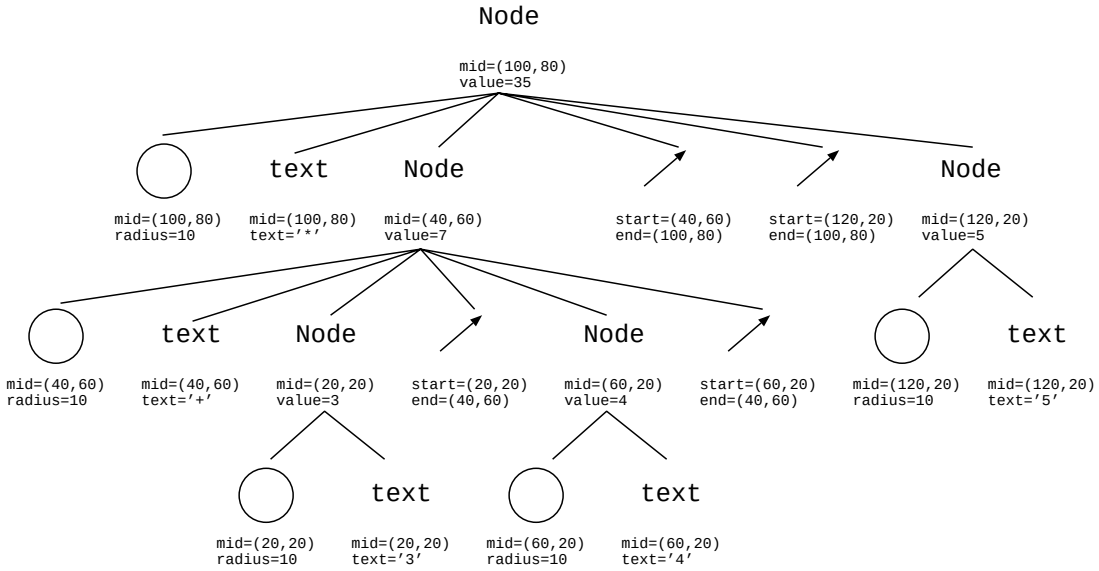


図 2.7: 図 2.6に対する解析木

$\{T(\vec{\theta}), T'_1(\vec{\theta'_1}), \cdots, T'_m(\vec{\theta'_m})\} \subseteq S, \vec{\theta} = F$ であり， $S' = S \cup \{T_1(\vec{\theta_1}), \cdots, T_n(\vec{\theta_n})\} \backslash \{T(\vec{\theta})\}$ である．ここで $\subseteq$， $\cup$， $\backslash$ はそれぞれマルチセットの包含，和演算，差演算である．

逆に， $S \Rightarrow_G S'$ ならば文の解析をおこなう場合は，

$$S = S' \cup T \backslash \{T_1(\theta_1), \cdots, T_n(\theta_n)\}$$

である．

G の言語 $lang(G)$ は $\{S_T(\vec{\theta})\} \Rightarrow^*_G S$ （ここで $S_T(\vec{\theta})$ は開始トークン）であるような終端文 S の集合である．

normal と exist のみの CMG では $CompTree_{CMG}$ を決定性のある文法として書くことはできない．ここまでの表現力は PLG と同等であるといえる．しかし CMG ではネガティブ制約を用いることで決定性を持たせることができる．

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

と書くと， $T''_1(\vec{\theta_1})$ から $T''_\ell(\vec{\theta_\ell})$ すべてが制約を満たさないときに限り生成規則が適用される．

ネガティブ制約を用いると先の例の一番目の生成規則は次のように書き換えられる．

$$
\begin{aligned}
&Node(N_{mid}, N_{value}) ::= \\
&\quad circle(C_{mid}, C_{radius}),\ text(T_{mid}, T_{text})\ where \\
&\quad not\ exists\ line(L_{start}, L_{end})\ where \\
&\quad C_{mid} = T_{mid},\ C_{mid} = L_{end}\ and \\
&\quad N_{mid} = C_{mid},\ N_{value} = T_{text}.
\end{aligned}
$$

円に入ってくる矢印がないことを明示することにより，演算子のノードと数字のノードを区別している．

さらに，コレクション生成規則を次のように定義する．

$$
\begin{aligned}
T(\vec{x}) \ ::= \ & all\ T'''_1(\vec{x'''_1}), \cdots, all\ T'''_k(\vec{x'''_k}) \\
& \vec{x} \ = \ F.
\end{aligned}
$$

F は $T'''_1(\vec{x'''_1})$，$\cdots$，$T'''_k(\vec{x'''_k})$ のうちいずれか一つの種類のトークンの属性のうちいくつかから成るものを一つの要素とし，その種類のトークンすべてにわたってそのような要素を集めたマルチセットを作る関数である．

第 3 章

CMG とその拡張

3.1 CMG と他の形式化との比較

われわれは図形の文法の記述方法として Constraint Multiset Grammars（CMG）を採用することとした．CMG では表現できるビジュアル言語のクラスが広い．PG では一つの図形単語が持っているのは一つの点のみであるが CMG では任意個の点を図形単語の属性として持つことができる．RG では二つの図形単語間の関係しか記述できないが CMG では任意の数の図形単語の属性間の関係を記述できる．CMG の考案者である Marriott は [8] で PG，RG および Unification Grammars[9] を用いて表した文法はすべて CMG を用いて書き換えることができることを示している．

また，われわれはユーザが自分でビジュアル言語を定義できることを目標としているので，記述，修正，理解が容易であることが重要であると考えた．CMG 以外の記法では図形単語の属性，構成要素の種類と数，構成要素の属性間の制約などをすべて混在させて記述するので理解が困難であるが，CMG ではこれらを別々に記述するため比較的わかりやすい．

3.2 CMG の解析アルゴリズム

Spatial Parsing のアルゴリズムには CMG に付属のものを使用した．後の説明のために [10] からアルゴリズム（図 3.1）を引用し，簡単に説明する．なおここでは変数はイタリックで表記する．

図 3.1に示したアルゴリズムは新たな図形文字が入力されるたびにインクリメンタルに解析がおこなわれるアルゴリズムである．*ParseForest* は生成規則が適用される対象となるトークンデータベースである．*SCC* は生成規則のデータベースで，*SCC* 中では低い階層の図形単語を作るための生成規則から順に並べられている．手続き AddToken はトークンデータベースに与えられた図形文字 t を挿入するものである．手続き EvalRule は生成規則 R を使って新たな図形単語を *ParseForest* に挿入し，生成に用いられた図形単語を *ParseForest* から削除するものである．

```
procedure Parse(t)
  AddToken(t)
  for each SCC in order do
    repeat
      for each rule R in the SCC do
        EvalRule(R)
      end for
    until ParseForest is unchanged
  end for
end
```

図 3.1: CMG 解析アルゴリズム

3.3 CMG へのアクションの導入

本研究ではスクリプト言語の存在を仮定し，CMG の生成規則にアクションを導入した．アクションを持った CMG の生成規則は次のようになる．

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

Action 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」として定義する．アクションの中では，属性名の最初と最後を@ で囲んだ部分はスクリプト言語に渡される前に構成要素や新たに生成されたトークンの属性の値を表す文字列として扱われる．また，トークンの名前の最初と最後を@ で囲むとそのトークンに付けられた id を表す文字列として扱われる．この id はトークンの属性の値の変更など，図形文へのフィードバックをおこなうときに用いられる． script 指令も導入するが，これについては例の中で詳しく述べる．

図 2.1の例のようなビジュアル言語は自然言語による次のような二つの規則によって表すことができる．

1. ノードは円の中に数字が書いてある．

2. ノードは 2 つのノードが矢印によって円につながれ，その円の中には演算子が書かれている．

これをわれわれの拡張 CMG で表すと次のようになる．

```
 1:node(string sort,point mid,
 2:      integer value) ::=
 3:   C:circle,T:text
 4:   where (
 5:      C.mid == T.mid
 6:   ) {
 7:      sort = {string "number"};
 8:      mid = C.mid;
 9:      value = to_int(T.text);
10:   } {
11:      display("@sort@ was parsed.");
12:      display("value = @value@");
13:   }
14:
15:node(string sort,point mid,
16:      integer value) ::=
17:   N1:node,N2:node,L1:line,
18:      L2:line,C:circle,T:text
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:   ) {
26:      sort = {string "operator"};
27:      mid = C.mid;
28:      value = {script.integer {
29:         @N1.value@@T.text@@N2.value@}};
30:   } {
31:      display("@sort@ was parsed.");
32:      display("value = @value@");
33:   }
```

1 行目から 13 行目までが数字を表すノードの定義である． 1， 2 行目ではその図形単語の名前と属性を定義している．名前は node であり，三つの属性 sort， mid， value を持ち，それらの型はそれぞれ string， point， integer である． 3 行目では図形単語がどのような構成要素から構成されるのかを定義している．ここでは，一つの円（circle）と，一つの文字列（text）から構成されるということを定義している． 4 行目から 6 行目は，図形単語の構成要素間の制約を表している．ここでは円 C の mid という属性の値と，文字列 T の mid という属性の値が等しいという制約を表している． 7 行目から 10 行目は制約が成り立ったときに，定義中の図形単語の属性にどのような値を与えるかということを定義している． 7 行目では属性 sort に「number」という文字列定数を与えることを， 8 行目では mid と円 C の mid が常に等しくなることを， 9 行目では value に文字列 T の属性 text を integer 型に変換して代入することを定義している． 11 行目と 12 行目が CMG を拡張した部分「アクション」である．この図形単語の属性 sort の値が「number」で，属性 value の値が「3」であり， display が表示するための命令であれば，この図形単語が認識されたときには，

```
number was parsed.
value = 3
```

と表示される． 15 行目以降では，演算子に二つのノードがつながっているものを定義している．ほとんどは数字を表すノードと同じであるが， 28， 29 行目で script 指令を用いている． script 指令は複数の構成要素の値から図形単語の属性の値を生成するのに用いられる．script 指令中ではアクションと同様の方法によって複数の構成要素の値を参照し，一つの文字列にする．生成された文字列をスクリプト言語によって解釈させ，その結果を得る．結果の型はここでは特に規定しない．得られた結果を図形単語の属性の型に変換したものをその図形単語の属性の値とする． script 指令を使った属性値の書き方は {`script.`*type* {*script*}} という形式である．ただし， *script* はスクリプト言語に解釈させる文字列で， *type* は得られた結果を今後どの型として扱うかを示す型の名前である． script 指令がないとすると，あらかじめ CMG 自体の解析系に準備しておいた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，スクリプト言語が解釈できる文字列さえ生成できれば後はスクリプト言語によって値を求められる．このことは属性の値に応じて動的に関数を変更できることを意味し，そのために script 指令を用いると記述量を削減できる．たとえば，N1.value が「3」， T.text が「+」， N2.value が「4」という値を持っているとすると，

「`@N1.value@@T.text@@N2.value@`」は「`3+4`」という文字列に変換される．これをスクリプト言語によって解釈させて「7」という値を得る．属性 value は integer 型なので，「7」は integer 型に変換されて value の値となる． script 指令がない場合， where の中で演算子の種類をチェックし，それに対応した演算により属性 value の値を求めるようなルールを演算子の種類の数だけ書かなければならない．

文法に決定性がない場合，一度認識されたトークンが取り消されることが考えられる．

その場合実行されてしまったアクションも取り消す必要がある．しかし CMG では文法を決定性のあるものとして書くことができ，解析アルゴリズムは決定性のある文法のみを扱っているのでこのようなアクションの取り消しについては考慮する必要がない．

3.4 図形文へのフィードバック

ビジュアルシステムには図の描き換えをおこなうようなものがある．このようなシステムの記述は CMG ではできない．そこで， CMG でトークンの生成，削除，属性の変更をおこなえるように拡張する．これはアクション中で次のように記述する．

生成 $create\ C(\vec{\theta_c})$

削除 $delete\ id$

属性の変更 $alter\ id\ x_a\ value$

アクション中に $create\ C(\vec{\theta_c})$ と書くと次のようになる．

$$S = S' \cup T \backslash \{T_1(\vec{\theta_1}), \cdots, T_n(\vec{\theta_n})\} \cup C(\vec{\theta_c})$$

アクション中に $delete\ id$ と書くと次のようになる．

$$S = S' \cup T \backslash \{T_1(\vec{\theta_1}), \cdots, T_n(\vec{\theta_n})\} \backslash D(\vec{\theta_d})$$

ここで， $D(\vec{\theta_d})$ は id が id であるようなトークンである．
アクション中に $alter\ id\ x_a\ value$ と書くと次のようになる．

$$S = S' \cup T \backslash \{T_1(\vec{\theta_1}), \cdots, T_n(\vec{\theta_n})\} \backslash A(\vec{\theta_a}) \cup A'(\vec{\theta_a'})$$

ここで， $A(\vec{\theta_a})$ は id が id であるようなトークン， $A'(\vec{\theta_a'})$ は $A(\vec{\theta_a})$ の属性 x_a に $value$ を代入したものである．

第 4 章

システム「恵比寿」

われわれは Spatial Parser Generator を持ったビジュアルシステム「恵比寿」を作成した．Marriott らは [8] の中で Chomsky の分類に対応させて CMG のクラス分けをおこなっている．今回実装をおこなったのは CMG のうち exist を扱うことができるものであり，その分類によると type1（文脈依存）に属する．

本章では恵比寿の機能を（1）図形エディタ，（2）図形による CMG の定義，（3）解析とトークン間の意味的な関係の保存，（4）アクションの実行，（5）デバッグ機能に分けて述べる．

4.1 図形エディタ

システムの画面例を図 4.1に示す．図 4.1の画面の上半分を定義窓と呼び，ビジュアル言語の作製者はここで文法を定義する．下半分を実行窓と呼び，ユーザはここで実際に解析，実行したい図形を入力する．これは通常のテキスト言語ではエディタを用いてプログラミングすることに相当する．定義窓，実行窓における図形の入力では，通常のドローツールにあるような，コピー，削除，整列といった操作をおこなうことができる．扱える図形の種類は楕円，長方形，直線，テキスト文字列，円弧， GIF イメージがあり，これらは線の太さ，色，フォントなどの属性を指定できる．図形についての詳細は付録 A.3に示す．

4.2 図形による CMG の定義

CMG はその他のビジュアル言語の記述方法に比べればわかりやすいとはいえ，二次元以上の情報である「図形」を一次元の情報である「文字」を用いて表すために直感的な理解が困難である．エンドユーザでも簡単にビジュアルシステムを記述できるようにすることは本システムの目標の一つであるので，恵比寿ではまず図形を用いて大まかな文法を与えた後にそれから CMG ベースの属性，制約を生成し， CMG 入力部と呼ばれるウィンドウに書き出す（図 4.2）．そのあとユーザはそれを修正して文法を決定するという方式でビジュアル言語の文法とアクションを記述する．なお図 4.2のウィンドウの各部に書かれてい

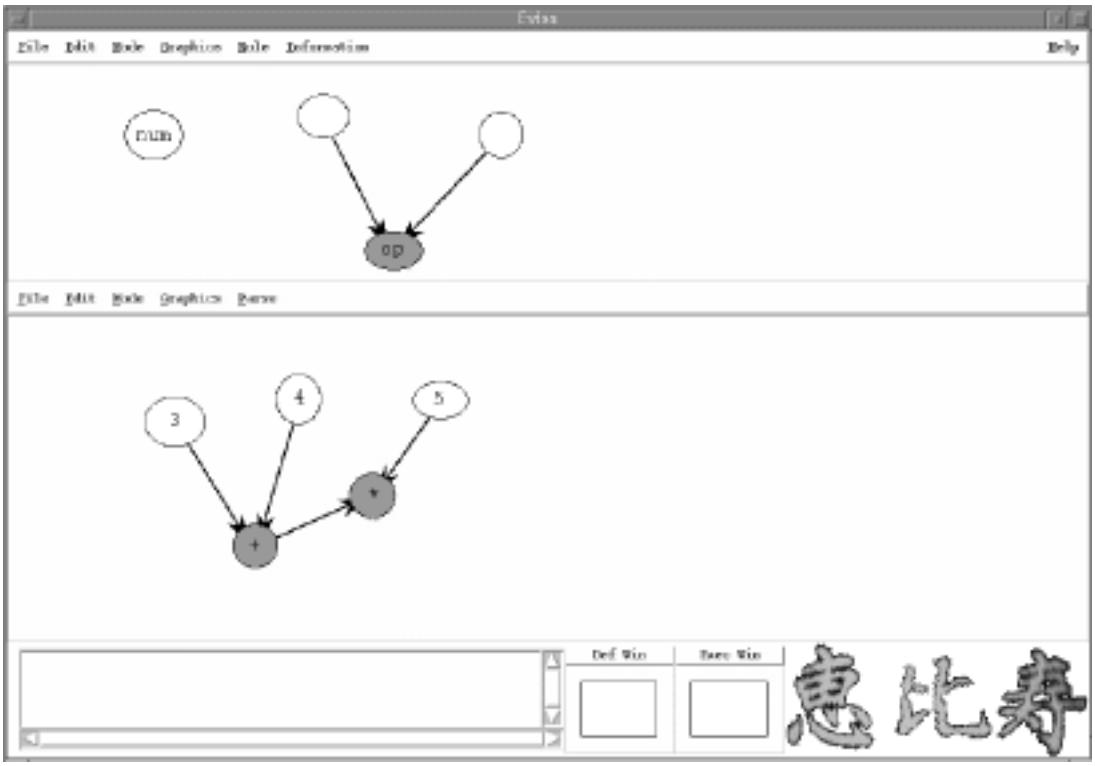


図 4.1: 恵比寿の実行例

る内容は 3.3節の CMG による記述の 15 行目から 32 行目に書いたものとほぼ同様である．CMG 入力部は上から順に名前（name），属性（attributes），アクション（action），構成要素間の制約（constraints），構成要素の数と種類を書く欄にわかれている．構成要素の数と種類を書く欄はさらに左から順に normal， exist， not_exist， all にわかれている．これは 3.1節で述べたように CMG がそれぞれを別々に記述しているために可能なことである．ユーザは一つの図形単語としたい図形を選択して CMG 入力部を開く．するとシステムは図形から構成要素とそれらの属性間に成り立っている制約，構成要素の属性の具体的な値を生成し，それらを CMG 入力部に書き出す．ここで，構成要素の属性の具体的な値は意味を持たないことが多いので，後から削除する必要がないよう書き出すか書き出さないかをユーザが設定できるようにした．ユーザはここに新たに名前，属性，アクションを追加し，システムによって生成された制約のうち必要のないものを削除していく．

CMG 入力部における図形単語の一つの属性は型，属性名，属性値から構成される．属性値は以下の 3 つのうちのいずれかである．

1. 一つの構成要素の属性をコピーする場合 *:v-type.v-id.attr* で， *v-type* は 0， 1， 2， 3 のいずれかでそれぞれ normal， exist， not_exist， all を表す． *v-id* は一つの生成規則の中の *v-type* で表される構成要素に順番に付けられた id である． *attr* は属性名である．

図 4.2: CMG 入力部

2. 複数の構成要素から script 指令によって新たな値を作る場合：`script.`*type* と *script* から成るリスト． *type* は新たに作られた値の型， *script* はスクリプト言語に渡すための文字列．「@*v-type.v-id.attr*@」の形をした部分はその属性値に置換される．

3. 定数 : 型と値から成るリスト

ここで， 1 の場合は構成要素の属性の値が変化したときには新たに作られた属性の値も変化するが， 2 の場合は構成要素から新たに定数を作っているので構成要素の値が変っても属性の値は変らない．

4.3 解析とトークン間の意味的な関係の保存

ビジュアル言語の定義が終わったら実行窓に実際に Spatial Parsing したい図形を入力する．恵比寿では Spatial Parsing のモードは二種類ある．新たな図形の入力があるたびに Spatial Parsing されるモードと，ユーザからの要求があったときのみに Spatial Parsing されるモードである． Spatial Parsing によってトークンが認識されると，そのトークンを認識した生成規則に書かれた制約がトークンの属性間に課せられる．たとえばトークン A がトークン B， C を構成要素としていたとする．トークン B， C の中心が一致しているという制約がトークン A を認識した生成規則中にあれば，その制約がトークン B， C の属性間に課せられ，以降の編集においてはトークン B を移動させるとトークン C もその中心が

トークン B の中心と一致するように移動し，結果としてトークン A という一つの図形単語として移動をおこなうことができる．

4.4 アクションの実行

トークンが認識されたときにはその時点で図形単語の生成規則に書かれたアクションが実行される．アクション中には Tcl/Tk の任意のスクリプトを記述できるので，値の計算や，ウィジェットの生成などもおこなうことができる．ビジュアルプログラミングシステムではプログラムの視覚化表現（図形文）を描き換えることによって実行の視覚化をおこなうことが多い．そのため恵比寿では図形文へのフィードバックをおこなうための手続きも提供している．これらの手続きをアクション中に記述することで実行窓の図形を移動，削除，生成したり，実行窓の図形の属性の値（色や文字列の値，線の太さなど）を変更することができる．実行窓の図形の描き換えはトークンデータベース（3.2節）に反映される．たとえば画面上から円を削除するとその円を表していたトークンはトークンデータベース中から削除され，そのトークンを構成要素としていたトークンも削除される．

4.5 デバッグ機能

定義窓における文法の作成時，実行窓における図形文の作成時のいずれの場合においてもデバッグのための機能が必要である．恵比寿はビジュアルシステムの作成時および作成されたビジュアルシステムを使用する際のデバッグ機能も提供している．それらは以下のものである．

トークンのリストの表示 すでに認識されたトークンのリストを表示する．リストの各行はトークンの id，そのトークンが上位のトークンに構成要素として使用されているかどうか，トークンの種類，そのトークンの構成要素という四つの情報により構成されている．各行の上でマウスボタンをクリックすることでそのトークンの属性を表示し，さらに実行窓にあるそのトークンを選択する．

構文木の表示 図形文を解析していく際にできる構文木を表示する． normal， exist， all として使用される構成要素はそれぞれ青，赤，緑の線によって親と結ばれる．

解析時の情報の表示 解析をおこなっていく際の様々な情報を表示する．情報には解析されたトークンの種類，その属性，解析をおこなうために使用している生成規則の id，解析中の構成要素の組み合わせ，成り立っているかどうかチェック中の制約，削除された図形単語の id などがある．

恵比寿のより詳しい操作方法については付録 Aを参照されたい．

第 5 章

実装

恵比寿の実装には Tcl/Tk と一部 C 言語を用いた．Tcl/Tk はインタプリタ言語であるので，3.3節で仮定したスクリプト言語として Tcl/Tk をそのまま利用できるという利点がある．

5.1 制約解消系

恵比寿ではトークンの属性を変数とし，それらの属性の間に制約を課す．制約解消系には SkyBlue [11] の C 言語による実装に手を加え，これを Tcl から呼び出すインタフェースを作成し，使用した．もともとの実装では整数変数の間の制約二つ（二つの変数の値が等しいという `eq` 制約，三つの変数 a，b，c があった時にそれらの間に $a + b = c$ という関係があることを表す `plus` 制約）が実装されていた．恵比寿では属性の型としてこの他に浮動小数点数，文字列，点（x，y 座標の組）を扱いたいのでこれらの型を実装した．SkyBlue の変数には id（以下，*c-id*），値，型を与えた．また，「左にある」「上にある」といった位置関係を表現するためには数値の大小関係に関する制約が必要であるのでこのような制約を実装した．Tcl から SkyBlue を使うためのインタフェースでは Tcl から SkyBlue の変数や制約を操作できる．変数に対しては作成，削除，値の変更，値を得る，型を得るなどの操作ができる．制約に対しては，作成，削除，制約している変数の *c-id* を得るなどの操作ができる．

5.2 CMG

本節では，主にどのようにして図形間に制約を課すのかについて述べる．本節では変数名など実際の文字列はタイプライタ体で，値が変るものはイタリックで表記する．

恵比寿の実装では図 3.1の *ParseForest* を連想配列 `D`，R を連想配列 `P` としている．`P` のそれぞれの要素は表 5.1のようになっている．ただし，*p-id* は一つの生成規則に付けられた id である．*SCC* 中で同じ階層にある生成規則は *p-id* が若いものから順に適用される．

恵比寿では各トークンの属性の値を SkyBlue の変数とすることで各属性の値が変った

名前	値
P(*p-id*.name)	図形単語の名前
P(*p-id*.attrs)	図形単語の属性のリスト
P(*p-id*.script)	図形単語が解析されたとき実行される action
P(*p-id*.constraints)	構成要素の属性間の制約のリスト
P(*p-id*.negative_constraints)	ネガティブ制約のリスト
P(*p-id*.normal)	図形単語の normal の構成要素のリスト
P(*p-id*.exist)	図形単語の exist の構成要素のリスト
P(*p-id*.not_exist)	図形単語の not_exist の構成要素のリスト
P(*p-id*.all)	図形単語の all の構成要素のリスト

表 5.1: 生成規則の内部表現

ときの制約解消を SkyBlue におこなわせている． *c-id* はトークンのデータベースである D に保存されている． D の添字は，「*f-id.* 属性名」の形をしている．ここで， *f-id* はトークンの id である．たとえば 3 という *f-id* を持つトークンの「mid」という属性は， D(3.mid) によって表される．この属性の値と型は， *c-id* を用いて SkyBlue の変数の値，型を得ることによって知ることができる．

構成要素となるトークン間には次のようにして制約が課せられる． EvalRule 中ではトークンのデータベース D に現在あるトークンを構成要素として生成規則のデータベース P にある制約を満たすかどうかチェックされる．構成要素となるトークン間に SkyBlue によって実際に制約を課すのは P の各々の制約が満たされたときであってはならない．すべての制約が満たされたときにのみ新たなトークンとして認識されるためである．したがって，すべての制約が満たされたときに SkyBlue によって実際にトークン間に課せられるべき制約のリスト C を準備し， P の制約のうち満たされているものから順に C に加えていき，新たに認識されたトークンを D に挿入するときにまとめて C に蓄えられた SkyBlue の制約を生成する．

新たに作られたトークンの属性には次のようにして制約が課せられる． 4節で述べた「一つの構成要素の属性をコピーする場合」，「複数の構成要素から script 指令によって新たな値を作る場合」，「定数」の三種類それぞれについて考えなければならない．まず，一つの構成要素の属性をコピーする場合は，新たな図形単語の属性を表すものを D($f_1.attr_1$)，コピーしたい構成要素のトークンの属性を表すものを D($f_2.attr_2$) とすると， D($f_2.attr_2$) に入っていた *c-id* を D($f_1.attr_1$) に入れることによって実現できる．次に，複数の構成要素から script 指令によって新たな値を作る場合，下に述べる方法で属性名から実際の値を得てからそれをスクリプト言語によって解釈させて計算結果を得る．この結果を値とし，script 指令によって与えられた型を型とする新たな SkyBlue の変数を作成し，その *c-id* を D($f_1.attr_1$) に入れることによって実現できる．定数の場合，与えられた型と値を用いて新

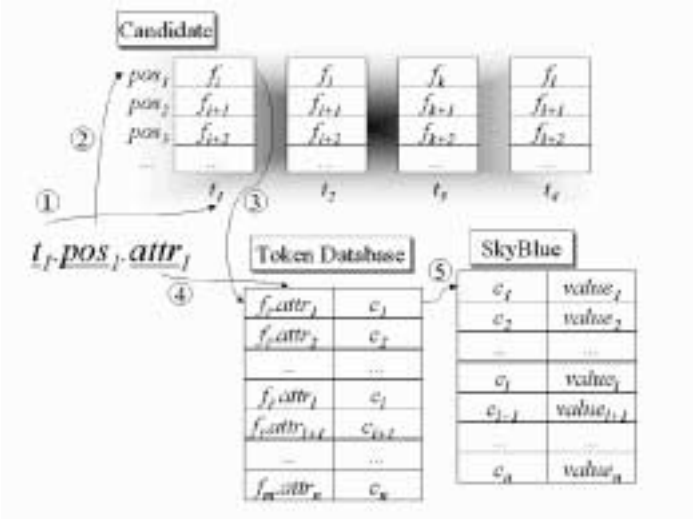


図 5.1: 属性名から値を得る手順

たな SkyBlue の変数を作成し，その *c-id* を D(f_1.$attr_1$) に入れることによって実現できる．

script 指令やアクション中で @*v-type*.*v-id*.*attr*@ のような属性名（*v-type*， *v-id* は 4節と同じもの）から実際の値を得る手順を以下および図 5.1に示す．

1. 一つの生成規則の構成要素となり得るトークンの候補から *v-type*（図 5.1ではその例として t_1 となっている）を用いてその構成要素の種類（normal， exist， not_exist， all）を特定する．

2. それの *v-id*（図 5.1では pos_1）番目の要素を取り出すことにより *f-id*（図 5.1では f_i）を得る．

3. *f-id* がトークンデータベースから検索する際の第一キーとなる．

4. D(*f-id*.*attr*) によって *c-id* を得る．

5. *c-id* を用いて SkyBlue の変数の値を得る．

第 6 章

恵比寿によるビジュアルシステムの作成例

本章では恵比寿を用いてビジュアルシステムを作成する例を挙げる．各節では計算の木を実行するシステム，計算の木を実行し図の描き換えをおこなうシステム， GUI を記述するためのビジュアル言語の処理系 [12]， VISPATCH [13] [14]， HI-VISUAL [15] [16] を恵比寿を用いて作成する．

6.1 計算の木を実行するシステム 1

恵比寿の実行例として， 3.3節に示したことをするアプリケーションを恵比寿を用いて作成する過程を述べる．ただしプログラムを書くユーザが円の中心と文字列の中心を合わせる操作をおこなうのはわずらわしいので，「==」としては位置がほぼ等しいという制約である「`vp_close`」を用いることとする．また， sort が number である `node` の色は白で，sort が operator である `node` の色は濃いグレーで表すことにする．

まず数字を表すノードを定義するために，定義窓に円（circle）を入力し，その中心付近に適当な文字列（text）を入力する．これらをまとめて選択し，メニューバーの Generate から Make New Production Rule を選ぶと CMG 入力部が開く．この時点でシステムによって構成要素の種類の欄には，

```
circle
text
```

と書かれ，構成要素間の制約の欄には，

```
vp_close 0.mid 1.mid
eq 0.radius {integer 13}
...
```

などと書かれる． 1 行目は circle （0）の中心（mid）と text （1）の中心（mid）がほぼ一致しているという制約であり， 2 行目以降は実際に入力された circle， text が持っていた半径，色，書かれている文字列，フォントの種類などの具体的な値である．ここではこれら

の具体的な値は必要ないので削除する．さらに名前を書く欄にnode と書き，属性値を書く欄には，

```
string sort {string number}
point mid 0.mid
integer value 1.text
```

と書く．1 行目は sort の属性値が「number」であることを，2 行目は mid の属性値が circle（0）の中心（mid）であることを，3 行目は value の値が text（1）の文字列（text）と同じであることを表している．アクションを書く欄にはたとえば，

```
puts "@sort@ was parsed."
puts "value = @value@"
```

と書く．これによりこの図形単語が認識されると，@ で囲まれた部分が実際の属性の値に置換された後，Tcl のインタプリタに渡されて実行される．

これで数を表すノードの定義が完了したので CMG 入力部のウィンドウを閉じる．ウィンドウを閉じると定義した図形単語に対応した生成規則の内部表現が作られる．同様にして演算子に二つのノードがつながっているものを定義する．

作成したビジュアル言語で描いたものを実際に実行するときには実行窓に Spatial Parsing したい図形（たとえば図 2.1）を入力していくと，Spatial Parsing のモードが Auto であれば図形を入力するたびに，モードが On demand であればメニューバーの Parse から Parse を選ぶたびに定義されたビジュアル言語の文法に従って実行窓の図形が Spatial Parsing される．仮にすべて入力してから On demand モードで Spatial Parsing したとすると，トークンが認識されるたびにアクションが実行され，

```
number was parsed.
value = 3
number was parsed.
value = 4
number was parsed.
value = 5
operator was parsed.
value = 7
operator was parsed.
value = 35
```

と表示される．一度 Spatial Parsing が成功すると，図形間には実際に制約が課せられる．実行窓のノード（たとえば中に「+」と書かれたノード）の円を移動させようとすると文法にしたがってその中に書かれている「+」という文字列と，それにつながっている矢印とが移動する（図 6.1）．

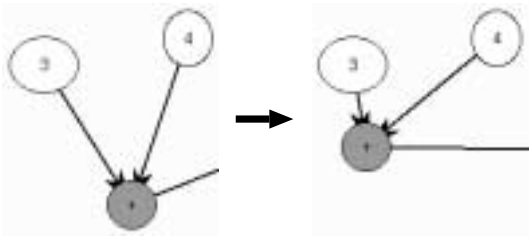


図 6.1: ノードとしての移動

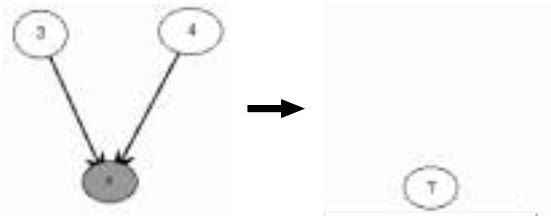


図 6.2: 図の描き換え

6.2 計算の木を実行するシステム 2

二つ目の例として，作成したプログラムを描き換えるようなものを挙げる．規則としては 3.3節に示したものとほとんど同じであるが 33 行目以降を次のように変更したものであるとする．

```
33:       delete {@N1@,@N2@,@L1@,@L2@};
34:       alter @C@ text to_string(@value@);
35:   }
```

これは恵比寿では次のように記述する．

```
delete {@0@ @1@ @2@ @3@}
alter @4@ text @value@
```

手続き `delete` はトークンの id のリストを受け取り，それらを画面から削除し，関連する SkyBlue の変数と制約を削除するものである．手続き `alter` はトークンの id と属性名と新しい値を受け取り，その属性を表す SkyBlue の変数の値を受け取った値に更新するものである．これにより，属性 sort が operator であるような `node` が認識されると，つながっていた二つのノードと，二つの矢印が消え，円の中に書かれていた演算子は計算結果に描き換えられる（図 6.2）．

6.3 GUI を記述するためのビジュアル言語の処理系

視覚的な情報をテキスト言語を用いて記述する場合に完成イメージが想像しにくい．これはビジュアル言語が広く利用されている背景の一つであると思われる． GUI はその名が示すとおりもともと視覚的な情報である．そのため GUI もテキスト言語で記述する場合，その大きさやレイアウトなど，実際に処理系を用いて実行してみないと完成図が想像しにくい． GUI を視覚的に定義するための環境としてはインタフェースビルダがあるが，これらは特定の環境の中で GUI を定義するものであり，システムに依存してしまうためにポータビリティに欠けていた．このことから視覚的な情報を視覚的に，かつ処理系に依存しないように定義できることが望ましいということが言える．本節ではツールキットを用いるような GUI を記述するためのビジュアル言語について述べる． GUI をビジュアル言語によって記述することには次のような利点がある．

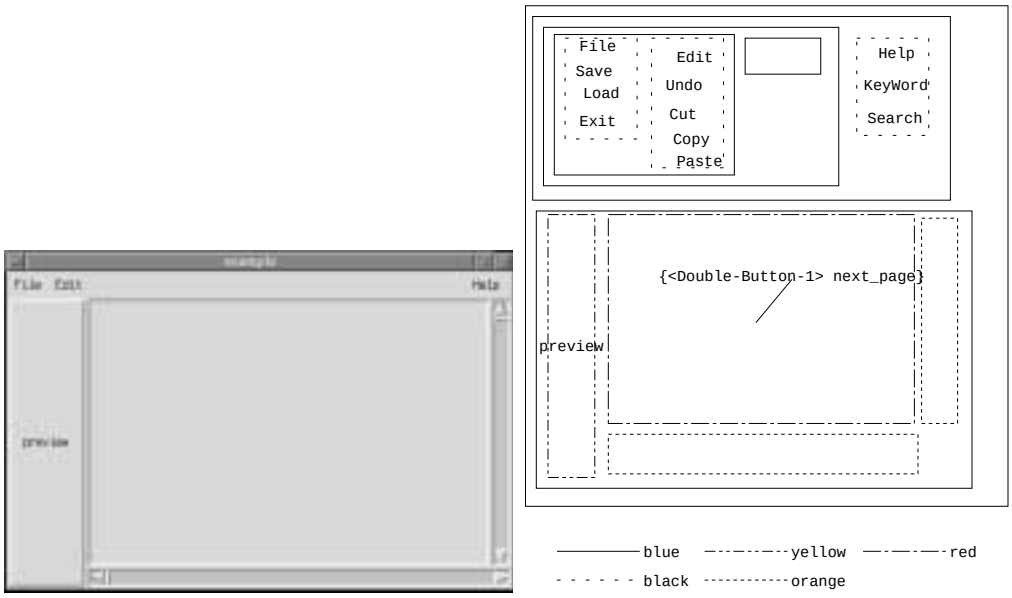
- 完成イメージがつかみやすい
- 「言語」と「処理系」の切り離しによって，
  - 図形（プログラム）の編集に好みのエディタを使用できる．
  - 一つの言語に対して複数の処理系を作ることができる．

6.3.1 GUI を記述するためのビジュアル言語の定義

本節では GUI を記述するためのビジュアル言語を定義する．定義は自然言語と CMG 風の表記を用いてボトムアップにおこなう．

GUI の構成要素として以下のようなものを想定する．ウィジェットとして，フレーム，スクロールバー，ボタン，テキスト，メニューを持つ．フレームウィジェットは，ウィジェットを内部に持つことによってウィジェットの階層構造を作る．スクロールバーウィジェットは特定のテキストウィジェットの表示範囲をスクロールさせる．ボタンウィジェットは特定の手続きを呼び出すボタンである．テキストウィジェットは内部にテキストを表示する．メニューウィジェットはメニューである．メニューはその子供としてボタンを持つことができる．これらのウィジェットにはバインディングをすることができる．バインディングは特定のイベントが発生したときにそのイベントに関連付けられた手続きを呼び出す機構である．

次に各々のウィジェットやバインディングなど，ビジュアル言語の最も表層に現れるものを定義する．フレームウィジェット（`Frame`），スクロールバーウィジェット（`Scroll`），およびテキストウィジェット本体（`_TextW`）はそれぞれ青色，オレンジ色，赤色の長方形によって表す．ボタンウィジェット（`Button`）は黄色の長方形の中心に文字列が描かれたものによって表す．メニューウィジェット（`Menu`）は黒色の長方形の中にメニューの名前を表す文字列，メニューの内容を表す文字列を縦に並べたものによって表す．また，ウィジェットの中心と文字列とを直線で結んだものによってバインディングを表す．バインディ



(a) (b)

表 6.1: GUI の例とそれを表すビジュアルプログラム

ングに関しては後にもう少しきちんと定義する．各々のウィジェットに色を付けることには，(1) 記述したときの理解を容易にする，(2) ビジュアル言語自体の定義を容易にする，という二つの意味がある．

次に定義したビジュアル言語を用いて実際に GUI を記述する例を示す．作成したい GUI の例とそれを表すプログラムを図 6.1に示す．図 6.1(a) で，メニューの中にはいくつかのものがあり，テキストウィジェットでマウスの左ボタンをダブルクリックすると`next_page`という手続きを呼び出すものとする．この GUI を記述するためのビジュアルプログラムは図 6.1(b) のようになる．ただし白黒なので色は図 6.1(b) のいちばん下に示したような線の種類によってあらわす．

メニューウィジェットを表す図は，内部の文字列などの情報を記述する必要があるために実際に生成されるメニューウィジェットの見た目とは異なる．また，バインディングも実際には画面上に現れるものではないが，ビジュアル言語の上では図の中に現れてきている．このことは GUI が視覚的な情報だけから構成されるわけではないことに起因している．

次に定義したものを組み合わせて最終的に一つの GUI として解析するための規則を定義する．テキストウィジェット本体にはスクロールバーがつく場合があるので，これを含めてテキストウィジェット（`TextW`）として定義する．テキストウィジェットはスクロールバーなし（`text`），垂直スクロールバー付き（`texty`），水平スクロールバー付き（`textx`），垂直水平両方のスクロールバー付き（`textboth`）の 4 種類に分けて定義する．`type` が`text`,

textboth であるTextW を CMG 風に表すと以下のようになる.

```
TextW ::=  T:_TextW {
  type = "text";
}

TextW ::= T:_TextW, S1:Scroll, S2:Scroll where (
  left_of(T,S1) &&
  above(T,S2)
) {
  type = "textboth";
}
```

ウィジェット（_WIDGET）はフレームウィジェット，ボタンウィジェット，テキストウィジェット，メニューウィジェットのいずれかとしてあらわされる．スクロールバーは単体では意味を成さないのでWIDGET には含めない．type がframe であるような_WIDGET を CMG 風に表すと以下のようになる.

```
_WIDGET ::= F:Frame {
    type = "frame";
}
```

ウィジェットにはバインディングがある場合がある．_WIDGET の中心からバインディングをあらわす文字列の中心までを線で結んだものを新たにWIDGET として定義する．バインディングがある場合のWIDGET を CMG 風に表すと以下のようになる.

```
WIDGET ::= W:_WIDGET, T:text, L:Line where (
  L.start == W.mid &&
  L.end == T.mid
)
```

GUI はWIDGET を上下または左右に組み合わせて構成される． CMG 風に表すと，次の三つの再帰的な規則から成る.

```
GUI ::=  W:WIDGET

GUI ::= G1:GUI, G2:GUI, F:Frame where (
  left_of(G1,G2) &&
  cover(F,G1) &&
  cover(F,G2)
)
```

```
GUI ::= G1:GUI, G2:GUI, F:Frame where (
  above(G1,G2) &&
  cover(F,G1) &&
  cover(F,G2)
)
```

6.3.2 GUI を記述するためのビジュアル言語の処理系の実装

GUI を記述するためのビジュアル言語の処理系を恵比寿上に実装した．この処理系は 12 個のルールから構成され，まず構造を解析して内部表現を生成し，その内部表現を用いて実際に GUI を生成する．内部表現から GUI を生成するための補助コードは Tcl/Tk で 250 行程度である．詳しくは付録 A.5.2で述べる．

6.3.3 本節のまとめ

本節では GUI を記述するためのビジュアル言語の定義と，定義したビジュアル言語によって実際に GUI を記述する例を示した．議論を単純化するために GUI の構成要素であるウィジェットの種類を 5 種類のみとしたがさらに種類が増えたとしてもビジュアル言語の定義は本節で述べたのとほぼ同様の手法によって拡張することができる．本節で述べたビジュアル言語以外のデザインでも同じ GUI を定義することはできる．たとえばビジュアル言語の三次元化（外観に二次元を使い，奥行き方向をバインディング，メニューの中身に使う）によって，よりビジュアル言語の見た目そのままが得られる GUI となるようにデザインするのも面白いだろう．

6.4 VISPATCH

6.4.1 VISPATCH の概要

VISPATCH[13] は図形的なルールをもとに図形を描き換えるビジュアル言語である．描き換えはユーザやシステムによって発生されるイベント（マウスボタンを押す，離すなど）によって開始される． VISPATCH によるプログラムの例を図 6.3に示す． VISPATCH のプログラムは大きくイベントセンサ，ディスパッチ図形，ルール領域の三つの部分に分けることができる．図 6.3において，中央の右上に向かっている二重の矢印がディスパッチ図形であり，その左側および右側の影のついた長方形がそれぞれイベントセンサ，ルール領域である．ルール領域の中には複数のルールが記述されている．長方形が二つ横に並んだものがルールである．ルールのうち左側の長方形がヘッドであり，右側の長方形がボディである．ルールの中で細い矢印はドラッグイベントを表わしている．イベントセンサにおいてイベントが発生すると，ディスパッチ図形が指しているルール領域の中にヘッドに描かれた図形によって表わされる条件が成立しているルールがあるかどうかを探す．条件が成立しているものがあればその図形をボディに描かれた図形に描き換える．図 6.3においてはイベント

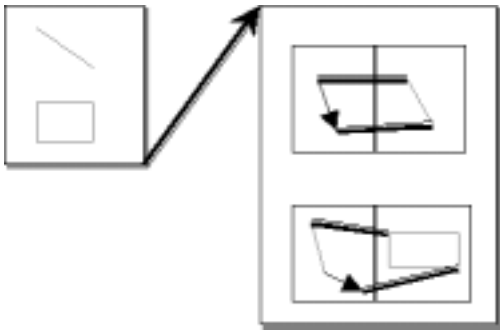


図 6.3: VISPATCH のプログラム

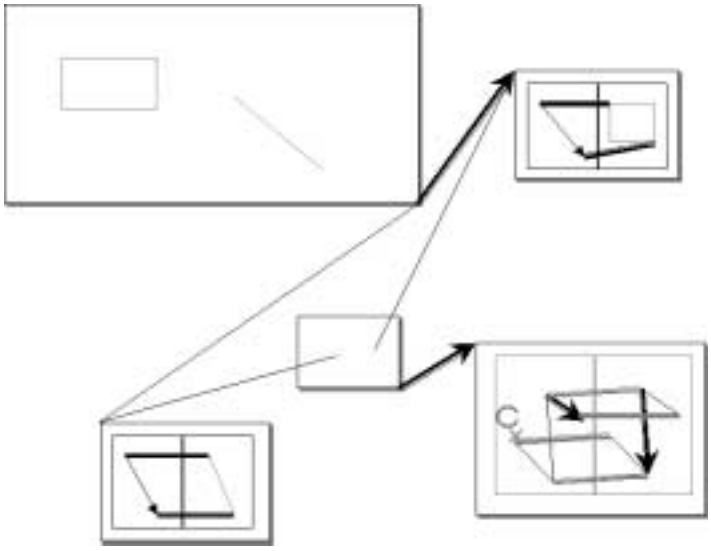


図 6.4: VISPATCH におけるルールの変更

センサで何もないところでドラッグイベントを発生させると，上のルールによってドラッグイベントは直線になる．この際，ドラッグイベントの始点が直線の始点に，終点が終点になる．これはルール中のヘッドとボディの間を結んでいる線（Equivalence Point）によって表わされている．また，直線の端点でドラッグイベントを発生させると，直線とドラッグイベントは直線の別の端点とドラッグイベントの終点を対角線とする長方形になる．

図 6.4では右下のルールの中でクリックイベントを用いている．クリックイベントは C という文字の下に×印が小さく描かれたものによって表わされる．このルールによって中央の小さなイベントセンサの中にある直線の端点でクリックイベントが発生すると左上の大きなイベントセンサから出ているディスパッチ図形の向きが変更される．これはモード変更であり，左上のイベントセンサでは同じドラッグイベントでも長方形と直線を描き分けることができる．

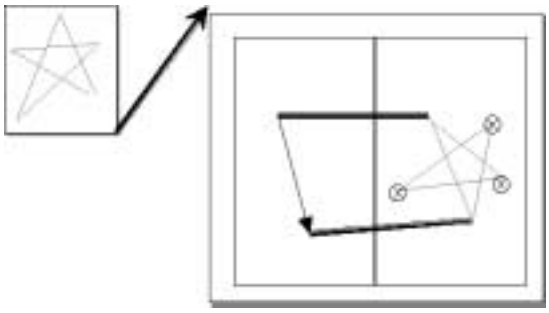


図 6.5: VISPATCH における点の生成

これまでの例ではボディの図形を指定するための点はヘッドの図形によって与えられていたが，図 6.5ではポイントジェネレータを用いて新たな点を生成することによって星型を描いている．ポイントジェネレータは基準となる図形との関係から新たな点を生成する．この関係は複数考えられる．たとえば図 6.5では円の中に×印が描かれたポイントジェネレータによって，基準となる直線（これはヘッドのドラッグイベントによって決められる）に対しての角度と，その直線との長さの比を指定して新たな点を生成する．

6.4.2 恵比寿による VISPATCH の実装

本節では恵比寿による VISPATCH の実装について述べる．図 6.6は恵比寿上に実装した VISPATCH でイベントセンサにおいてドラッグイベントが発生したときにそこから二分木を描くようなプログラムを実行したところである．

実装の対象となる機能は以下のものとした．

扱う図形 直線，長方形，ディスパッチ図形（ディスパッチ図形は二重矢印ではなく，太い矢印とする）

扱うイベント ドラッグイベント，クリックイベント（クリックイベントは C の下に×印ではなく， C だけとする）

ポイントジェネレータ 図 6.5で述べた回転および拡大縮小をおこなうもの（円の中に×印が書かれたものではなく，ただの円とする）

また，イベントセンサおよびルール領域は影付きの長方形ではなく，ただの長方形とした．

6.4.1節で述べたように， VISPATCH はイベントによって描き換えが開始される，つまり Event Driven である．恵比寿では図形の追加，削除，変更によって解析が開始される，つまり Data Driven であり，イベントの発生によって解析を開始するのではないので，イベントセンサにおいてイベントが発生したらルールのヘッドにあるのと同じ図形を生成し，それによって描き換えを開始する方式とした．

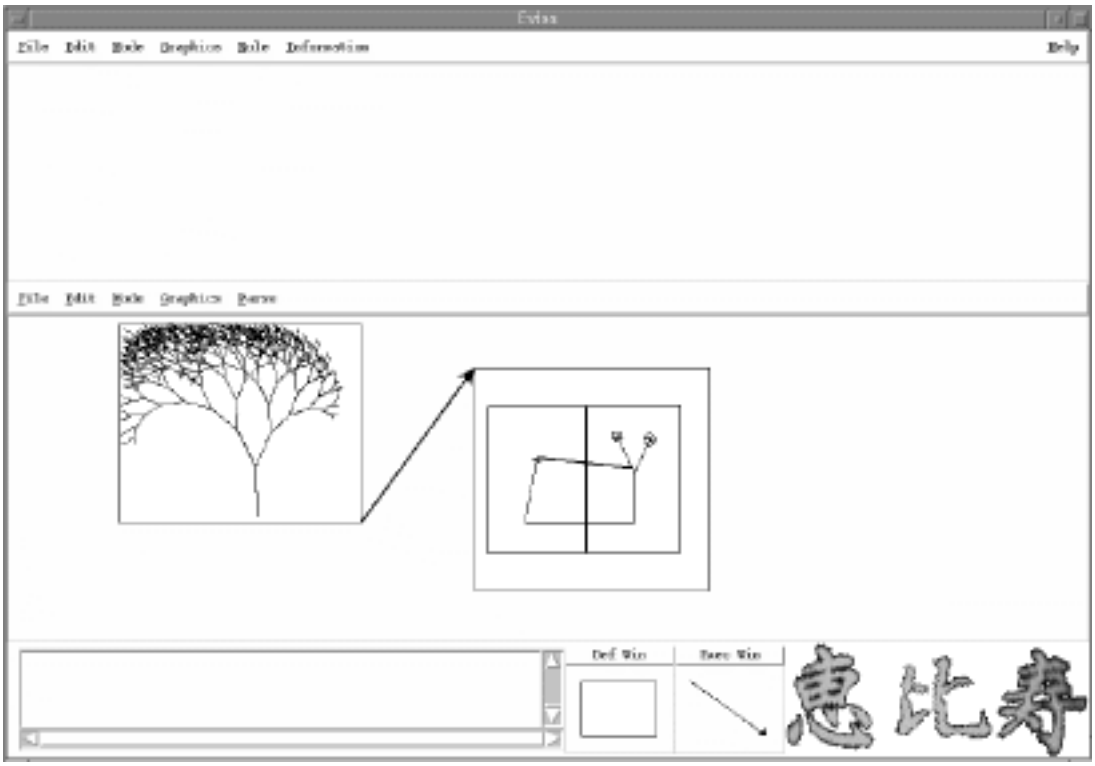


図 6.6: 恵比寿上に実装した VISPATCH

VISPATCH を定義するための恵比寿の生成規則は全部で 24 個ある．ヘッド（Head）は一つの長方形である．ヘッドのすぐ右側にはボディとなるべき長方形がなくてはならない．また，ヘッドおよびボディとなるべき長方形はルール領域となるべき長方形に囲まれていなければならない．ヘッドの中の直線，長方形，ディスパッチ図形はそれぞれ LineH, RectH, DispatchH として定義される．ヘッドの中のドラッグイベント，クリックイベントはそれぞれ DragH, ClickH として定義される．ボディ（Body）は一つの長方形である．ボディすぐ左にはヘッドがなければならない．ボディの中の直線，長方形，ディスパッチ図形はそれぞれ LineB, RectB, DispatchB として定義される．ボディの中のドラッグイベントは DragB として定義される．ポイントジェネレータ（PointGen）はボディの中の円として定義される．ルール（Rule）は隣同士のヘッドとボディとして定義される．ヘッドとボディの図形の点の対応を示す直線（EqPoint）はヘッドに始点を持ち，ボディに終点を持つ直線として定義される．ルール領域の枠（RuleFrame）は左上の点にディスパッチ図形の終点を持つ長方形として定義される．ルール領域（RuleArea）はルール領域の枠およびその中のすべてのルールとして定義される．ディスパッチ図形（Dispatch）を定義するためにまず折れ線矢印（Arrow）を定義する．折れ線矢印は再帰的に定義される．再帰の終了条件となる生成規則は「ただの矢印は折れ線矢印である」というものである．もう一つの生成規則は「折れ線矢印はただの矢印の終点に折れ線矢印がある」というものである．ディス

パッチ図形は折れ線矢印の始点と終点に長方形があるものとして定義される．イベントセンサ（EventSensor）は右下にディスパッチ図形の始点がある始点を持つ長方形として定義される．ルールの中の図形およびイベントセンサすべてを集めるルールが AllRule である．ドラッグイベント（Drag）は終点と始点がイベントセンサの中にある矢印として定義される．描き換えの結果として生成される，始点のみがイベントセンサの中にある矢印は Garbage という生成規則により削除される．ドラッグイベントのうち，マウスの移動距離が小さいものがクリックイベントである．したがって，十分短いドラッグイベントをクリックイベントに描き換える生成規則 DragToClick を定義した．

AllRule の中ではアクションとして VISPATCH の各々のルールから恵比寿の生成規則を生成する手続きを呼び出す．一つの VISPATCH のルールに対して一つの恵比寿の生成規則が生成される．このための補助コードは Tcl/Tk で 500 行弱である．ルール領域は自分がどのイベントセンサとつながっているかを知っているので，生成される恵比寿の生成規則では自分がつながっているイベントセンサの中でイベントが発生したときのみ制約が成り立つように記述されている．したがって二つのイベントセンサ A，B があるとして A でイベントが発生しても B では描き換えは起こらない．ヘッドに描かれた図形が恵比寿の生成規則では構成要素となる．また，ボディに描かれた図形は恵比寿の生成規則ではアクション中で新たに生成される図形単語となる．

ルールのヘッドはすべてドラッグイベントを表す矢印か，クリックイベントを表す文字「C」を含んでいる．したがってそのルールから生成される恵比寿の生成規則もそれらを必ず含むことになる．CMG の解析アルゴリズムでは決定性のあるものしか解析することはできない．たとえば図 6.3のような二つのルールのヘッドはどちらも矢印を含んでいる．仮にイベントセンサ中で直線と端点を共有する矢印（ドラッグイベント）があったとしても，上のルールに対応する恵比寿の生成規則が適用されてドラッグイベントは直線に描き換えられる場合もあるし，意図した通りに下のルールに対応する恵比寿の生成規則が適用されて直線とドラッグイベントが長方形に描き換えられる場合もある．この非決定性をなくすためには一般にはネガティブ制約を用いなければならない．しかしながら，生成される恵比寿の生成規則はすべて直線（矢印を含む）と文字列，すなわち図形文字だけから構成されるので 3.1における SCC ではもっとも低い階層に位置する．5.2節で述べたように恵比寿では同じ階層にある生成規則は生成規則の id が若いものから順に適用されるので，ヘッドにある図形の数が多いものほど若い id を付けることによって，意図した通りに解析をおこなうことができる．ここで，ヘッドにある図形の数が同じ場合は，id が付けられる順番が不定となるが，それぞれのヘッドは生成規則の制約によって区別されるので生成規則が適用される順番によって解析結果が異なることはない．

定義された生成規則によってディスパッチ図形の向きが描き換えられるなど，図形の変更が起こると恵比寿の解析が開始され，生成規則の再定義がおこなわれる．たとえば図 6.4に示したような VISPATCH プログラムを恵比寿上に作成した場合について考えてみる．中央のイベントセンサに端点を持つ二つの直線のうち，左側の直線の端点においてクリックイ

ベントを発生させたた場合，右下のルールに対応する生成規則によってDispatchがいったん削除され，その後向きを変えて太い矢印が描かれる．Dispatchが削除された時点でそれを構成要素としていたEventSensorが削除され，それにともないEventSensorを用いていたAllRuleが削除される．太い矢印はArrow，Dispatchと認識が進み，Dispatchができた時点でEventSensor，AllRuleが再構成される．このようにして生成規則の再定義が実装されている．

もともとVISPATCHはルール（つまりVISPATCHプログラム）もデータ（VISPATCHによる描き換えの対象）として扱うことができたが，恵比寿上に実装することによってVISPATCHの処理系そのものとVISPATCHプログラムとを恵比寿の生成規則という同じレベルで記述したことになる．

6.5 HI-VISUAL

HI-VISUAL[15] [16]は二つ以上のアイコンの重ねあわせによって動作を指定するビジュアル言語である．紙アイコンの上に鉛筆アイコンを重ねるとエディタが起動し，紙アイコンをコピー機アイコンの上に重ねるとファイルをコピーする，という具合である．この関係は一意に定まるとは限らない．たとえば紙と鉛筆の組み合わせとしては紙で鉛筆を包むというものも考えられる．アイコンはすべて物体（はさみ，鉛筆など）を表わし，動作（切る，書くなど）を表わすものはない。アイコンはオブジェクトであり，アイコンの重ねあわせはオブジェクト間のメッセージの送信であると見ることができる．

HI-VISUALの最も中心的な概念である二つのアイコンの重ね合わせによる動作の指定を恵比寿上に実装した（図6.7）．一つのアイコンは一つのGIFイメージを構成要素として持ち，各アイコンは一つの生成規則として表される．アイコンのうち，同種のアイコンでもそれぞれを区別する必要があるもの（ファイル，ディレクトリなど）には属性として名前が付けられている．一つの動作もまた一つの生成規則として表される．動作を表す生成規則は構成要素として二つのアイコンを持つ．二つのアイコンが重なっているときにその生成規則が適用され，アクションとして指定した動作が実行される．動作を表す生成規則には重なりの順番（どのアイコンが画面の手前に描かれているか）が考慮される．これによってたとえば「紙で鉛筆を包む」というのと「鉛筆で紙に書く」というのとを区別する．アイコンはファイル，バインダ，手，ペン，ゴミ箱，ドア，引き出し，はさみを実装した．これらを重ね合わせたときの動作としては表6.5に示すものを実装した．

下のアイコン	上のアイコン	動作
ファイル	ペン	エディタを開いてファイルを編集
バインダ	ファイル	ファイルをバインダにしまう
バインダ	手	バインダの中にあるファイルを見せる
ゴミ箱	ファイル	ファイルを捨てる
ゴミ箱	バインダ	バインダを捨てる

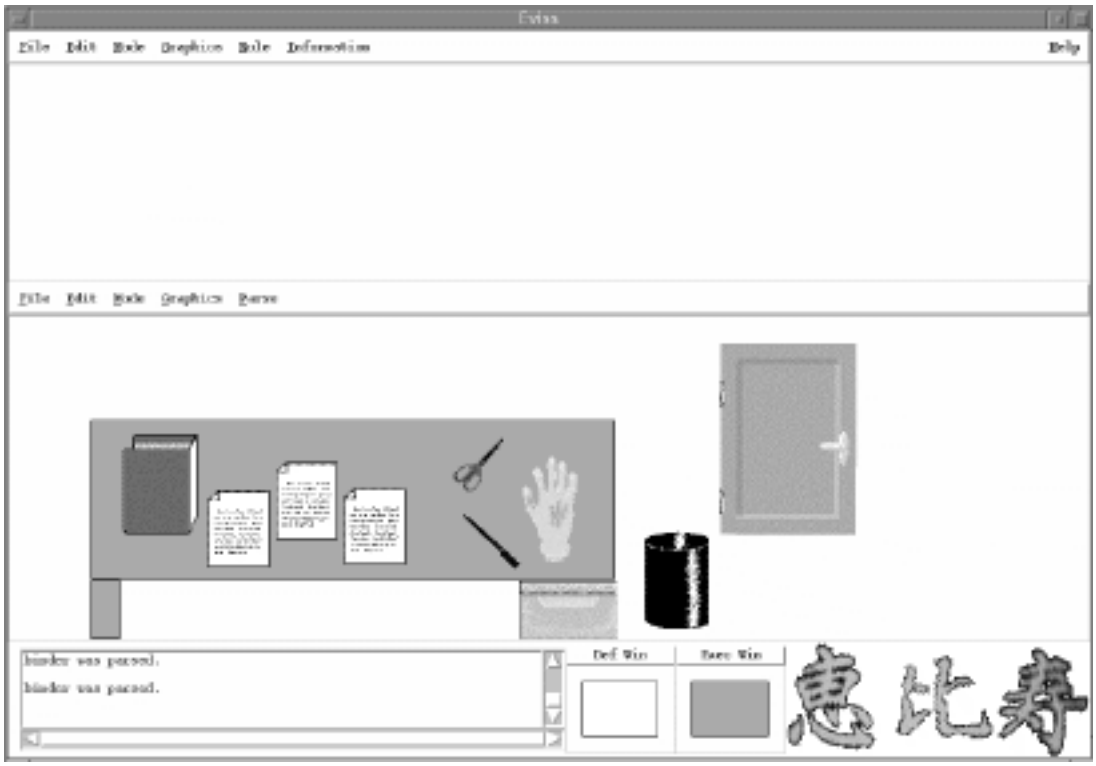


図 6.7: 恵比寿上に実装した HI-VISUAL

第 7 章

議論

本章では恵比寿とオブジェクト指向プログラミング，論理プログラミングとの関係について考察する．

7.1 恵比寿とオブジェクト指向プログラミングとの対応

恵比寿の生成規則はクラスに対応するといえる．このとき図形単語の属性をクラスの属性とみなすことができる．一つの生成規則が一つのクラスの定義になるとは限らない．たとえば GUI を作成するためのビジュアル言語の処理系の実装においてはクラス GUI を定義するために三つの生成規則を用いていた． HI-VISUAL の実装においては生成規則はファイル，ペンなど，どのようなクラスがあるかという定義をおこなうものと，それらが重ねられたときにどのようなメソッドが呼び出されるかという定義をおこなうものとがあった．

GUI の例での図形単語_WIDGET は構成要素がFrame のもの，Button のもの，TextW のもの，Menu のものの四種類があった．この四つの生成規則によってクラス Frame， Button, TextW， Menu のスーパークラス_WIDGET を定義したと見ることができる．一方，図形単語を定義する際に構成要素の属性の値に制約を用いて制限を設けることによって，構成要素の図形単語を親とするサブクラスを定義したことになる．たとえば VISPATCH の例におけるLineB，LineH は直線を継承してそれぞれボディにある，ヘッドにあるという制限を受けたものである．

構成要素をまとめたものが定義する図形単語であるので，これを集約とみなすことができる．計算の木の例において数字の node は楕円とテキスト文字列から構成されていた．再帰的な生成規則は再帰的な集約であるといえる．

exist の構成要素として使用した図形単語の id を属性として持つことによりその図形単語を参照することが可能となる．これは関連であるとみなすことができる． VISPATCH の例における EventSensor では exist の Dispatch を通してそれとつながれている RuleArea の id を知っていた．この id を用いることで RuleArea の属性を参照していた．

トークンをオブジェクトとみなすことができる．恵比寿では図形文字のトークンしか直

接生成することができないので，オブジェクトはもっとも基本的なものを組み合わせてより複雑なオブジェクトを構成していくという形で生成する．興味深いのは VISPATCH の例における `AllRule` という図形単語である．この図形単語のインスタンスは生成規則を生成するトークンである．これはクラスを生成するオブジェクトであるといえる．

7.2 恵比寿と論理プログラミングとの対応

恵比寿の一つの生成規則は論理プログラムにおける一つのルールに対応する．図形単語の名前が述語名に対応する．

解析をおこなうことが問い合わせに対応する．

構成要素が論理プログラミングにおける変数であるといえる．図形単語間の属性に制約を与えることによってユニフィケーションを実現していると見ることができる．

実装面ではトークンデータベースがゴールキューである．ゴールのキューの中にルールによって書き換えられるものがあったらリダクションをおこなうということとトークンデータベースの中で生成規則が適用できるものがあったら図形単語を認識するということが対応している．

第 8 章

関連研究

8.1 ビジュアルシステム生成系

これまでビジュアル言語の文法を記述する方法とそれに基づいて Spatial Parsing をおこなうアルゴリズムはいくつか提案されているが，実装となるとあまり多くはない [17]. SPARGEN[18] は PLG の拡張である OOPLG に基づいている． SPARGEN では解析結果を利用することができるが，解析後にプログラムである図形の間に意味的な関係を保存するような機能はない． [10] に述べられているシステムは CMG に基づいている．このシステムでは図形間に意味的な関係を保存するような機能はあるが，解析結果を利用することができない.

恵比寿は図形を用いたインタラクションを主眼としているために図を解析してテキストの生成をおこなうことはできるが，逆にテキストから図を生成するのは困難と思われる. TRIP2[19] では 2 つの中間表現を準備することによってテキストから図，図からテキストへの相互変換を可能としている．しかし， TRIP2 でおこなえるのはルールに従ってテキスト，図を生成することのみであり，恵比寿のように action を実行することによって任意のスクリプトを実行できるわけではない．また， TRIP2 ではテキストから図を生成するときに制約に基づいて位置を決定しているが，そのあと図形間に制約が課せられるわけではないので，恵比寿のように図形単語単位での移動や変形はできない.

恵比寿がすべてを円などの基本的な図形文字を描いて Spatial Parsing していき，図形間に制約を与えることで図形単語を生成するのに対し， ThingLab[20] ではオブジェクト指向の概念に基づいて図形単語を構築し，図形文字 (基本図形）だけでなくすでに定義した図形単語（ThingLab ではオブジェクト）を直接生成することができる．この方法は Spatial Parsing の効率化や図形入力の効率化の面において恵比寿でも参考にしたいと考えている．しかしながらこの方法では文法に基づいて図形間に制約を動的に追加するわけではないので他のツール，もしくはフリーハンドで描いた図を解析して制約を与えるということはできない．今後この二つの方法のバランスをどこで取るかということが問題となると考えられる.

実行可能制約を用いた図形エディタ [21] では実行可能制約を用いることによって編集が

なされたときには提供されている任意の操作が実行されるように指定できる．このことで図形単語ごとの操作をおこなうことができるが，恵比寿のようにビジュアル言語の意味を解析することはできない．

8.2 生成規則の視覚化と図形文へのフィードバック

[22] では図の要素を視覚的に表した変数を Prolog 風の言語で使用することによってビジュアル言語の定義を試みている．視覚的に表したものは実際に表現したいもの以上の情報を持ってしまう．たとえば二つの円が交わっていることを示したいとしてそのような図を描いたとすると，その図は交わっているということ以外に線の太さ，色，円の大きさや直線の長さ，実際の座標などの必要のない情報まで持ってしまう．トポロジカルな関係のみに注目するのであればこれで問題ないが，恵比寿ではもっと一般な幾何的な位置関係や，大きさ，色なども考える必要がある．したがってこの方法は恵比寿には適用できない．

図の描き換えをおこなうシステムとしては ChemTrains[23]， BITPICT[24]， Visulan[25]，VISPATCH[13]（6.4.1節）などがある．これらは視覚的にルールを定義している．このような描き換えをするシステムのルールには階層がない．恵比寿における生成規則は階層的であるという点でこれらのシステムと異なる．

ChemTrains は図形の接続関係と包含関係，というトポロジカルな関係のみに注目しているために図のみで一意にルールを定義できている． ChemTrains でも VISPATCH と同様にルールはヘッド（pattern picture）とボディ（result picture）から成る図形として定義される．ルールのヘッドとボディの図形の対応は図形のトポロジカルな関係をマッチさせることによって指定されるという点においては Equivalence Point によって明示的に指定する VISPATCH より優れている．

Visulan や BITPICT でのルールは ChemTrains や VISPATCH 同様ヘッドとボディから成る．ヘッド及びボディは $n \times n$ 個の二次元の点のならび（ビットマップ）である（Visulan には三次元版も存在する）．図形文とヘッドとのマッチングをおこない，図形文の中でマッチしたビットマップをボディのビットマップに描き換える． Visulan や BITPICT においては幾何的な位置関係のみに注目している． Visulan や BITPICT は点のみを図形文字とするビジュアル言語であるといえる．

VISPATCH は図形間の関係として接続関係のみに注目しているために図形間の関係を図を用いて一意に定義できる．さらにルールの描き換えも同じプログラミングの枠組みの中でおこなえる点において上に述べた三つのシステムよりも優れている．ただし Equivalence Point を使ってルールのヘッドとボディのどの点が対応するのかを明示的に示す必要があるのでルールの作成は手間がかかる．また Equivalence Point によって対応を指定することは Equivalence Point がない点の生成のために Point Generator を必要とすることになり，VISPATCH プログラムの可読性の低下を招いている．

第 9 章

おわりに

本研究ではビジュアル言語の文法の記述方法の一つである CMG にスクリプト言語を用いてアクションの概念を導入し，アクション付きの CMG を用いてビジュアル言語の文法と動作を与えると定義されたビジュアル言語の処理系となるようなシステム「恵比寿」を作成した．本論文では CMG へのアクションの導入と恵比寿について述べ，恵比寿上で異なるタイプのアプリケーションを作成する例を示した．

今後は図形の入力法の効率化をおこないたいと考えている．それには ThingLab のように図形単語を直接入力することが有効であると考えられる．また，ルールを完全に図形のみで入力できるようにすることも今後の課題の一つである．

謝辞

本研究を進めるにあたり終始ご指導下さった主査の田中二郎助教授および副査の西原清一教授，細野千春助教授に心から感謝いたします．寺茂夫さんには恵比寿を使用することでデバッグに協力していただきました．田中研究室のみなさんからは恵比寿のユーザインタフェースに関して多くの貴重な助言をいただきました．東京工業大学の松岡聡先生，志築文太郎さんからは研究の初期の段階で研究の方向性について助言をいただきました．また，NTT 基礎研究所の原田康徳さんは「一枚の紙」というコンセプトに関する議論に付き合ってくださいました．ここに感謝の意を表します．

参考文献

[1] Henry F. Korth and Abraham Silberschatz. *DATABASE SYSTEM CONCEPTS*. McGraw-Hill Book Company, 1986.

[2] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall International, 1991.

[3] 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, 1998. To appear.

[4] Eric J. Golin. Parsing Visual Languages with Picture Layout Grammars. *Journal of Visual Languages and Computing*, No. 2, pp. 371–393, 1991.

[5] G. Costagliola, S. Orefice, G. Polese, G. Tortora, and M. Tucci. Automatic Parser Generation for Pictorial Languages. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, pp. 306–313, 1993.

[6] F. Ferrucci, G. Tortora, M. Tucci, and G. Vitiello. A Predictive Parser for Visual Languages Specified by Relation Grammars. In *Proceedings of the 1994 IEEE Symposium on Visual Languages*, pp. 245–252, 1994.

[7] Kim Marriott. Constraint Multiset Grammars. In *Proceedings of the 1994 IEEE Symposium on Visual Languages*, pp. 118–125, 1994.

[8] Kim Marriott and Bernd Meyer. Towards a Hierarchy of Visual Languages. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, pp. 196–203, 1996.

[9] Kent Wittenburg, Louis Weitzman, and Jim Talley. Unification-based Grammars and Tabular Parsing for Graphical Languages. *Journal of Visual Languages and Computing*, No. 2, pp. 347–370, 1991.

[10] Sitt Sen Chok and Kim Marriott. Automatic Construction of User Interfaces from Constraint Multiset Grammars. In *Proceedings of the 1995 IEEE Workshop on Visual Languages*, pp. 242–249, 1995.

[11] Michael Sannella. Constraint Satisfaction and Debugging for Interactive User Intefaces. Technical report, University of Wasington, 1994. available from `http://www.cs.washington.edu/research/constraints/sannella-phd.html` C code is available from `ftp://ftp.cs.washington.edu/pub/constraints/code/SkyBlue/c/`

[12] 馬場昭宏, 田中二郎. GUI を記述するためのビジュアル言語. インタラクティブシステムとソフトウェア V, 日本ソフトウェア科学会 WISS'97, pp. 135–140. 近代科学社, 1997.

[13] 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.

[14] 原田康徳, 宮本健司. グラフ書き換え言語とそのインタラクティブシステムへの応用. インタラクティブシステムとソフトウェア V, 日本ソフトウェア科学会 WISS'97, pp. 199–204. 近代科学社, 1997.

[15] 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*, pp. 34–41, 1986.

[16] 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*, pp. 254–259, 1991.

[17] Volker Haarslev and Michael Wessel. GenEd – An Editor with Generic Semantics for Formal Reasoning about Visual Notations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, pp. 204–211, 1996. available from `http://kogs-www.informatik.uni-hamburg.de/~haarslev/publications/`

[18] Eric J. Golin and Tom Magliery. A Compiler Generator for Visual Languages. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, pp. 314–321, 1993.

[19] Satoshi Matsuoka, Shin Takahashi, Tomihisa Kamada, and Akinori Yonezawa. A General Framework for Bidirectional Translation between Abstract and Pictorial Data. *ACM Transactions on Information Systems*, Vol. 10, No. 4, pp. 408–437, 1992.

[20] Alan Borning. The Programming Language Aspects of Thinglab, a Constraint-Oriented Simulation Laboratory. *ACM Transaction on Programming Languages and Systems*, Vol. 3, No. 4, pp. 353–387, 1981.

[21] 服部隆志. 編集操作におけるマクロと制約の統合. 田中二郎（編）, インタラクティブシステムとソフトウェア IV, 日本ソフトウェア科学会 WISS'96, pp. 41–49. 近代科学社, 1996.

[22] Bernd Meyer. Pictures Depicting Pictures On the Specification of Visual Languages by Visual Grammars. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pp. 41–47, 1992.

[23] Brigham Bell and Clayton Lewis. Chemtrains: A Language for Creating Behaving Pictures. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, pp. 188–195, 1993.

[24] George W. Furnas. NEW GRAPHICAL REASONING MODELS FOR UNDERSTANDING GRAPHICAL INTERFACES. In *Proceedings of CHI'91*, pp. 71–78, 1991.

[25] 山本格也. ビットマップに基づくプログラミング言語 Visulan. インタラクティブシステムとソフトウェア III, 日本ソフトウェア科学会 WISS'95, pp. 151–160. 近代科学社, 1995.

付録 A

恵比寿のマニュアル

A.1 はじめに

A.1.1 恵比寿とは

恵比寿とは図形を用いた言語（図形言語）を解析，実行する一種のビジュアルプログラミングシステムです．これまでのビジュアルプログラミングシステムとの大きな違いは図形言語の文法とその動作（アクション）を与えることで様々なアプリケーションに変わるという点です．たとえば 計算の木を実行するアプリケーションや， GUI を作るアプリケーション などを文法と動作を与えるだけで作ることができます．

恵比寿のもう一つの特徴は，いくつかの図形が一つの意味のあるまとまり（図形単語）として認識されたら，今後それらをひとまとまりとして操作することができるという点です．たとえば，円の中にラベルとして文字が書いてあるものをノード，それらのノード間をつなぐ直線をエッジとしてグラフを表現するようなものを考えます．これまでの図形エディタなどを用いると，ノードのつもりで円を動かしても文字とエッジであるはずの直線がその円の動きに追随しませんが，恵比寿では文法中に書かれた 制約 を用いることで円を動かせば文字と直線もちゃんと追随します．これまでの図形エディタでもグループ化があるじゃないか，と思うかもしれません．しかしグループ化では一つの図形単語ではなく，グループ全体をひとまとまりとして操作してしまいます．一つのノードだけを動かそうと思ってもグループ化していた図形全体が動いてしまうのです．恵比寿では一つのノードを動かしたときにはそれにつながっているエッジの端点だけが追随します．

A.1.2 インストール

動作環境

現在次の環境で動作を確認しています．

- マシン /OS:SUNW,Ultra-1/SunOS5.5.1， SUNW,SPARCstation-5/SunOS5.4
- Tcl/Tk:7.5/4.1， 7.6/4.2， 8.0/8.0
- gcc:2.7.2.2

内容物

恵比寿は以下のファイル，ディレクトリから構成されています．

README	「恵比寿とは」と「起動と終了」が書かれています．
INSTALL	「インストール」が書かれています．
Makefile	make が使用します．
*.tcl	恵比寿の本体です．
get_font_size	フォントサイズのリストを得るために使われます．
.eviss	恵比寿の設定ファイルです．
examples/	例が入っています．
help/	ヘルプ（HTML 形式）が入っています．
images/	恵比寿が使用する GIF イメージが入っています．

Makefile の設定

Makefile 中のマクロを書き換えます．

VPG_DIR　恵比寿がインストールされているディレクトリのパス名

TCL_DIR　この配布キットで提供される Tcl のライブラリのパス名

場合によっては次のものも変更する必要があるかもしれません．

WISH　wish のコマンド名

TCLSH　tclsh のコマンド名

make

コマンドラインから次のコマンドを実行します．% はプロンプトです．

```
% make
```

これでeviss という名前のファイルができます．

A.1.3　起動と終了

恵比寿を起動するには，恵比寿がインストールされているディレクトリでコマンドラインから次のコマンドを実行します．% はプロンプトです．

```
% eviss
```

恵比寿を終了させるには，定義窓もしくは実行窓の File メニューから Exit を選択します．もし最後にルールをセーブしたのよりも後にルールが書き換えられていたら，実際に終了していいかどうか聞いてきます．

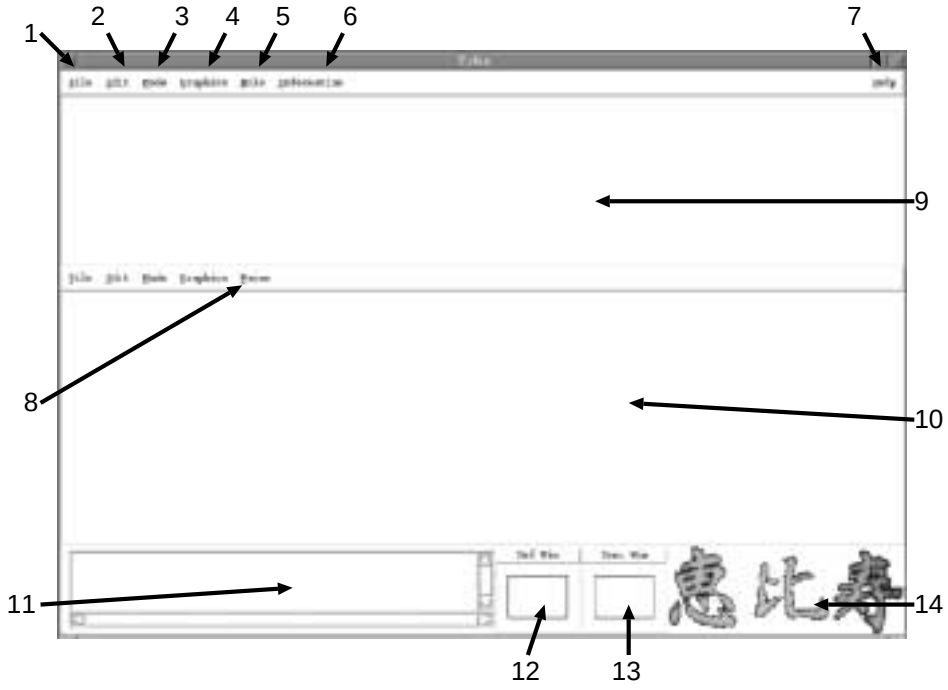


図 A.1: 各部の名称

A.1.4 各部の名称

恵比寿の外観を図 A.1に示します.

1 File メニュー A.2.1節参照.

2 Edit メニュー A.2.2節参照.

3 Mode メニュー A.2.3節参照.

4 Graphics メニュー A.2.4節参照.

5 Rule メニュー A.2.5節参照.

6 Information メニュー A.2.6節参照.

7 Help メニュー A.2.8節参照.

8 Parse メニュー A.2.7節参照.

9 定義窓 ここに図形単語のルールの定義の雛型を描きます.

10 実行窓 ここに描かれた図形が解析されます.

11 システムメッセージ 解析やシステムの状態に関するさまざまな情報を表示します．何を表示するかは Information メニューの Message で変更することができます．また，マウスの左ボタンをクリックすることで表示されているものを消すことができます．

12 定義窓の図形例 これから定義窓に描かれる図形がどんなものであるかは，画面の最も下に出ています．「Def Win」と書かれたところに描かれている図形が定義窓に描かれる図形のサンプルです．これは図形の属性を設定するときに変化します．

13 実行窓の図形例 これから実行窓に描かれる図形がどんなものであるかは，画面の最も下に出ています．「Exec Win」と書かれたところに描かれている図形が実行窓に描かれる図形のサンプルです．これは図形の属性を設定するときに変化します．

14 タイトル 恵比寿のタイトルです．マウスの左ボタンをクリックするとバージョン情報が表示されます．

A.1.5 ヘルプの使い方

ヘルプは，「目次」，「はじめに」，「メニュー」，「図形」，「ルール」，「例」から成ります．各ページのいちばん上のイメージをクリックするとそのページに移ることができます．また，目次からは各ページにリンクが張ってあります．

A.1.6 基本的な操作の流れ

まず定義窓で図形を描き，選択して Make New Rule し，ルールを定義し，さらにいくつかルールを定義し，実行窓に図形を描くと解析，実行されます．詳しくは例を見て下さい．

A.1.7 図形の描き方の基本

図形を描くにはまず描く図形の種類を選択します．恵比寿では長方形（rectangle），楕円（oval），円弧（arc），テキスト文字列（text），GIF イメージ（image），直線（line）を描くことができます．描く図形の種類の変更は，描きたい窓のメニューの Mode から描きたい図形を選択することでおこないます．他の窓で変更するとそちらの窓に描く図形の種類が変更されますので注意してください．最初は長方形が選択されています．各図形は線の太さなどの属性を持っています．

A.1.8 設定ファイル

設定ファイルは恵比寿を起動するときの様々な初期設定をします．もしも自分のホームディレクトリに．`eviss` という名前のファイルがあればそれを設定ファイルとして読み込みます．ないときは恵比寿をインストールしたディレクトリの`.eviss` が用いられます．ま

変数名	説明	取り得る値
mode	何が描かれるか	rectangle, oval, line, arc, text, image, select
ImageFile	イメージの名前とファイル名のペアが書かれたファイルの名前	ファイル名
outlinecolor	図形の線の色	red, green, yellow, blue, black, white, orange, none
fillcolor	図形の塗り潰す色	red, green, yellow, blue, black, white, orange, none
linewidth	線の太さ	0.25, 0.5, 1, 2, 4, 6, 8
linetype	直線の矢印の位置	none, first, last, both
fonttype	テキストのフォントの種類	courier, helvetica
halign	水平整列	top, middle, bottom
valign	垂直整列	left, center, right
Vp_CloseError	vp_close における誤差	整数値
NeedActualValue	ルールを作るときに実際の値を出すかどうか	0, 1
InitialRuleFile	恵比寿の起動時に呼ばれるルールを記述したファイルの名前	ファイル名
NeedRuleList	恵比寿の起動時にルールのリストを表示状態にするかどうか	0, 1
MaxSysMessages	システムメッセージの最大行数	整数値
NeedTokenList	恵比寿の起動時に図形単語のリストを表示状態にするかどうか	0, 1
SysMessage(***)	*** （表 A.2を参照のこと）で表されるものをシステムメッセージに表示するかどうか	0, 1
WriteToStdout	システムメッセージに表示するものを標準出力にも表示するかどうか	0, 1
Browser	ヘルプを見るための HTML のブラウザ	存在するブラウザ名
ParseMode	解析を自動でおこなうか，要求したときにおこなうか	auto, on_demand
UseInitialFunc	初期化手続きを呼ぶかどうか	0,1
InitialFuncName	初期化手続きが書かれたファイル名	ファイル名
InitialOpenDir	ファイルを開くときのデフォルトのディレクトリ	存在するパス名

表 A.1: 設定ファイルで設定できること

た，設定ファイル中に間違いがあると，間違っている設定は正しく設定し直され，その旨システムメッセージに表示されます．設定ファイルはこのデフォルトの.evissをコピーして書き換えることをお奨めします．設定ファイルで設定できるものを表 A.1に示します．

SysMessage の要素名と意味は以下の通りです．

A.1.9 実装されていない機能

次の機能はまだ実装されていません．

- UNDO 機能
- グループ化
- 一度描いた図形の形状の変更
- 後から描き加えた図形単語を all として認識させること
- 図形単語を all として認識させるために制約を与えること

要素名	説明
type	新しく作られた図形単語の種類
attributes	新しく作られた図形単語の属性
rule_id	現在解析に使用しているルールの Id
combinations	現在解析中の図形単語の組み合わせ
constraints_t	現在チェック中の制約のうち真のもの
constraints_f	現在チェック中の制約のうち偽のもの
negative_constraints_t	現在チェック中のネガティブ制約のうち真のもの
negative_constraints_f	現在チェック中のネガティブ制約のうち，偽のもの
delete	削除した図形の Id
search_neg_cn	ネガティブ制約表によってチェックしている図形単語の Id
candidates	これからチェックしようとしている組合せの候補となるもの

表 A.2: システムメッセージに表示できること

- ルールの不備により起こるエラーのチェック
- 定義窓の図形を CMG 入力部に書く時に，基本的な図形単語（長方形，テキスト文字列など）以外のものを書くこと

A.1.10 バグ

現在次のバグが見つかっています.

- CMG 入力部で，余計な空行があるとうまく解析できないことがある.
- 図形単語の描き換えをして図形を消すと，後からそれを使おうとして問題が生じることがある．たとえば計算の木を描き換える例で，後から手で付け加えるとエラーが出る.
- 図形を選択するときに失敗することがある.
- ルールを作った時に action 欄に空行が追加されてしまう，もしくは最後の一行が消されてしまう.
- 制約によっては図形単語の移動ができなくなってしまうことがある（制約の強さの問題?）.
- 解析後に複数の図形を移動させるとマウスの移動量よりも多く移動してしまうことがある.
- 制約が満たせなくても何のメッセージも出さない.
- on_circle 制約がうまくつかない.
- 過剰な制約を与えると制約のサイクルができる.
- Parse Initialize したあとに新たに描いてもうまく動かないことがある.

- ルールの不備などのために解析が途中で失敗して Tcl のエラーが発生すると解析が初期化されているわけでもないのに初期化できなくなることがある.
- 直線を移動させようとすると，始点は動かず，マウスカーソルの位置が直線の中点となるように直線が変形する.

A.2 メニュー

A.2.1 File メニュー

画面の保存（セーブ）

画面を保存するには保存したい窓の File メニューから Save window を選択します.

Tk のバージョンが 4.2 以上の場合はファイル名を入力，または選択するウィンドウが開きますのでファイル名を入力，または選択してください. 1）入力する部分でリターンキーを押す，2） Save と書かれたボタンを押す，3）ファイル名をダブルクリックする，のいずれかをおこなうとウィンドウが消え，指定したディレクトリに画面が保存されます．ファイルの拡張子は.can としてください．ファイル名に拡張子として.can をつけていない場合には拡張子.can が付加されます．保存するのをやめたいときには， Cancel ボタンを押してください.

Tk のバージョンが 4.1 以下の場合はファイル名を入力するウィンドウが開きますのでファイル名を入力してください．リターンキーを押すとウィンドウが消え，恵比寿を起動したディレクトリに画面が保存されます．パスを指定すれば他のディレクトリに保存することもできます．ファイルには拡張子として.can が付加されます．保存するのをやめたいときには，ファイル名を入力せずにリターンキーを押すか，ファイル名を入力するウィンドウの cancel ボタンを押してください.

保存した画面を呼び出すには，画面の呼び出し（ロード）をします.

画面の呼び出し（ロード）

保存した画面を呼び出すには，呼び出したい窓の File メニューから Load window を選択します.

Tk のバージョンが 4.2 以上の場合はファイル名を入力，または選択するウィンドウが開きますのでファイル名を入力，または選択してください. 1）入力する部分でリターンキーを押す，2） Open と書かれたボタンを押す，3）ファイル名をダブルクリックする，のいずれかをおこなうとウィンドウが消え，指定したディレクトリから画面が呼び出されます．ファイル名に拡張子として.can をつけていない場合には拡張子.can が付加されます．呼び出しをやめたいときには， Cancel ボタンを押してください.

Tk のバージョンが 4.1 以下の場合はファイル名を入力するウィンドウが開きますのでファイル名を入力してください．リターンキーを押すとウィンドウが消え，恵比寿を起動し

たディレクトリから画面が呼び出されます．パスを指定すれば他のディレクトリから呼び出すこともできます．指定したファイルが存在しない場合は，正しいファイル名が入力されるまで何回でもファイル名を入力するウィンドウが開いてファイル名を聞いてきます．拡張子の.can は付ける必要はありません（付けると呼び出せません）．呼び出しをやめたいときには，ファイル名を入力せずにリターンキーを押すか，ファイル名を入力するウィンドウのcancel ボタンを押してください．

画面を保存するには画面の保存（セーブ）をします．

恵比寿の終了

恵比寿を終了するには定義窓もしくは実行窓の File メニューから Exit を選択します．この際画面の保存は行われませんので注意してください．もし最後にルールをセーブした後にルールが書き換えられていた場合は，セーブするかどうか聞いてきます．「Yes」を選ぶとルールをセーブした後に終了します．「No」を選ぶと，本当に終了していいかどうか聞いてきます．ここで「Ok」を選ぶとセーブせずに終了します．

A.2.2 Edit メニュー

図形のコピー

図形をコピーするには，まずコピーしたい図形を選択します．次に，コピーしたい図形が描かれている窓の Edit メニューから Copy を選択します．すると元の図形の右下にコピーされた図形が表示されます．このとき元の図形の他に，コピーによって新たにできた図形が選択されます．

図形の削除

図形を削除するには，削除したい図形を選択します．次に，削除したい図形が描かれている窓の Edit メニューから Delete Selected Items を選択します．すると，選択されていた図形が削除されます．

窓の図形すべてを削除するには，削除したい窓の Edit メニューから Delete All Items を選択します．すると，窓からすべての図形が削除されます．一度削除した図形を復活させることはできませんので注意してください．

実行窓の図形は図形単語として認識されていきます．一つの図形を削除することによって図形単語としての条件を満たさなくなることがあります．その場合，これまでに図形単語間に与えられた制約はなくなります．

すべての図形の選択

窓の図形すべてを選択するには，選択したい窓の Edit メニューから Select All Items を選択します．すると，窓の全ての図形が選択されます．このとき，描く図形の種類のモー

ドは，select に変ります.

UNDO 機能（やり直し）

UNDO 機能は現在ありません.

A.2.3 Mode メニュー

図形の種類

これから描く図形の種類を決めるには，Mode メニューから描きたい図形の種類を選びます．定義窓か実行窓でマウスの右ボタンをクリックすることでも同じメニューが開きます．長方形（rectangle），楕円（oval），円弧（arc），テキスト文字列（text），GIF イメージ（image），直線（line）があります.

図形の選択

描いた図形を選択するにはメニューの Mode から select を選択します．選択したい図形の上でマウスの左ボタンをクリックすると図形が選択され，ハンドルと呼ばれる長方形が表示されます．このとき，すでに他に選択されている図形があった場合はその図形の選択は解除されます．シフトキーを押しながらマウスの左ボタンをクリックすると他の図形の選択を解除せずに新たに図形を選択することができます.

複数の図形をまとめて選択するには Mode から select を選択してからマウスの左ボタンをクリックして二つの点を指定します．するとこの二つの点によって定まる長方形の領域に含まれている図形が選択されます.

選択をすべて解除するにはマウスの右ボタンをクリックします．一つだけ選択を解除したいときにはシフトキーを押しながらマウスの右ボタンをクリックします.

A.2.4 Graphics メニュー

図形の属性の設定

描く図形の属性には線の色（Outline Color），塗り潰す色（Fill Color），線の太さ（Line Width），線の種類（Line Type），フォントの種類（Font Type），フォントの大きさ（Font Size）があります．これらの属性の値を設定することができます．また，図形の重なりの順番を変更することができます.

線の色

長方形（rectangle），楕円（oval），円弧（arc）の輪郭となる線の色を設定します．Graphics メニューの Outline Color を選択すると色名が書かれたサブメニューが開きます

のでそこから色を選択してください．赤（red），緑（green），黄色（yellow），青（blue），黒（black），白（white），オレンジ（orange），無色（none）を使うことができます．

塗り潰す色

長方形（rectangle），楕円（oval）での塗り潰す色，およびテキスト（text），直線（line）の色を設定します． Graphics メニューの Fill Color を選択すると色名が書かれたサブメニューが開きますのでそこから色を選択してください．赤（red），緑（green），黄色（yellow），青（blue），黒（black），白（white），オレンジ（orange），無色（none）を使うことができます．

線の太さ

長方形（rectangle），楕円（oval），円弧（arc），直線（line）の線の太さを設定します． Graphics メニューの Line Width を選択すると太さが書かれたサブメニューが開きますのでそこから太さを選択してください． 0.25point， 0.5point， 1point， 2point， 4point， 6point， 8point を使うことができます．

線の種類

直線（line）の始点，終点に矢印を付けるかどうかを設定することができます． Graphics メニューの Line Type を選択すると，矢印を付ける場所が書かれたサブメニューが開きますのでそこから矢印を付ける場所を選択してください．矢印なし（none），始点に付ける（first），終点に付ける（last），両方に付ける（both）があります．

フォントの種類

テキスト文字列（text）ではフォントの種類を設定することができます． Graphics メニューの Font Type を選択すると，フォントの種類が書かれたサブメニューが開きますのでそこからフォントの種類を選択してください． courier， helvetica を使うことができます．

フォントの大きさ

テキスト文字列（text）ではフォントの大きさを設定することができます． Graphics メニューの Font Size を選択すると，フォントの大きさが書かれたサブメニューが開きますので，そこからフォントの大きさを選択してください．使用できるフォントの大きさの種類はインストール時に決められ，システムによって異なります．

重なりの順番の変更

図形の重なりの順番を変更するには，順番を変えたい図形を選択します．順番を上にする場合， Graphics メニューから Raise を，下にする場合， Lower を選択します．すると，図形の重なりの順番が変わります．

新しい GIF イメージの追加

新しい GIF イメージを追加するには，定義窓か実行窓の Graphics メニューから Add New Image を選択します．するとファイル名を入力するためのウィンドウが開きますので，ここで GIF イメージのファイル名を入力して下さい．ここでキャンセルすると新しい GIF イメージは追加されません．次に， GIF イメージの名前を入力して下さい．ここでキャンセルすると， GIF イメージには， eviss_image*（* は数字）という名前が付けられます．これで新しい GIF イメージが追加されます．この GIF イメージは恵比寿の終了時まで使うことができます．恒久的に使いたい場合は， GIF イメージのリストを設定して下さい．

図形の属性の変更

一度描いた図形の属性を変更するには，まず属性を変更したい図形を選択します．次に，図形の属性を設定するのと同じ方法で Graphics メニューから属性を選ぶことで属性が変更されます．選択した図形がその属性を持っていない場合は属性は変更されません．

実行窓の図形は図形単語として認識されていきます．属性の変更をおこなうことによって図形単語としての条件を満たさなくなることがあります．その場合，これまでに図形単語間に与えられた制約はなくなります．

グループ化

グループ化の機能は現在ありません．

A.2.5 Rule メニュー

新しいルールを作る

新しいルールを作るには一つの図形単語としたい図形を選択して， Rule メニューから Make New Production Rule を選びます．するとルールを入力するためのウィンドウが開きます．このウィンドウは上から順に name， attributes， action， constraints，構成要素に分れています． name は図形単語の名前を書く欄です． attributes は図形単語の属性を書く欄です． action は図形単語のアクションを書く欄です． constraints は制約を書く欄です．構成要素はさらに normal， exist， not_exist， all に分れています．この四つの欄には図形単語の構成要素の種類 を書きます． normal の構成要素の種類の欄には選択された図

形の一覧が，また， constraints の欄には現在成り立っている eq 制約及び vp_close 制約の一覧が自動的に生成されます．

新しいルールを書き終わったらルール入力用のウィンドウのいちばん下の Ok と書かれたボタンを押してください．ルールが修正され，ルール入力用のウィンドウが消えます．ここで Cancel と書かれたボタンを押すと修正はされずにルール入力用のウィンドウが消えます．

すでにあるルールを変更することもできます．

ルールを削除する

ルールを削除するには， Rule メニューから Delete Production Rule を選びます．そうすると，削除するルールの id を入力するためのウィンドウが開きます． id はルールのリストを表示することで見ることができます．スライダーを使って id を設定してください．id を直接入力することもできます． id を入力したら Ok と書かれたボタンを押してください．中止するときには Cancel と書かれたボタンを押してください． Ok と書かれたボタンを押すと，念のためにもう一度確認してきますので，ここでも Ok または Cancel ボタンを押してください． Ok ボタンを押すとルールが削除され， Cancel ボタンを押すとなにもおきません．

ルールのリストで削除したいルールをマウスの右ボタンでクリックすることでも同様にルールを削除することができます．

すべてのルールを削除する

すべてのルールを削除するには， Rule メニューから Delete All Production Rules を選びます．最後にセーブした後にルールが書き換えられていた場合，全てのルールを削除していいか確認してきますので，削除してよければ Ok と書かれたボタンを，削除をやめるときは Cancel ボタンを押してください．

ルールをコピーする

ルールをコピーするには， Rule メニューから Copy Production Rule を選びます．そうすると，コピーするルールの id を入力するためのウィンドウが開きます． id はルールのリストを表示することで見ることができます．スライダーを使って id を設定してください．id を直接入力することもできます． id を入力したら Ok と書かれたボタンを押してください．中止するときには Cancel と書かれたボタンを押してください． Ok ボタンを押すとルールがコピーされ， Cancel ボタンを押すとなにもおきません．

ルールのリストを表示する

ルールのリストを表示するには Rule メニューから Show Rule List を選びます．するとこれまでに作ったルールの名前のリストが書かれたウィンドウが開きます．このウィンドウを消すには，いちばん下にある Dismiss と書かれたボタンを押してください．

このウィンドウでは，各ルールに対して次の三つの操作を行うことができます．

ルールの状態の変更 状態を変更したいルールの上でマウスの左ボタンをクリックすると，ルールの状態が， active または disabled に変ります．

ルールの変更 変更したいルールの上でマウスの中ボタンをクリックすると新しいルールを作るときと同じウィンドウが開きます．修正が終了したら画面のいちばん下の Ok と書かれたボタンを押してください．ルールが修正され，ルール入力用のウィンドウが消えます．ここで Cancel と書かれたボタンを押すと，修正はされずにルール入力用のウィンドウが消えます．

ルールの削除 削除したいルールの上でマウスの右ボタンをクリックすると，削除していいか確認してきますので，削除してよければ Ok ボタンを，やめるときは Cancel ボタンを押してください．

すでにあるルールを変更する

すでにあるルールを変更するには Rule メニューから Revise Production Rule を選びます．そうすると，変更するルールの id を入力するためのウィンドウが開きます． id はルールのリストを表示することで見ることができます．スライダーを使って id を設定してください． id を直接入力することもできます． id を入力したら Ok と書かれたボタンを押してください．中止するときには Cancel と書かれたボタンを押してください． Ok ボタンを押すと新しいルールを作るときと同じウィンドウが開きます． Cancel ボタンを押すとなにもおきません．

修正が終了したらルール入力用のウィンドウのいちばん下の Ok と書かれたボタンを押してください．ルールが修正され，ルール入力用のウィンドウが消えます．ここで Cancel と書かれたボタンを押すと修正はされずにルール入力用のウィンドウが消えます．

ルールの保存（セーブ）

ルールを保存するには Rule メニューから Save Production Rules を選択します．状態が active であるルールのみがセーブされます．

Tk のバージョンが 4.2 以上の場合はファイル名を入力，または選択するウィンドウが開きますのでファイル名を入力，または選択してください． 1）入力する部分でリターンキーを押す， 2） Save と書かれたボタンを押す， 3）ファイル名をダブルクリックする，のい

ずれかをおこなうとウィンドウが消え，指定したディレクトリにルールが保存されます．ファイルの拡張子は.rule としてください．ファイル名に拡張子として.rule をつけていない場合には拡張子.rule が付加されます．保存するのをやめたいときには， Cancel ボタンを押してください．

Tk のバージョンが 4.1 以下の場合はファイル名を入力するウィンドウが開きますのでファイル名を入力してください．リターンキーを押すとウィンドウが消え，恵比寿を起動したディレクトリにルールが保存されます．パスを指定すれば他のディレクトリに保存することもできます．ファイルには拡張子として.rule が付加されます．保存するのをやめたいときには，ファイル名を入力せずにリターンキーを押すか，ファイル名を入力するウィンドウの cancel ボタンを押してください．

保存したルールを呼び出すには，ルールの呼び出し（ロード）をします．

ルールの呼び出し（ロード）

保存したルールを呼び出すには， Rule メニューから Load Production Rules を選択します．これまでのルールに追加されて新しいルールがロードされます．ロードされたルールは active 状態になっています．

Tk のバージョンが 4.2 以上の場合はファイル名を入力，または選択するウィンドウが開きますのでファイル名を入力，または選択してください． 1）入力する部分でリターンキーを押す， 2） Open と書かれたボタンを押す， 3）ファイル名をダブルクリックする，のいずれかをおこなうとウィンドウが消え，指定したディレクトリからルールが呼び出されます．ファイル名に拡張子として.rule をつけていない場合には拡張子.rule が付加されます．呼び出しをやめたいときには， Cancel ボタンを押してください．

Tk のバージョンが 4.1 以下の場合はファイル名を入力するウィンドウが開きますのでファイル名を入力してください．リターンキーを押すとウィンドウが消え，恵比寿を起動したディレクトリからルールが呼び出されます．パスを指定すれば他のディレクトリから呼び出すこともできます．指定したファイルが存在しない場合は，正しいファイル名が入力されるまで何回でもファイル名を入力するウィンドウが開いてファイル名を聞いてきます．拡張子の.rule は付ける必要はありません（付けると呼び出せません）．呼び出しをやめたいときには，ファイル名を入力せずにリターンキーを押すか，ファイル名を入力するウィンドウの cancel ボタンを押してください．

ルールを保存するにはルールの保存（セーブ）をします．

vpclose の誤差を変更する

vp_close の誤差を変更するには， Rule メニューの， Change Error of vp_close を選択します．そうすると， vp_close の誤差を入力するためのウィンドウが開きます．スライダーを使って 0 から 100 までの間で数値を設定してください．数値を直接入力することも

できます．変更が終了したら Ok と書かれたボタンを押してください．変更を中止するときには Cancel と書かれたボタンを押してください．

実際の値を表示するかどうか

新しい図形単語を作るときには定義窓に描かれた図形の大きさや位置などの実際の値が CMG 入力部に書き出されます．これを書き出されないようにするためには， Rule メニューの Need Actual Value のチェックをはずしてください．マウスの左ボタンをクリックすることによりチェックを付けたりはずしたりすることができます．

A.2.6 Information メニュー

図形単語のリストを表示する

図形単語のリストを表示するには Information メニューから Show Token List を選びます．するとこれまでに認識された図形単語のリストが書かれたウィンドウが開きます．このリストの各行は 4 つの情報から成ります．これらは左から順に，図形単語の Id，すでに使われたかどうか（使われていなければ exists，使われていれば deleted），図形単語の種類，構成要素の Id のリストです．構成要素の Id のリストはさらに二つに分かれています．一つ目は normal の構成要素であり，二つ目は exist の構成要素です．リスト上の各図形単語の上でマウスの左ボタンをクリックすると，その図形単語の属性の値がシステムメッセージに表示され，その図形単語の構成要素（normal）すべてが選択されます．図形単語の削除，追加などがおこなわれた時にはそれが反映されます．

図形単語のリストを木構造で表示する

図形単語のリストを木構造上に表示するには Information メニューから Show Visualized Token List を選びます．するとこれまでに認識された図形単語のリストが描かれたウィンドウが開きます．青い線が normal を示し，赤い線が exist を示し，緑色の線が all を示します．ある図形単語を示すものをクリックすると，その図形単語の属性の値がシステムメッセージに表示され，その図形単語の構成要素（normal）すべてが選択されます．図形単語の削除，追加などが行なわれてもそれは反映されません．

システムメッセージをクリアする

システムメッセージをクリアするには， Information メニューの， Clear Messages を選択します．そうすると，システムメッセージがクリアされます．

名前	説明	初期状態
Type	解析されたものの種類	表示
Attributes	解析されたものの属性	表示
Rule Id	現在解析に使用しているルールの Id	非表示
Combinations	現在解析中の組み合わせ	非表示
Constraints(true)	成り立っているかどうかチェックしようとしている制約のうち真のもの	非表示
Constraints(false)	成り立っているかどうかチェックしようとしている制約のうち偽のもの	非表示
Negative Constraints(true)	成り立っているかどうかチェックしようとしている制約のうち，not_exist に書かれた構成要素を含むもので，かつ真のもの	非表示
Negative Constraints(false)	成り立っているかどうかチェックしようとしている制約のうち，not_exist に書かれた構成要素を含むもので，かつ偽のもの	非表示
Delete	削除された図形単語の種類と Id.	非表示
Search Negative Constraint Table	ネガティブ制約表によってチェックしている図形単語の Id	非表示
Candidates	これからチェックしようとしている組合せの候補となるもの	非表示

システムメッセージの最大行数を変更する

システムメッセージの最大行数を変更するには， Information メニューの， Change Max Lines を選択します．そうすると，最大行数を入力するためのウィンドウが開きます．スライダーを使って 10 から 1000 までの間で数値を設定してください．数値を直接入力することもできます．変更が終了したら Ok と書かれたボタンを押してください．変更を中止するときには Cancel と書かれたボタンを押してください.

システムメッセージに表示するものを変更する

解析をおこなっているときに情報を表示したり非表示にしたりすることができます．この切り替えをおこなうには， Information メニューの Message にあるものの上でマウスの左ボタンをクリックすることでチェックを付けたりはずしたりしてください．表示できるものには次のものがあります.

システムメッセージの内容を標準出力にも表示する

システムメッセージの内容を標準出力にも表示するかどうかを切替えるには， Information メニューの Write to standard output のチェックを付けたりはずしたりすることによりおこなえます.

バージョン情報を表示する

バージョン情報を表示するには， Information メニューの Version を選択して下さい.

A.2.7 Parse メニュー

解析のモード

解析のモードには Auto と On Demand があります．解析モードが Auto のときは，実行窓に新しい図形が描かれるたびに解析がおこなわれます．一方，解析モードが On Demand のときには要求があったときにのみ解析がおこなわれます．解析のモードを Auto にするには実行窓の Parse メニューから Auto を選びます．解析のモードを On Demand にするには実行窓の Parse メニューから On Demand を選びます．

図形の解析

解析のモードが On Demand モードのときに実行窓に描いた図形を解析する要求を出すには実行窓の Parse メニューから Parse を選びます．このとき，最後に「Parse」をおこなった後に作成された図形が新たに解析に用いられます．

選択した図形の解析

解析のモードが On Demand モードのときに，実行窓にある図形のうち，選択した図形を用いて解析をおこなうには，実行窓の Parse メニューから Parse Selected Items を選びます．一度解析に用いられた図形を選択していた場合の動作は保証されませんので注意してください． Parse Selected Items を行う際にはまず，解析の初期化をおこなうことをお奨めします．

解析の初期化

これまでにおこなった解析を無効にするためには，解析の初期化をおこないます．初期化手続きをおこなうことにしていると，このときに一般の初期化以外に初期化手続きのファイルが読まれて実行されます．解析の初期化をおこなうためには実行窓の Parse メニューから， Initialize を選択します．初期化されている場合には，メニューの Initialize の文字は灰色になっていて選択することができません．実行窓のすべての図形の削除をおこなったときにも解析の初期化がおこなわれます．

初期化手続きのファイル名の変更

初期化手続きをおこなうファイルのファイル名を変更するには，実行窓の Parse メニューから， Change Initial Func を選択します．

Tk のバージョンが 4.2 以上の場合はファイル名を入力，または選択するウィンドウが開きますのでファイル名を入力，または選択してください． 1）入力する部分でリターンキーを押す， 2） Open と書かれたボタンを押す， 3）ファイル名をダブルクリックする，のいずれかをおこなうとウィンドウが消え，初期化手続きをおこなうファイルのファイル名が変

更されます．なお，ファイル名の拡張子が.tcl以外のものはファイル名のリストに表示されません．

Tk のバージョンが 4.1 以下の場合はファイル名を入力するウィンドウが開きますのでファイル名を入力してください．リターンキーを押すとウィンドウが消え，初期化手続きをおこなうファイルのファイル名が変更されます．

初期化手続きをおこなうかどうか

実行窓の Parse メニューから， Use Initial Func をクリックすることで初期化手続きをおこなうかどうかを変更することができます．

A.2.8 Help メニュー

ヘルプ

Help メニューから Help を選択すると，環境変数 BROWSER で設定された HTML のブラウザが起動され，このヘルプを見ることができます．

A.3 図形

A.3.1 長方形

長方形の描き方

長方形（rectangle）を描くには，まず，長方形の左上の頂点をマウスの左ボタンをクリックすることで指定します．ボタンを押したままマウスを移動させる（ドラッグする）ことで最初の点と現在のマウスカーソルがある点を対角線とする長方形が表示されます．マウスのボタンを離すと，長方形が描かれます．

長方形の属性

長方形を描くときには以下の属性を設定することができます．

型	属性名	説明
integer	*linewidth*	線の太さ（Line Width）
string	*linecolor*	線の色（Outline Color）
string	*innercolor*	内部の色（Fill Color）

内部属性

長方形は内部では以下の属性を持っています．

型	属性名	説明
point	$leftupper$	左上の点の座標（x,y 座標の組）
integer	lu_x	左上の点の x 座標
integer	lu_y	左上の点の y 座標
point	$rightlower$	右下の点の座標（x,y 座標の組）
integer	rl_x	右下の点の x 座標
integer	rl_y	右下の点の y 座標
point	mid	対角線の交点の座標（x,y 座標の組）
integer	mid_x	対角線の交点の x 座標
integer	mid_y	対角線の交点の y 座標
integer	$width$	長方形の x 方向の長さ（幅）
integer	$height$	長方形の y 方向の長さ（高さ）

lu_x, lu_y, rl_x, rl_y は描いたときに決定されます．そのほかの属性はこの四つの属性から次の式により生成されます．

$$\begin{aligned} leftupper &= (lu\_x, lu\_y) \\ rightlower &= (rl\_x, rl\_y) \\ mid &= (mid\_x, mid\_y) \\ mid\_x &= (lu\_x + rl\_x)/2 \\ mid\_y &= (lu\_y + rl\_y)/2 \\ width &= rl\_x - lu\_x \\ height &= rl\_y - lu\_y \end{aligned}$$

一つの属性の値が変更されるとこれらの式を満たすように他のすべての属性の値が変更されます．

A.3.2 楕円

楕円の描き方

楕円（oval）を描くには，まず，楕円に外接する長方形の左上の頂点をマウスの左ボタンをクリックすることで指定します．ボタンを押したままマウスを移動させる（ドラッグする）ことで最初の点と現在のマウスカーソルがある点を対角線とする長方形に内接する楕円が表示されます．マウスのボタンを離すと，楕円が描かれます．

楕円の属性

楕円を描くときには以下の属性を設定することができます．

型	属性名	説明
integer	$linewidth$	線の太さ（Line Width）
string	$linecolor$	線の色（Outline Color）
string	$innercolor$	内部の色（Fill Color）

楕円の内部属性

楕円は内部では以下の属性を持っています．

型	属性名	説明
point	$leftupper$	外接する長方形の左上の点の座標（x,y 座標の組）
integer	lu_x	外接する長方形の左上の点の x 座標
integer	lu_y	外接する長方形の左上の点の y 座標
point	$rightlower$	外接する長方形の右下の点の座標（x,y 座標の組）
integer	rl_x	外接する長方形の右下の点の x 座標
integer	rl_y	外接する長方形の右下の点の y 座標
point	mid	中心の座標（x,y 座標の組）
integer	mid_x	中心の x 座標
integer	mid_y	中心の交点の y 座標
integer	$diameter\_x$	x 方向の直径
integer	$diameter\_y$	y 方向の直径
integer	$radius\_x$	x 方向の半径
integer	$radius\_y$	y 方向の半径

lu_x, lu_y, rl_x, rl_y は描いたときに決定されます．そのほかの属性はこの四つの属性から次の式により生成されます．

$$
\begin{aligned}
leftupper &= (lu\_x, lu\_y) \\
rightlower &= (rl\_x, rl\_y) \\
mid &= (mid\_x, mid\_y) \\
mid\_x &= (lu\_x + rl\_x)/2 \\
mid\_y &= (lu\_y + rl\_y)/2 \\
diameter\_x &= rl\_x - lu\_x \\
diameter\_y &= rl\_y - lu\_y \\
radius\_x &= diameter\_x/2 \\
radius\_y &= diameter\_y/2
\end{aligned}
$$

一つの属性の値が変更されるとこれらの式を満たすように他のすべての属性の値が変更されます．

A.3.3 円弧

円弧の描き方

円弧（arc）を描くには，まず，円弧を含む楕円に外接する長方形の左上の頂点をマウスの左ボタンをクリックすることで指定します．ボタンを押したままマウスを移動させる（ドラッグする）ことで最初の点と現在のマウスカーソルがある点を対角線とする長方形に内接する楕円が表示されます．マウスのボタンを離すと，円弧の始点となる角度（0° ～ 360°）を入力するためのウィンドウが開きます．入力したらリターンキーを押してください．次に始点からどのくらいの角度だけ広がるか（0° ～ 360°）を入力するためのウィンドウが開きます．入力したらリターンキーを押してください．これで円弧が描かれます．

円弧の属性

円弧を描くときには以下の属性を設定することができます．

型	属性名	説明
integer	*linewidth*	線の太さ（Line Width）
string	*linecolor*	線の色（Outline Color）

円弧の内部属性

円弧は内部では以下の属性を持っています.

型	属性名	説明
point	$leftupper$	円弧が一部となる楕円に外接する長方形の左上の点の座標（x,y 座標の組）
integer	lu_x	円弧が一部となる楕円に外接する長方形の左上の点の x 座標
integer	lu_y	円弧が一部となる楕円に外接する長方形の左上の点の y 座標
point	$rightlower$	円弧が一部となる楕円に外接する長方形の右下の点の座標（x,y 座標の組）
integer	rl_x	円弧が一部となる楕円に外接する長方形の右下の点の x 座標
integer	rl_y	円弧が一部となる楕円に外接する長方形の右下の点の y 座標
point	mid	円弧が一部となる楕円の中心の座標（x,y 座標の組）
integer	mid_x	円弧が一部となる楕円の中心の x 座標
integer	mid_y	円弧が一部となる楕円の中心の交点の y 座標
integer	$diameter\_x$	円弧が一部となる楕円の x 方向の直径
integer	$diameter\_y$	円弧が一部となる楕円の y 方向の直径
integer	$radius\_x$	円弧が一部となる楕円の x 方向の半径
integer	$radius\_y$	円弧が一部となる楕円の y 方向の半径

lu_x, lu_y, rl_x, rl_y は描いたときに決定されます．そのほかの属性はこの四つの属性から次の式により生成されます.

$$
\begin{aligned}
leftupper &= (lu\_x, lu\_y) \\
rightlower &= (rl\_x, rl\_y) \\
mid &= (mid\_x, mid\_y) \\
mid\_x &= (lu\_x + rl\_x)/2 \\
mid\_y &= (lu\_y + rl\_y)/2 \\
diameter\_x &= rl\_x - lu\_x \\
diameter\_y &= rl\_y - lu\_y \\
radius\_x &= diameter\_x/2 \\
radius\_y &= diameter\_y/2
\end{aligned}
$$

一つの属性の値が変更されるとこれらの式を満たすように他のすべての属性の値が変更されます.

A.3.4 テキスト文字列

テキスト文字列の描き方

テキスト文字列（text）を描くには，まず，テキスト文字列の中心となる点をマウスの左ボタンをクリックすることで指定します．するとテキスト文字列を入力するウィンドウが開きますのでそこにテキスト文字列を入力してください．リターンキーを押すとそのウィンドウが消え，指定した点にテキスト文字列が描かれます.

テキスト文字列を描くのをキャンセルするには，文字列を入力せずにリターンキーを押すか，文字列を入力するウィンドウの cancel ボタンを押してください.

テキスト文字列の変更

一度描いたテキスト文字列を変更するには，描く図形の種類をテキスト文字列（text）にします.

テキスト文字列の属性

テキストを描くときには以下の属性を設定することができます.

型	属性名	説明
string	*color*	色（Fill Color）
string	*font*	フォントの種類（Font Type）
string	*font*	フォントの大きさ（Font Size）

テキスト文字列の内部属性

テキストは内部では以下の属性を持っています.

型	属性名	説明
point	*mid*	中心の座標（x,y 座標の組）
integer	*mid_x*	中心の x 座標
integer	*mid_y*	中心の y 座標
string	*text*	書かれている文字列
string	*font*	フォント

mid_x， mid_y， $text$， $font$ は描いたときに決定されます． mid は mid_x， mid_y から次の式により生成されます.

$$mid \quad = \quad (mid\_x, mid\_y)$$

一つの属性の値が変更されるとこれらの式を満たすように他のすべての属性の値が変更されます.

A.3.5 GIF イメージ

GIF イメージの描き方

GIF イメージ（image）を描くには，まず， GIF イメージの中心となる点をマウスの左ボタンを押すことで指定します．このとき描くことができる GIF イメージのリストが表示されますのでそこから表示させたい GIF イメージを選択して左ボタンを離して下さい．するとリストが消え，指定した点に GIF イメージが描かれます.

GIF イメージのリスト

GIF イメージを描く時に表示されるリストの内容は，起動時に設定ファイルに書かれている変数 `ImageFile` で表されるファイルに書かれているものによって決まります．このファイルの各行には GIF イメージに付ける名前とその GIF イメージのファイル名が書かれています．リストとしてこのリスト以外のものを使いたい場合は，`ImageFile` を別なファイル名に設定し，そのファイル中にリストを書いて下さい．恵比寿の実行中に新しく GIF イメージを追加することもできます．

GIF イメージの属性

GIF イメージを描くときには以下の属性を設定することができます．

型	属性名	説明
string	*image*	GIF イメージの名前

GIF イメージの内部属性

GIF イメージは内部では以下の属性を持っています．

型	属性名	説明
point	*mid*	中心の座標（x,y 座標の組）
integer	*mid_x*	中心の x 座標
integer	*mid_y*	中心の y 座標

mid_x， mid_y，は描いたときに決定されます． mid は mid_x， mid_y から次の式により生成されます．

$$mid \quad = \quad (mid\_x, mid\_y)$$

一つの属性の値が変更されるとこれらの式を満たすように他のすべての属性の値が変更されます．

A.3.6 直線

直線の描き方

直線（line）を描くには，まず，直線の始点をマウスの左ボタンをクリックすることで指定します．ボタンを押したままマウスを移動させる（ドラッグする）ことで最初の点と現在のマウスカーソルがある点を結ぶ直線が表示されます．マウスのボタンを離すと直線が描かれます．

直線の属性

直線を描くときには以下の属性を設定することができます.

型	属性名	説明
integer	*linewidth*	線の太さ（Line Width）
string	*color*	線の色（Outline Color）
string	*arrow*	線の種類（Line Type）

直線の内部属性

直線は内部では以下の属性を持っています.

型	属性名	説明
point	*start*	始点の座標（x,y 座標の組）
integer	*start_x*	始点の x 座標
integer	*start_y*	始点の y 座標
point	*end*	終点の座標（x,y 座標の組）
integer	*end_x*	終点の x 座標
integer	*end_y*	終点の y 座標
point	*mid*	始点と終点の中点の座標（x,y 座標の組）
integer	*mid_x*	始点と終点の中点の x 座標
integer	*mid_y*	始点と終点の中点の y 座標
integer	*width*	外接する長方形の x 方向の長さ（幅）
integer	*height*	外接する長方形の y 方向の長さ（高さ）

$start\_x$, $start\_y$, end_x, end_y は描いたときに決定されます. そのほかの属性はこの四つの属性から次の式により生成されます.

$$\begin{aligned} start &= (start\_x, start\_y) \\ end &= (end\_x, end\_y) \\ mid &= (mid\_x, mid\_y) \\ mid\_x &= (start\_x + end\_x)/2 \\ mid\_y &= (start\_y + end\_y)/2 \\ width &= end\_x - start\_x \\ height &= end\_y - start\_y \end{aligned}$$

一つの属性の値が変更されるとこれらの式を満たすように他のすべての属性の値が変更されます.

A.4 ルール

図形言語はいくつかの図形単語 によって構成されます. それぞれの図形単語の記述をルールと呼びます. ルールは CMG （Constraint Multiset Grammars）という形式に基づいて CMG 入力部に記述します. 具体的には, 図形単語の名前 , 属性 , アクション , 構成要素 , 構成要素間の制約を書きます. 恵比寿では一つのルールは active か disabled のどちらかの状態をとります. 解析には active 状態にあるルールのみが使用されます. メニューからルールの状態の変更をおこなうことができます.

A.4.1 CMG 入力部

CMG 入力部は以下のような構成になっています.

- 図形単語の名前
- 属性
- アクション
- 構成要素間の制約
- 構成要素

これらのウィンドウでは入力の他に次のようなキーボード操作をおこなうことができます. C-* はコントロールキーを押しながら * を押すことを意味します.

操作	結果
C-F	右のウィンドウに移動
C-B	左のウィンドウに移動
C-N	下のウィンドウに移動
C-P	上のウィンドウに移動
C-xC-s	ルールの作成（もしくは変更）
C-xC-c	CMG 入力部を消す

A.4.2 用語

図形単語

普通のテキスト言語ではなにかを表すためにいくつかの文字をつなげて単語を作ります. この考え方を図形言語にあてはめると，図形言語で何かを表すためにはいくつかの図形を組み合わせて図形単語を作るということになります. たとえば円の中にテキストが書いてあったらノードという図形単語である，という具合いです.

図形単語とテキスト言語の単語の大きな違いとして，テキスト言語では文字を縦や横など一次元的に並べて単語を作るのに対し，図形言語では円やテキストなどの図形を二次元的に組み合わせて作るという点が挙げられます. このことは，テキスト言語の単語では構成要素である文字と文字との関係は「隣り合っている」というものだけだったのに対し，図形言語の図形単語では構成要素である図形と図形との間に，「内包している」，「同じ点を共有している」，「右側にある」など様々な関係があるということを意味しています. また，図形単語では複数の図形単語を組み合わせてより複雑な図形単語を表すことができます. 「目」，「鼻」，「口」，「輪郭」という図形単語を組み合わせて「顔」という図形単語を作る，というのがその例です.

A.4.3　制約

制約とは

恵比寿では図形間の関係を表すのに「制約」を用います．制約とは，いくつかの変数の間に常になり立っている関係です．たとえば，三つの変数 a , b , c の間に $a+b=c$ という制約が成り立っているとします．今，a，b の値がそれぞれ 3，4 だとすると，c の値は 7 です．このとき，a の値を 5 に変えたとします．すると，この制約を成り立たせるためには，c の値を 9 にするか，または b の値を 2 にしなければなりません．また，別な例として，三つの変数 a，b，c の間に（1） $a = b$，（2） $b = c$ という二つの制約が成り立っているとします．今，a の値が 5 だとしたら，b，c の値も 5 です．このとき，c の値を 8 にしたら，（2）から b の値は 8 になり，その影響を受けて（1）から a の値も 8 にしなければなりません．このように，複数の変数の間になり立っている制約を常に成り立たせるための機構を制約解消系と呼びます．恵比寿の内部では制約解消系として SkyBlue というものを使用しています．

変数の型

変数の型には次のものがあります．

名前	説明
integer	整数
string	文字列
point	点

eq 制約

eq 制約は二つの変数の値が等しい（Equal）ことを意味します．この制約が一度成り立つと一つの変数の値が変っても制約解消系によって常にこの制約が成り立つように他の変数の値が書き換えられます． eq 制約は具体的には次のように記述します．ただし $var1$ と $var2$ は変数名です．

`eq` $var1$ $var2$

変数 $var1$，$var2$ の型としては integer， string， point を用いることができます．ただし，一方が point 型の時は他方も point 型である必要があります．

neq 制約

neq 制約は二つの変数の値が等しくない（Not EQual）ことを意味します．この制約は一度成り立っても一つの変数の値が変ると維持されません． neq 制約は具体的には次のように記述します．ただし $var1$ と $var2$ は変数名です．

`neq` $var1$ $var2$

変数 $var1$，$var2$ の型としては integer， string， point を用いることができます．

gt 制約，ge 制約

変数名 が $var1$， $var2$ であるような二つの変数があったときに，

`gt` $var1$ $var2$

と記述すると $var1$ は $var2$ より大きい（Greater Than）ことを，

`ge` $var1$ $var2$

と記述すると $var1$ は $var2$ より大きいか等しい（Greater or Equal）ことを，それぞれ意味します．この制約が一度成り立つと一つの変数の値が変っても制約解消系によって常にこの制約が成り立つように他の変数の値が書き換えられます．

変数 $var1$， $var2$ の型としては integer を用いることができます．

lt 制約，le 制約

変数名が $var1$， $var2$ であるような二つの変数があったときに，

`lt` $var1$ $var2$

と記述すると $var1$ は $var2$ より小さい（Less Than）ことを，

`le` $var1$ $var2$

と記述すると $var1$ は $var2$ より小さいか等しい（Less or Equal）ことを，それぞれ意味します．この制約が一度成り立つと一つの変数の値が変っても制約解消系によって常にこの制約が成り立つように他の変数の値が書き換えられます．

変数 $var1$， $var2$ の型としては integer を用いることができます．

vpclose 制約

vp_close 制約は二つの変数（integer 型， double 型， point 型のいずれか）の値が十分に近いことを意味します．この制約が一度成り立つと eq 制約が課せられ，一方の値が変化すると他方も同じ値に変化します． vp_close 制約は次のように記述します．ただし $var1$ と $var2$ は変数名です．

`vp_close` $var1$ $var2$

oncircle 制約

on_circle 制約はある点が一つの円周上にあることを意味します．この制約は一度成り立っても一つの変数の値が変ると維持されません． on_circle 制約は具体的には次のように記述します．

`on_circle` p mid $radius$

ただし p， mid は point 型の変数， $radius$ は integer 型の変数をあらわす変数名で， p は円周上の点， mid は円の中心， $radius$ は円の半径です．

A.4.4 文法

変数名

恵比寿での変数は現在定義中の図形単語の構成要素となる図形単語の一つの属性を表しています．恵比寿では変数を次の形で表します．

$type.id.name$

$type$ は構成要素となる図形単語が normal 制約， exist 制約， not_exist 制約， all 制約のどこで用いられるかによって決まり，順に `0`， `1`， `2`， `3` となります． id は， $type$ で表される制約の中で何番目の構成要素であるかを表します． $name$ はその図形単語の属性の名前です．たとえば normal 制約の 3 番目の構成要素の mid という属性を表す変数の変数名は次のようになります．

```
0.3.mid
```

名前

定義する図形単語の名前には空白を含まない英数字を使用することができます．

属性

定義する図形単語の一つの属性は次の形式で表します．

$type$ $name$ $value$

$type$ は属性の型， $name$ は属性名， $value$ は属性の値です．属性名には空白を含まない英数字を使用することができます．

属性の値は次の三つのうちのいずれかです．

1. 構成要素の属性の値を継承した値

2. 新たに作る定数

3. 構成要素の属性の値から合成した値

1 は継承したい構成要素の属性の変数名です． 2 は次の形式で記述します．

{*type value*}

type は型， *value* は定数の値です． 3 は次の形式で記述します．

{script.*type* {*script*}}

type は型， *script* は Tcl のスクリプトですが，変数名をアットマーク（@）で囲むことによって構成要素の属性の値を参照することができます． 1 の場合は構成要素の属性の値が変更されると，新たに作られる図形単語の属性の値も変化しますが， 3 の場合はスクリプト中で参照する構成要素の属性の値が変化してもその影響を受けません．すなわち， 3 の場合は構成要素の属性の値をもとに新たな定数を作っていると考えてください．

構成要素

定義する図形単語の構成要素には構成要素の名前を書きます．最初から定義されている構成要素は rectangle （長方形）， circle （楕円）， text （テキスト文字列）， line （直線）， image （GIF イメージ）， arc （円弧）があります．例えば一つの直線と二つの楕円を normal の構成要素として持つ場合， normal の構成要素の欄に次のように記述します．

```
line
circle
circle
```

normal の欄には，定義する図形単語の部品となる構成要素を書きます．これらすべての存在が前提となります．つまり，ここに書かれたもののうち一つでも欠けると図形単語としては認識されません（つまり「and」）．

exist の欄には，定義する図形単語を認識するために，図全体の中のどこかに存在する必要があるものを書きます．これも一つでも欠けると図形単語としては認識されなくなります（つまり「and」）．ここに書かれたものは図形単語の部品としては使用されません．

not_exist の欄には，定義する図形単語を認識するために，その図全体の中のどこかに存在してはならないものを書きます．ここに書かれたもののうちどれか一つでも存在したら図形単語としては認識されません（つまり「or」）．

all の欄に書くと，制約を満たすもの全てを集めます．インクリメンタルに all の要素として認識する機能と， all の欄に書かれたものを使った制約を書く機能は，まだサポートされていません．

制約

構成要素の欄に書いたものを使って制約をこの欄に書きます．制約の中に， not_exist の構成要素が入っている場合は，その制約はネガティブ制約となります．通常の制約は成り立ったときに真となり，実際に図形間に制約を課しますが，ネガティブ制約はその制約が成り立たなかったときに真となり，図形間には制約を課しません．

A.4.5 アクション

アクションの書き方

アクションは基本的には Tcl のスクリプトですが，次の三種類をアットマーク（@）で囲むと，その部分がまず以下の表に示すものに置換されます．

書き方	置換されるもの
name	新たに作られた図形単語の *name* で表される属性の値
type.*id*	実行窓に描かれた図形単語を一つのルールの中の *type*.*id* で表される構成要素に当てはめますが，実行窓の図形単語自体も id を持っています． *type* .*id* によってその図形単語の id を参照します．
type.*id*.*name*	*type*.*id*.*name* で表される構成要素の図形単語の属性（変数）の値

恵比寿が供給する手続き

恵比寿はアクションの中で Tcl のスクリプトを書くのに便利な手続きをいくつか提供しています．

- 図形単語の描き換えをおこなうもの
- システムメッセージへの書き込みをおこなうもの
- カウンタ

図形単語の描き換えをおこなうための手続きとしては，図形単語の属性の値の変更と図形単語の削除をおこなうものがあります．

図形単語の属性の値を変更するための手続きとして，次のものがあります．

`alter` *item_id attr_name new_value*

これで *item_id* を id として持つ図形単語の属性 *attr_name* の値が *new_value* にかわります．

図形単語の削除を行うための手続きとして次のものがあります．

`delete` {?*item_id*1? ?*item_id*2? ?...?}

これで *item_id*1, *item_id*2, …を id として持つ 0 個以上の図形単語が削除されます.

これらの手続きを使用して図形単語の描き換えをおこなうと認識された図形単語としての条件を満たさなくなることがあります. その場合でもこれまでに図形単語間に与えられた制約は残ります.

システムメッセージへの書き込みをおこなうための手続きとして, 次のものがあります.

`sys_message` *message* ?*color*?

これでメッセージ *message* をシステムメッセージに書き込むことができます. *message* の部分では Tcl によって置換がおこなわれますので変数の値の表示などデバッグに使うこともできます. *color* を指定するとその色で表示されます. 指定しない場合は黒で表示されます.

カウンタは次の二つの形態で提供しています.

1. 操作名 カウンタ名
2. カウンタ名 操作名

まず 1 の形態のカウンタの説明をします. 操作としては, 生成, カウンタの値を一つ進めてその値を得る, 現在の値を得る, の三つです. それぞれ次のようにします. *name* がカウンタの名前です.

`create_counter` *name*
`new_value` *name*
`get_value` *name*

`create_counter` によって生成されたカウンタは以降この形態で使用します. `new_value` は, 「カウンタ名 n」(n は新しい値) の形を返してきます. `get_value` は「カウンタ名 n」(n は現在の値) の形を返してきます.

次に 2 の形態のカウンタの説明をします. 操作としては, 生成, カウンタの値を一つ進めてその値を得る, 現在の値を得る, 0 に戻す, 削除があります.

`counter` ?*name*?
name `new_value`
name `get_value` ?`-num`?
name `reset`
name `delete`

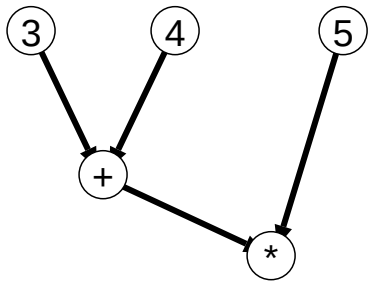


図 A.2: 計算の木

`counter` は *name* を名前とするカウンタを一つ生成します．これは *name* を名前とするコマンドとそれに関連する変数を生成することを意味します．ちょうど Tk のウィジェットの生成と同じです． *name* を指定しないと「`counter`n」（n はカウンタの Id）という名前のカウンタを生成します．`new_value` と`get_value` では，「カウンタ名_n」という形式のものを返しますが，`get_value` で `-num` オプションを付けるとカウンタ名を付けないものを返します． `reset` はカウンタを 0 に戻して「カウンタ名 0」という形を返します． `delete` はカウンタの変数と *name* というコマンドを削除して *name* を返します．

初期化手続き

解析の初期化がおこなわれる時に一回だけ評価されるファイルです．

A.5 使用方法 – 例を通して –

A.5.1 基本編

概要

ここでは計算の木を実行するようなビジュアル言語を定義する例を通して恵比寿の基本的な使用法を述べます．とりあえずは図 A.2のような図を描くと， 35 という結果を得ることができるようなものとします．

normal のみを用いた場合

ここでは normal のみを用いて定義する方法を作り方のコツを交えながら詳しく述べます．基本的には次の二つのルールを定義することによってこの言語を定義できます．

1. ノードは円の中に数字が描いてある．
2. ノードは二つのノードが矢印によって円につながれ，その円の中には演算子が描かれている．

図 A.3: 数字のノード

1 のルールを実際に定義していく過程を述べます.

1. 定義窓に楕円（円）を一つ描きます.
2. その円の中心付近に適当なテキスト文字列（図 A.3では「num」と描かれていますが, なんでも構いません）を描きます.
3. それらの図形をまとめて選択します.
4. Rule メニューから Make New Production Rule を選択します. CMG 入力部が開きます. 設定を変えていなければ, constraints の欄に, `vp_close 0.0.mid 0.1.mid` と書かれ, normal の欄に,

   ```
   circle
   text
   ```

   と書かれます.
5. name の欄に

   ```
   node
   ```

   と書きます. これは新しく定義する図形単語の名前が node であることを表します.
6. attributes の欄に

   ```
   point mid 0.0.mid
   integer value {script.integer {set dummy @0.1.text@}}
   integer left 0.0.lu_x
   integer right 0.0.rl_x
   ```

   と書きます. name の欄から attributes の欄への移動は Control-N を用いると便利です. 1 行目は新しく定義する図形単語の属性 mid の型を point 型にし, それを 0.0.mid, つまり normal（0）に書かれたものの 0 番目, すなわち circle の属性 mid と同じものとしています. 2 行目は, script 指令を用いて, 0.1.text（つまり, text の文字列）の値を integer 型に変換して属性 value の値としています. 「`set dummy` 〜」は型変換を行なうための一般的な方法です（もっといい方法があるかもしれませんが）. 3, 4 行目はそれぞれ属性 left, right を 0.0（circle）の lu_x（左上の点の x 座標）, rl_x（左下の点の x 座標）と同じものにしています.

7. action の欄に

   ```
   puts "value = @value@"
   ```

   と書きます．これは，この図形単語が認識された時に標準出力に属性 value の値を使って，

   ```
   value = 4
   ```

   のように表示するように指定しています．

8. constraints の欄に

   ```
   eq 0.0.innercolor {string white}
   ```

   と書きます．これは，0.0 の innercolor（塗りつぶす色）が`{string white}`，つまり文字列定数の ”`white`”であることを意味しています．この制約が必要になる理由は後で述べます．

9. Ok と書かれたボタンを押します． CMG 入力部が閉じます．

これで一つ目のルールが定義できました．実行窓に描いて認識させてみましょう．まず，白い楕円を一つ描いて，その中心付近にテキスト文字列「3」を描いてみて下さい．設定を変更していなければシステムメッセージに青い文字で「`node was parsed.`」と書かれ，そのあとさらに「`mid = <220 95>`」のように属性とその値がずらずらと書き出されるはずです．そして標準出力には「`value = 3`」と表示されていると思います．表示されない場合次の理由が考えられます（「……」のあとは対処法です）．

- 楕円の中心と文字列の中心が一致していない……いったん図を消してから，中心を一致させるように描くか，または，`vp_close` の誤差を広くとる．
- ルールが正しくない……前の 1 〜 9 を見ながら正しくルールを書く．その後図を描き直す．
- 解析のモードが On Demand になっている…… Parse メニューから Parse を選択する．

認識されたら，テキストか楕円のどちらかを選択して動かしてみましょう．もう一方がいっしょに動きます．次に実行窓に白い楕円を二つ描いて，そのあとテキスト文字列「4」「5」をそれぞれの中心に描いてみましょう．「3」を描いたノードの時と同じように「4」「5」のノードが認識されると思います．テキスト文字列を先に描いてから楕円を描いても構いませんが，その場合，あとから楕円の重なりの順番を奥にしないと，テキスト文字列が見えません．

次に二つ目のルールを定義しましょう． Make New Production Rule して下さい．図 A.4のような図を描いて選択してからでもなにも選択しない状態でも構いません．このくら

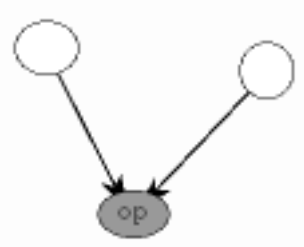


図 A.4: 演算子のノード

い複雑な図になるとなにも選択しない状態で Make New Production Rule した方がルールを作りやすいかもしれません.

CMG 入力部の各欄を次のようにしてから Ok ボタンを押して下さい.

欄	内容
name	node
attributes	point mid 0.4.mid integer value {script.integer {expr @0.2.value@@0.5.text@@0.3.value@}} integer left 0.2.left integer right 0.3.right
action	puts "value = @value@"
constraints	vp_close 0.0.start 0.3.mid vp_close 0.0.end 0.4.mid vp_close 0.1.start 0.2.mid vp_close 0.1.end 0.4.mid vp_close 0.4.mid 0.5.mid eq 0.4.innercolor {string green} lt 0.2.right 0.3.left
normal	line line node node circle text

constraints 欄の「eq 0.4.innercolor {string green}」は，楕円の塗りつぶす色が緑色であることを意味します．これは一つ目のルールの楕円の塗りつぶす色が白であるという制約に対応します．もしもこのようにしてそれぞれを区別しないとしたら，演算子を示すための楕円とテキスト文字列も数字を示すためのノードとして一つ目のルールによって認識されてしまいます．こうなると全体を認識することができません．ここでは塗りつぶす色を使って区別しましたが，例えば楕円と長方形にして区別したり，線の色で区別するという方法もあります.

これで二つ目のルールも定義できました．最初の例を実行窓に描いて認識させてみましょう．演算子を描く楕円を塗りつぶす色を緑色にするのを忘れないで下さい．うまく認識されれば標準出力には 35 という結果が表示されているはずです．数字のノードを動かしてみて下さい．一つのノードを動かすと，左右の位置関係が変わらないように他のノードも動きます.

なお，この例は，恵比寿がインストールされているディレクトリにあるexamples というディレクトリの下にあります．ファイル名は以下のとおりです．

- fvps.rule ……ルール
- fvps.can ……ルールの定義に使った図
- fvps_test.can ……例として実行に用いた図

exist を用いた場合

前の normal のみを用いた例では二つ目のルールにおいて演算子につながっている二つのノードと矢印も含めて一つの新しいノードとしていました．通常はノードは一つの円とその中のテキスト文字列としたいのではないかと思います．これを実現するためには， exist を使う必要があります．前の例での二つ目のルールを次のように書き換えて下さい． name 欄と action 欄は変更の必要はありません．

欄	内容
attributes	point mid 0.0.mid integer value {script.integer {expr @1.2.value@@0.1.text@@1.3.value@}} integer left 1.2.left integer right 1.3.right
constraints	vp_close 1.0.start 1.3.mid vp_close 1.0.end 0.0.mid vp_close 1.1.start 1.2.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.left
normal	circle text
exist	line line node node

変更が終了したら前の例と同じ図を実行窓に描いてみて下さい．前の例と同様の結果が得られると思います．できることは前の例とまったく同じです．違いを確認するためには図形単語のリストを表示させてみて下さい．演算子を表すノードの構成要素の部分が違っています．前の例ではすべて normal の構成要素となっていたのがこの例では exist のところにノード二つと矢印（直線）があります．

なお，この例のルールはディレクトリexamples の下のfvps2.rule というファイルにあります．

not exist を用いた場合

normal のみを用いた例と exist を用いた例では数字のノードと演算子のノードを外見上区別する必要がありました． not_exist を使うことでこれを避けることができます．これに

は演算子のノードには必ず矢印が入ってきて，数字のノードには矢印が入ってこないことを利用します． exist を用いる例におけるルール 1 を次のように変更して下さい．とくに書いていないところは変更は必要ありません．

欄	内容
constraints	vp_close 0.0.mid 0.1.mid vp_close 0.0.mid 2.0.end
not_exist	line

次に， exist を用いる例におけるルール 2 を次のように変更して下さい．とくに書いていないところは変更は必要ありません．

欄	内容
constraints	vp_close 1.0.start 1.3.mid vp_close 1.0.end 0.0.mid vp_close 1.1.start 1.2.mid vp_close 1.1.end 0.0.mid vp_close 0.0.mid 0.1.mid lt 1.2.right 1.3.left

変更が終了したら，前と同じ図で，数字のノードと演算子のノードの色を同じにしたものを描いてみて下さい．色はどんな色でもいいですし，本当は同じである必要さえもありません．今までと同様の結果が得られると思います．

なお，この例は，ディレクトリexamples の下にあります．ファイル名は以下のとおりです．

- fvps3.rule ……ルール
- fvps_test2.can ……例として実行に用いた図

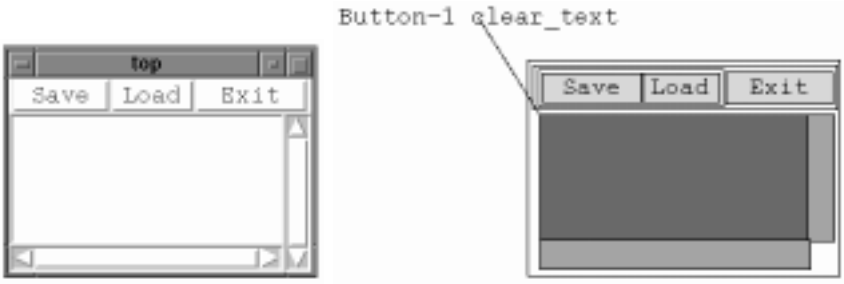
図の描き換え

これまでアクションでは値を表示させることしかしませんでしたが，ここでは描いた図そのものを描き換えていくようにします． not_exist を使う例のアクションを次のように変更して下さい．

欄	内容
action	puts "value = @value@" delete {@1.0@ @1.1@ @1.2@ @1.3@} alter @0.1@ text @value@

演算子を表すノードが認識されると， 2 行目の delete によってそのノードに入ってくる直線二つとノード二つを削除します．そのあと演算子が書かれていたテキスト文字列を，そのノードが持つ値に描き換えます．

なお，この例のルールはディレクトリexamples の下のfvps4.rule というファイルにあります．



(a) (b)

表 A.3: GUI の例 (a) とそれを表すビジュアルプログラム (b)

A.5.2 応用編

概要

ここでは前の例よりもっと大きな，もう少し実用的な例として GUI を作るビジュアル言語について述べます． GUI の構成要素としては，次のものを考えます．

名前	説明	言語での表現
フレームウィジェット	単体で使うとただの枠．この中に二つのウィジェットを並べることでウィジェットの階層を作る．	中が透明の長方形
テキストウィジェット	この中でテキスト文字列の編集ができる．	中が赤い長方形
ボタンウィジェット	この中に書かれた文字列で表されるコマンドを呼び出す．	中が黄色い長方形と，その中に書かれたテキスト文字列
スクロールバーウィジェット	テキストウィジェットの右側または下に付いてテキストのスクロールをコントロールする．	オレンジ色の長方形
バインディング	一つのウィジェットの動作（例えばマウスの左ボタンがクリックされたらテキストウィジェットの中身を消す，など）を規定する．	テキスト文字列と，それからウィジェットに繋がる直線

例えば図 A.3(a) に示すような GUI を生成するためには図 A.3(b) にあるようなものを描きます．

各ウィジェットの定義

ここではフレームウィジェット，ボタンウィジェット，スクロールバーウィジェット，テキストウィジェットを定義します．

まずはフレームウィジェットです．次のようなルール（今後ルール 0 と呼びます）を作って下さい．

欄	内容
name	Frame
attributes	integer width 0.0.width integer height 0.0.height point mid 0.0.mid integer lu_x 0.0.lu_x integer lu_y 0.0.lu_y integer rl_x 0.0.rl_x integer rl_y 0.0.rl_y string element {script.string {set dummy "frame @0.0.width@ @0.0.height@"}}
constraints	eq 0.0.innercolor {string {}}
normal	rectangle

attributes の欄は上から順に，幅，高さ，中心の座標，左上の点の x 座標，同 y 座標，右下の点の x 座標，同 y 座標，それが何かを表す文字列，を表します．これらの属性は後で必要になってきます． constraints の欄は中が塗りつぶされていないことを意味します．

次にボタンウィジェットです．ルール 0 をコピーしてルール 1 を作って下さい．コピーしたらルール 1 を次のように変更して下さい．

欄	内容
name	Button
attributes	integer width 0.0.width integer height 0.0.height point mid 0.0.mid integer lu_x 0.0.lu_x integer lu_y 0.0.lu_y integer rl_x 0.0.rl_x integer rl_y 0.0.rl_y string element {script.string {set dummy "button @0.0.width@ @0.0.height@ [list @0.1.text@]"}}
constraints	vp_close 0.0.mid 0.1.mid eq 0.0.innercolor {string yellow}
normal	rectangle text

constraints 欄で，長方形とテキスト文字列の中心が一致していることと長方形の中の色が黄色であることを要求しています．

次にスクロールバーウィジェットです．ルール 0 をコピーしてルール 2 を作って下さい．コピーしたらルール 2 を次のように変更して下さい．名前が書かれていない欄は変更は必要ありません．

欄	内容
name	Scroll
attributes	integer width 0.0.width integer height 0.0.height point mid 0.0.mid integer lu_x 0.0.lu_x integer lu_y 0.0.lu_y integer rl_x 0.0.rl_x integer rl_y 0.0.rl_y
constraints	eq 0.0.innercolor {string orange}

属性element がなくなったのは，スクロールバーが必ずテキストウィジェットと組になって使われることと関係しています．

次にテキストウィジェットです．ルール 0 をコピーしてルール 3 を作って下さい．コピーしたらルール 3 を次のように変更して下さい．名前が書かれていない欄は変更は必要ありません．

欄	内容
name	TextW
attributes	integer width 0.0.width integer height 0.0.height point mid 0.0.mid integer lu_x 0.0.lu_x integer lu_y 0.0.lu_y integer rl_x 0.0.rl_x integer rl_y 0.0.rl_y string element {script.string {set dummy "text @0.0.width@ @0.0.height@"}}
constraints	eq 0.0.innercolor {string red}

なお，ここまでのルールはディレクトリexamples の下のgui1.rule というファイルにあります．

実際にウィジェットを表示させる（1）

前の例ではウィジェットの定義をしましたが，ただ認識させるだけで実際にウィジェットが表示されるわけではありませんでした．実際に表示させるために，フレームウィジェット，ボタンウィジェット，テキストウィジェット（スクロールバーウィジェットはここでは扱いません．）の action の欄に動作を記述していきます．これには二通りの方法があります．一つは直接ウィジェットを生成するコマンドを書く方法で，もう一つは初期化ファイルに手続きを準備して，属性element を用いてウィジェットを生成する方法です．

まずは直接ウィジェットを生成するコマンドを書く方法について述べます．各ウィジェットの定義で作ったそれぞれのルールの action 欄を次のように変更して下さい．これらをそれぞれ 0’， 1’， 3’ と呼ぶことにします．

ルールの Id	内容
0	if {[winfo exists .top]} {destroy .top} toplevel .top frame .top.frame -width @width@ -height @height@ pack .top.frame
1	if {[winfo exists .top]} {destroy .top} toplevel .top set width [expr round(@width@/11)] set height [expr round(@height@/18)] set font "-*-*-*-*-*--18-*-*-*-m-110-*-*-" button .top.button -width $width -height $height \ -text @0.1.text@ -font $font -padx 0 -pady 0 pack .top.button
3	if {[winfo exists .top]} {destroy .top} toplevel .top set width [expr round(@width@/11)] set height [expr round(@height@/18)] set font "-*-*-*-*-*--18-*-*-*-m-110-*-*-" text .top.text -width $width -height $height -font $font pack .top.text

それぞれのルールの最初では土台となるトップレベルウィジェットがあればそれを消してから新しいトップレベルウィジェットを生成しています．フレームウィジェットでは属性width

とheight をそのまま用いてフレームウィジェットを生成します．ボタンウィジェットとテキストウィジェットでは幅と高さは文字数で指定しますので等幅フォントの幅と高さを使って文字数での幅と高さを計算し，それを用いてウィジェットを生成します．

今度は図形単語を描くとその種類に応じてその大きさのウィジェットが表示されるはずです．

なお，この例のルールは，ディレクトリexamples の下のgui2-1.rule というファイルにあります．

実際にウィジェットを表示させる（2）

次に初期化ファイルに手続きを準備して，属性element を用いてウィジェットを生成する方法です．まず恵比寿を起動する前に次のような五つの手続きを書いた初期化手続きのファイルを準備します．拡張子は.tcl にしてください．

```
proc create_gui {gui_list} {

    if {[winfo exists .top]} {
        destroy .top
    }

    toplevel .top

    pack [add_gui .top $gui_list]
}


proc add_gui {parent child} {

    set type [lindex $child 0]
    if {$type == "text"} {
        set name [create_text $parent $child]
    } elseif {$type == "button"} {
        set name [create_button $parent $child]
    } else {
        set name [create_frame $parent $child]
    }
    return $name
}


proc create_button {parent button} {

    set font "-*-*-*-*-*--18-*-*-*-m-110-*-*-"
    set width [expr round([lindex $button 1]/11)]
    set height [expr round([lindex $button 2]/18)]
    set text [lindex $button 3]
    set name $parent.button
    button $name -width $width -height $height -text $text \
            -font $font -padx 0 -pady 0
    return $name
```

```
}

proc create_frame {parent frame} {

    set width [lindex $frame 1]
    set height [lindex $frame 2]
    set name $parent.frame
    frame $name -width $width -height $height
    return $name
}

proc create_text {parent text} {

    set width [expr round([lindex $text 1]/11)]
    set height [expr round([lindex $text 2]/18)]
    set font "-*-*-*-*-*--18-*-*-*-m-110-*-*-"
    set name $parent.text
    text $name -width $width -height $height -font $font
    return $name
}
```

手続きcreate_gui はadd_gui を与えられたリストを使って呼び出して GUI を表示させます．手続きadd_gui は与えられたリストを使ってそれぞれの種類に応じたウィジェットを作ります．create_button，create_frame，create_text はそれぞれボタンウィジェット，フレームウィジェット，テキストウィジェットを作ります．

準備したら恵比寿を起動し，初期化手続きのファイル名をさっき作ったファイルにし，初期化手続きをおこなうように設定して下さい．次に，各ウィジェットの定義で作ったルール 0，1，3 の action 欄を次のように変更して下さい．この変更によってできたルールをそれぞれ 0”，1”，3” と呼ぶことにします．

```
create_gui @element@
```

実行窓に図形を描くと前と同じようにウィジェットが表示されていると思います．初期化手続きのファイルが少々面倒ですが今後の階層化のことなどを考えるとこの定義の方が便利です．

なお，この例は，ディレクトリexamples の下にあります．ファイル名は以下の通りです．

- gui2-2.rule ……ルール
- gui1.tcl ……初期化手続き

テキストウィジェットの定義

前の例ではフレームウィジェット，ボタンウィジェット，テキストウィジェットを実際に表示させられるようにしました．ここではテキストウィジェットの定義を変更して，スク

ロールバー付きのテキストウィジェットを表示できるようにします．水平のスクロールバーと垂直のスクロールバーがありますので，スクロールバーなし，垂直スクロールバー付き，水平スクロールバー付き，両方のスクロールバー付き，の 4 種類に分けて定義します．

まずはルール 3” を次のように変更します．何も書かれていない欄は変更の必要はありません．これをルール $3^{(3)}$ と呼ぶことにします．

欄	内容
attributes	integer width 0.0.width integer height 0.0.height point mid 0.0.mid integer lu_x 0.0.lu_x integer lu_y 0.0.lu_y integer rl_x 0.0.rl_x integer rl_y 0.0.rl_y string element {script.string {set dummy "text @0.0.width@ @0.0.height@ 0 0"}}
constraints	eq 0.0.innercolor {string red} vp_close 0.0.rl_x 2.0.lu_x vp_close 0.0.lu_y 2.0.lu_y vp_close 0.0.height 2.0.height vp_close 0.0.lu_x 2.1.lu_x vp_close 0.0.rl_y 2.1.lu_y vp_close 0.0.width 2.1.width
not_exist	Scroll Scroll

attributes の欄では属性`element` の最後に垂直，水平スクロールバーがついているかどうかを示す部分をつけて，それぞれ 0 としています． constraints の欄では not_exist に書いた二つの Scroll を使ったネガティブ制約（ここでは六つの vp_close）を書いています．vp_close の上から三つはテキストウィジェットを表す赤い長方形のすぐ右側に同じくらいの高さのスクロールバーウィジェットを表す Scroll がないことを要求しています．その次の三つは赤い長方形のすぐ下に同じくらいの幅の Scroll がないことを要求しています．これでスクロールバーなしのテキストウィジェットが定義できました．

次にルール $3^{(3)}$ をコピーしてルール 4 を作り，ルール 4 を次のように変更します．名前が書かれていない欄は変更の必要はありません．

欄	内容
attributes	integer width 0.0.width integer height 0.0.height point mid 0.0.mid integer lu_x 0.0.lu_x integer lu_y 0.0.lu_y integer rl_x 0.0.rl_x integer rl_y 0.0.rl_y string element {script.string {set dummy "text @0.0.width@ @0.0.height@ 1 0"}}
constraints	eq 0.0.innercolor {string red} vp_close 0.0.lu_x 0.1.lu_x vp_close 0.0.rl_y 0.1.lu_y vp_close 0.0.width 0.1.width vp_close 0.0.rl_x 2.0.lu_x vp_close 0.0.lu_y 2.0.lu_y vp_close 0.0.height 2.0.height
normal	rectangle Scroll
not_exist	Scroll

$3^{(3)}$ との違いはelement の最後から二番目の値が 0 から 1 に変わっていることと， not_exist にあった Scroll が一つ normal に移っていること（それにともない constraints の欄の変数の番号も変わっています）です．このルールによって水平スクロールバー付きのテキストウィジェットが定義できました．

次にルール 4 をコピーしてルール 5 を作り，ルール 5 を次のように変更します．名前が書かれていない欄は変更の必要はありません．

欄	内容
attributes	integer width 0.0.width integer height 0.0.height point mid 0.0.mid integer lu_x 0.0.lu_x integer lu_y 0.0.lu_y integer rl_x 0.0.rl_x integer rl_y 0.0.rl_y string element {script.string {set dummy "text @0.0.width@ @0.0.height@ 0 1"}}
constraints	eq 0.0.innercolor {string red} vp_close 0.0.rl_x 0.1.lu_x vp_close 0.0.lu_y 0.1.lu_y vp_close 0.0.height 0.1.height vp_close 0.0.lu_x 2.0.lu_x vp_close 0.0.rl_y 2.0.lu_y vp_close 0.0.width 2.0.width

4 との違いはelement の最後と最後から二番目の値の 0 と 1 が入れ替わっていることと，not_exist の Scroll が入れ替わっていること（見ためは同じですが， constraints の欄の変数に気をつけて下さい）です．これで垂直スクロールバー付きのテキストウィジェットが定義できました

次にルール 5 をコピーしてルール 6 を作り，ルール 6 を次のように変更します．名前が書かれていない欄は変更の必要はありません．

欄	内容
attributes	integer width 0.0.width integer height 0.0.height point mid 0.0.mid integer lu_x 0.0.lu_x integer lu_y 0.0.lu_y integer rl_x 0.0.rl_x integer rl_y 0.0.rl_y string element {script.string {set dummy "text @0.0.width@ @0.0.height@ 1 1"}}
constraints	eq 0.0.innercolor {string red} vp_close 0.0.rl_x 0.1.lu_x vp_close 0.0.lu_y 0.1.lu_y vp_close 0.0.height 0.1.height vp_close 0.0.lu_x 0.2.lu_x vp_close 0.0.rl_y 0.2.lu_y vp_close 0.0.width 0.2.width
normal	rectangle Scroll Scroll
not_exist	

5 との違いは element の最後から二番目の値が 1 に変わっていることと，not_exist にあった Scroll が normal に移っていること（それにともない constraints の欄の変数の番号も変わっています）です．これで垂直，水平両方のスクロールバーがついたテキストウィジェットが定義できました．

次に初期化手続きのファイル中の手続き create_text を次のように変更して下さい．

```
proc create_text {parent text} {

    set width [expr round([lindex $text 1]/11)]
    set height [expr round([lindex $text 2]/18)]
    set font "-*-*-*-*-*--18-*-*-*-m-110-*-*-"
    set hor [lindex $text 3]
    set ver [lindex $text 4]
    if {$hor == 0} {
        if {$ver == 0} {
            set name $parent.[new_value textW]
            text $name -width $width -height $height -font $font
            return $name
        } else {
            set name_f $parent.frame
            frame $name_f
            set name_t $name_f.textW
            set name_s $name_f.scroll
            text $name_t -width $width -height $height -font $font \
                    -yscrollcommand "$name_s set"
            scrollbar $name_s -command "$name_t yview" -orient vertical
            pack $name_t $name_s -side left -fill y
            return $name_f
        }
    } else {
        if {$ver == 0} {
```

```
            set name_f $parent.frame
            frame $name_f
            set name_t $name_f.textW
            set name_s $name_f.scroll
            text $name_t -width $width -height $height -font $font \
                    -xscrollcommand "$name_s set"
            scrollbar $name_s -command "$name_t xview" -orient horizontal
            pack $name_t $name_s -side top -fill x
            return $name_f
        } else {
            set name_f1 $parent.frame1
            frame $name_f1
            set name_f2 $name_f1.frame2
            frame $name_f2
            set name_t $name_f2.textW
            set name_s1 $name_f2.scroll1
            set name_s2 $name_f1.scroll2
            text $name_t -width $width -height $height -font $font \
                    -xscrollcommand "$name_s1 set" -yscrollcommand "$name_s2 set"
            scrollbar $name_s1 -command "$name_t xview" -orient horizontal
            pack $name_t $name_s1 -side top -fill x
            scrollbar $name_s2 -command "$name_t yview" -orient vertical
            pack $name_f2 $name_s2 -side left -fill y
            return $name_f1
        }
    }
}
```

これらの変更によって，スクロールバー付きのテキストウィジェットを生成できるようになりました．実際に実行窓に図形を描いてみて下さい．スクロールバーウィジェットを表すオレンジ色の長方形を描いたり消したりするたびに生成されるテキストウィジェットにスクロールバーが付いたり消えたりするはずです．

なお，この例は，ディレクトリexamples の下にあります．ファイル名は以下の通りです．

- gui3.rule ……ルール
- gui2.tcl ……初期化手続き

階層化（1）

これまでの例で個々のウィジェットを表示できるようになりました．実際の GUI の作成ではこれらのウィジェットを組み合わせて使用します．ウィジェットを並べるだけでも複数個のウィジェットの表示はできますが，配置などのきめの細かい設定をするためにはウィジェットの階層化をする必要があります．階層化にはフレームウィジェットを用います．ルールの定義を簡単にするために一つの階層には二つのウィジェットしか存在しないこととしま

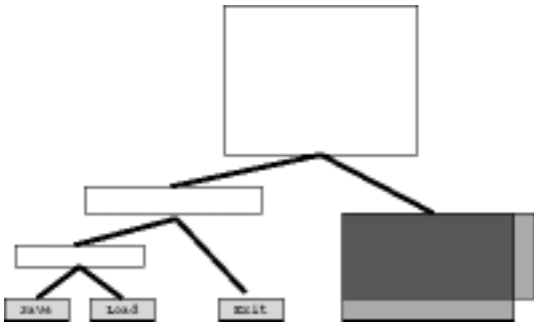


図 A.5: 図 A.3の階層構造

す．たとえば図 A.3は図 A.5のような階層構造を持っています．テキストウィジェットも本来は二つのスクロールバーとは別ですが，ここでは仮想的にこれで一つのウィジェットとして扱います．

ルールとしては次の五つを考える必要があります．これらを表すルールをそれぞれルール 7， 8， 9， 10， 11 とします．

- フレームウィジェットは GUI である．
- ボタンウィジェットは GUI である．
- テキストウィジェットは GUI である．
- GUI は一つのフレームウィジェットの中に二つの GUI が横にならんでいる．
- GUI は一つのフレームウィジェットの中に二つの GUI が縦にならんでいる．

まずはルール 7 を定義します．次のようなルールを書いて下さい．

欄	内容
name	GUI
attributes	integer width 0.0.width integer height 0.0.height point mid 0.0.mid integer lu_x 0.0.lu_x integer lu_y 0.0.lu_y integer rl_x 0.0.rl_x integer rl_y 0.0.rl_y string element 0.0.element
constraints	lt 0.0.lu_x 2.0.lu_x lt 2.0.rl_x 0.0.rl_x lt 0.0.lu_y 2.0.lu_y lt 2.0.rl_y 0.0.rl_y
normal	Frame
not_exist	rectangle

not_exist の欄に`rectangle` があるのは，フレームウィジェットが GUI として認識されるためにはそれが内部に GUI を持って階層を作るためのものではなく，ただの枠としての，ひとつのウィジェットであることを示すためです．

次にルール 8 を定義します．ルール 7 をコピーして次のように変更して下さい．名前が書かれていない欄は変更は必要ありません．

欄	内容
constraints	
normal	`Button`
not_exist	

次にルール 9 を定義します．ルール 8 をコピーして次のように変更して下さい．名前が書かれていない欄は変更は必要ありません．

欄	内容
normal	`TextW`

階層化（2）

ここまでの三つのルール（7， 8， 9）は今まで別々に扱われていたウィジェットを一種類にまとめるものでした．次の二つのルール（10， 11）はこれらを使って階層を作るものです．

まずルール 10 を定義します．次のようなルールを書いて下さい．

欄	内容
name	GUI
attributes	integer width 0.2.width integer height 0.2.height point mid 0.0.mid integer lu_x 0.2.lu_x integer lu_y 0.2.lu_y integer rl_x 0.2.rl_x integer rl_y 0.2.rl_y string element {script.string {make_gui_list @0.0@ @0.1@}} string id {script.string {new_value GUI}}
constraints	lt 0.2.lu_x 0.0.lu_x lt 0.0.rl_x 0.1.lu_x lt 0.1.rl_x 0.2.rl_x lt 0.2.lu_y 0.0.lu_y lt 0.2.lu_y 0.1.lu_y lt 0.0.rl_y 0.2.rl_y lt 0.1.rl_y 0.2.rl_y lt 2.0.lu_x 0.0.lu_x lt 0.0.rl_x 2.0.rl_x lt 2.0.lu_y 0.0.lu_y lt 0.0.rl_y 2.0.rl_y lt 0.2.lu_x 2.0.lu_x lt 2.0.rl_x 0.2.rl_x lt 0.2.lu_y 2.0.lu_y lt 2.0.rl_y 0.2.rl_y lt 2.1.lu_x 0.1.lu_x lt 0.1.rl_x 2.1.rl_x lt 2.1.lu_y 0.1.lu_y lt 0.1.rl_y 2.1.rl_y lt 0.2.lu_x 2.1.lu_x lt 2.1.rl_x 0.2.rl_x lt 0.2.lu_y 2.1.lu_y lt 2.1.rl_y 0.2.rl_y
normal	GUI GUI Frame
not_exist	Frame Frame

not_exist の欄の二つのFrame は， normal で使われているFrame が二つのGUI を囲む最小のFrame であることを表しています． attributes 欄の手続きmake_gui_list は，二つのGUI が横にならんでいるか縦にならんでいるかを判別して適切なリストを作るものです．

次にルール 11 を定義します．ルール 10 をコピーして次のように変更して下さい．名前が書かれていない欄は変更は必要ありません．

欄	内容
constraints	lt 0.2.lu_y 0.0.lu_y lt 0.0.rl_y 0.1.lu_y lt 0.1.rl_y 0.2.rl_y lt 0.2.lu_x 0.0.lu_x lt 0.2.lu_x 0.1.lu_x lt 0.0.rl_x 0.2.rl_x lt 0.1.rl_x 0.2.rl_x lt 2.0.lu_y 0.0.lu_y lt 0.0.rl_y 2.0.rl_y lt 2.0.lu_x 0.0.lu_x lt 0.0.rl_x 2.0.rl_x lt 0.2.lu_y 2.0.lu_y lt 2.0.rl_y 0.2.rl_y lt 0.2.lu_x 2.0.lu_x lt 2.0.rl_x 0.2.rl_x lt 2.1.lu_y 0.1.lu_y lt 0.1.rl_y 2.1.rl_y lt 2.1.lu_x 0.1.lu_x lt 0.1.rl_x 2.1.rl_x lt 0.2.lu_y 2.1.lu_y lt 2.1.rl_y 0.2.rl_y lt 0.2.lu_x 2.1.lu_x lt 2.1.rl_x 0.2.rl_x

基本的にはルール 10 と同じですが，縦にならぶか横にならぶかの違いが constraints の欄に表れています．

次にルール 0”， 1”， $3^{(3)}$， 4， 5， 6 の action 欄に書かれた手続きをすべて消して下さい．これらをルール 0， 1， 3， 4’， 5’， 6’ と呼ぶことにします．

次に，階層化された GUI を表示させるためのルールとして新たにルール 12 を作ります．次のルールを書いて下さい．

欄	内容
name	create
action	create_gui @0.0.element@
normal	GUI circle

これは GUI をすべて描いた後にそれを表示させるために楕円を一つ描くことで実際に表示させる手続きを呼び出すタイミングをユーザが与えるためのルールです．

次に初期化手続きのファイルを変更します．変更のポイントは次の点です．

- 手続き`make_gui_list` を作る．
- 手続き`add_gui` を階層に対応したものにする．
- 各ウィジェットを生成する手続きの中で，ウィジェットのパス名にそのウィジェットの Id を付ける．
- カウンタを生成する．

まずは手続きmake_gui_list を作ります.

```
proc make_gui_list {id1 id2} {

    global D

    set el1 [sb_var value $D($id1.element)]
    set el2 [sb_var value $D($id2.element)]
    if {[sb_var value $D($id1.rl_x)] <= [sb_var value $D($id2.lu_x)]} {
        return [list gui $el1 $el2 horizontal]
    } else {
        return [list gui $el1 $el2 vertical]
    }
}
```

二つのGUI を表す図形単語の Id を受け取り，座標を調べて横にならんでいれば,

```
gui GUI1 の要素のリスト GUI2 の要素のリスト horizontal
```

というリストを，そうでなければ,

```
gui GUI1 の要素のリスト GUI2 の要素のリスト vertical
```

というリストを返します.

次に手続きadd_gui を次のように変更します.

```
proc add_gui {parent child} {

    if {[lindex $child 0] == "gui"} {
        set name $parent.[new_value frame]
        frame $name
        set id1 [add_gui $name [lindex $child 1]]
        set id2 [add_gui $name [lindex $child 2]]
        if {[lindex $child 3] == "horizontal"} {
            pack $id1 $id2 -side left
        } else {
            pack $id1 $id2 -side top
        }
        return $name
    } else {
        set type [lindex $child 0]
        if {$type == "text"} {
            set name [create_text $parent $child]
        } elseif {$type == "button"} {
            set name [create_button $parent $child]
        } else {
            set name [create_frame $parent $child]
        }
        return $name
    }
}
```

これまでの部分は一番外側の if 文の else 以降に入っています．その前の部分が階層を処理する部分です．make_gui_list で作られたリストを元に，horizontal だったら横ならびに配置し，そうでなければ縦ならびに配置します．

次に各ウィジェットを生成する手続きの中で，ウィジェットのパス名にそのウィジェットの Id を付けます．これはウィジェットのパス名が一意に決まるようにするために必要になります．

```
proc create_button {parent button} {

    set font "-*-*-*-*-*--18-*-*-*-m-110-*-*-"
    set width [expr round([lindex $button 1]/11)]
    set height [expr round([lindex $button 2]/18)]
    set text [lindex $button 3]
    set name $parent.[new_value button]
    button $name -width $width -height $height -text $text \
            -font $font -padx 0 -pady 0
    return $name
}


proc create_frame {parent frame} {

    set width [lindex $frame 1]
    set height [lindex $frame 2]
    set name $parent.[new_value frame]
    frame $name -width $width -height $height
    return $name
}


proc create_text {parent text} {

    set width [expr round([lindex $text 1]/11)]
    set height [expr round([lindex $text 2]/18)]
    set font "-*-*-*-*-*--18-*-*-*-m-110-*-*-"
    set hor [lindex $text 3]
    set ver [lindex $text 4]
    if {$hor == 0} {
        if {$ver == 0} {
            set name $parent.[new_value textW]
            text $name -width $width -height $height -font $font
            return $name
        } else {
            set name_f $parent.[new_value frame]
            frame $name_f
            set name_t $name_f.[new_value textW]
            set name_s $name_f.[new_value scroll]
            text $name_t -width $width -height $height -font $font \
                    -yscrollcommand "$name_s set"
            scrollbar $name_s -command "$name_t yview" -orient vertical
            pack $name_t $name_s -side left -fill y
            return $name_f
```

```
        }
    } else {
        if {$ver == 0} {
            set name_f $parent.[new_value frame]
            frame $name_f
            set name_t $name_f.[new_value textW]
            set name_s $name_f.[new_value scroll]
            text $name_t -width $width -height $height -font $font \
                    -xscrollcommand "$name_s set"
            scrollbar $name_s -command "$name_t xview" -orient horizontal
            pack $name_t $name_s -side top -fill x
            return $name_f
        } else {
            set name_f1 $parent.[new_value frame]
            frame $name_f1
            set name_f2 $name_f1.[new_value frame]
            frame $name_f2
            set name_t $name_f2.[new_value textW]
            set name_s1 $name_f2.[new_value scroll]
            set name_s2 $name_f1.[new_value scroll]
            text $name_t -width $width -height $height -font $font \
                    -xscrollcommand "$name_s1 set" -yscrollcommand "$name_s2 set"
            scrollbar $name_s1 -command "$name_t xview" -orient horizontal
            pack $name_t $name_s1 -side top -fill x

            scrollbar $name_s2 -command "$name_t yview" -orient vertical

            pack $name_f2 $name_s2 -side left -fill y
            return $name_f1
        }
    }
}
```

変わったのは各ウィジェットの名前を付けるところでカウンタを使って新しい名前を得ているところです．ここでカウンタを使うので，初期化手続きのファイルのいちばん最後に次の部分を付け加えます．

```
foreach type {frame scroll button textW} {
    create_counter $type
}
```

これらの変更によって階層化されたGUIを生成できるようになりました．最初の例に示したような図を描いてみて下さい．最後に画面のどこかに楕円を描くとその図によって示されるGUIが生成されるはずです．

なお，この例は，ディレクトリ`examples`の下にあります．ファイル名は以下の通りです．

- gui4.rule ……ルール
- gui3.tcl ……初期化手続き

- gui.can ……最初の例に示した図

バインディング

これまでの定義によって階層的なGUIを生成することができるようになりました．ここではさらにバインディングを定義することによってさまざまな反応を得られるようにします．また，この言語で作ったGUIを恵比寿を介さずにwishから立ち上げられるようにする方法を追加します．

まず次に示すルール13を作って下さい．

欄	内容
name	Binding
constraints	vp_close 1.0.lu_x 0.1.end_x vp_close 1.0.lu_y 0.1.end_y vp_close 0.0.mid 0.1.start
normal	text line
exist	GUI

これがバインディングを表します．

次に，ルール7，8，9を次のように変更して下さい．名前が書かれていない欄は変更は必要ありません．

欄	内容
attributes	integer width 0.0.width integer height 0.0.height point mid 0.0.mid integer lu_x 0.0.lu_x integer lu_y 0.0.lu_y integer rl_x 0.0.rl_x integer rl_y 0.0.rl_y string element {script.string {list [new_value GUI] @0.0.element@}} string id {script.string {get_value GUI}}

属性elementが変更され，属性idが追加されています．elementはGUIとしてのIdと元となるウィジェットのelementのリストになっています．

次に，ルール10，11を次のように変更して下さい．名前が書かれていない欄は変更は必要ありません．

欄	内容
attributes	integer width 0.2.width integer height 0.2.height point mid 0.0.mid integer lu_x 0.2.lu_x integer lu_y 0.2.lu_y integer rl_x 0.2.rl_x integer rl_y 0.2.rl_y string element {script.string {make_gui_list @0.0@ @0.1@}} string id {script.string {new_value GUI}}

これは属性id が付加されています．

次に，ルール 12 を次のように変更して下さい．名前が書かれていない欄は変更は必要ありません．

欄	内容
attributes	string binding 3.0.binding
action	create_gui2 [list @0.0.element@ @binding@]
all	Binding

attributes では all のBinding を使ってバインディングのリストを作っています． action の手続き名が変わったのは，恵比寿を介さずに wish から立ち上げるための手続き名の方をcreate_gui とするためです．引数はこれまでのようにelement だけではなく，binding も渡しています．

次に初期化手続きを書き換えます．変更のポイントは次の点です．

- 単体で使えるようにするためにカウンタを定義しているライブラリを読み込むようにする．
- 単体で使うための手続きcreate_gui を作る．
- バインディングをおこなうための手続きcreate_binding を作る．
- これまでのcreate_gui をcreate_gui2 とし，受け取ったリストをファイルに書き出すように変更する．
- 大域変数GUI_ID を準備し，各ウィジェットを定義する手続きの中でこの値を定める．
- カウンタGUI を作る．

まず，初期化手続きのファイルの最初に次の 1 行を追加します．

```
source $env(TCL_DIR)/counter.tcl
```

これはカウンタを定義しているライブラリの読み込みです．環境変数TCL_DIR が設定されている必要があります．

次に，単体で使うための手続きcreate_gui を作ります．

```
proc create_gui {filename} {

    set fd [open $filename r]
    gets $fd gui_list
    close $fd

    pack [add_gui {} [lindex $gui_list 0]]
    create_binding [lindex $gui_list 1]
}
```

これはファイル名を受け取ってそのファイル中で定義されているGUI（解析の時に作られたリストの形式です）を生成するものです.

次にバインディングを行なうための手続きcreate_bindingを作ります.

```
proc create_binding {binding} {

    global GUI_ID

    foreach b $binding {
        set bndng [lindex $b 1]
        set sequence [lindex $bndng 0]
        set script [lindex $bndng 1]
        bind $GUI_ID([lindex $b 0]) <$sequence> $script
    }
}
```

GUI_IDは配列で，そのインデックスで表されるウィジェットのパス名を記録しています.

次に今までcreate_guiと呼んでいた手続きを変更してcreate_gui2を作ります.

```
proc create_gui2 {gui_list} {

    set fd [open GUI_FILE w]
    puts $fd $gui_list
    close $fd

    if {[winfo exists .top]} {
        destroy .top
    }

    toplevel .top

    pack [add_gui .top [lindex $gui_list 0]]
    create_binding [lindex $gui_list 1]
}
```

最初にGUI_FILEという名前でファイルを開き，そこに受け取ったGUIを表すリストを書き込みます．add_guiの呼び出しでは受け取ったリストの一番目の要素（属性element）を渡し，そのあとcreate_bindingに二番目の要素（属性binding）を渡しています.

次に，各ウィジェットを生成する手続きの中で配列GUI_FILEにウィジェットのパス名を設定するようにします．この際，ボタンウィジェットでは押された時にボタンに書かれている文字列を実行する部分も追加しています.

```
proc create_button {parent button g_id} {

    global GUI_ID

    set font "-*-*-*-*-*--18-*-*-*-m-110-*-*-"
    set width [expr round([lindex $button 1]/11)]
```

```
    set height [expr round([lindex $button 2]/18)]
    set text [lindex $button 3]
    set name $parent.[new_value button]
    button $name -width $width -height $height -text $text \
            -command "eval $text" -font $font -padx 0 -pady 0
    set GUI_ID($g_id) $name
    return $name
}


proc create_frame {parent frame g_id} {

    global GUI_ID

    set width [lindex $frame 1]
    set height [lindex $frame 2]
    set name $parent.[new_value frame]
    frame $name -width $width -height $height
    set GUI_ID($g_id) $name
    return $name
}


proc create_text {parent text g_id} {

    global GUI_ID

    set width [expr round([lindex $text 1]/11)]
    set height [expr round([lindex $text 2]/18)]
    set font "-*-*-*-*-*-*--18-*-*-*-m-110-*-*-"
    set hor [lindex $text 3]
    set ver [lindex $text 4]
    if {$hor == 0} {
        if {$ver == 0} {
            set name $parent.[new_value textW]
            text $name -width $width -height $height -font $font
            set GUI_ID($g_id) $name
            return $name
        } else {
            set name_f $parent.[new_value frame]
            frame $name_f
            set name_t $name_f.[new_value textW]
            set name_s $name_f.[new_value scroll]
            text $name_t -width $width -height $height -font $font \
                    -yscrollcommand "$name_s set"
            scrollbar $name_s -command "$name_t yview" -orient vertical
            pack $name_t $name_s -side left -fill y
            set GUI_ID($g_id) $name_t
            return $name_f
        }
    } else {
        if {$ver == 0} {
            set name_f $parent.[new_value frame]
            frame $name_f
```

```
            set name_t $name_f.[new_value textW]
            set name_s $name_f.[new_value scroll]
            text $name_t -width $width -height $height -font $font \
                    -xscrollcommand "$name_s set"
            scrollbar $name_s -command "$name_t xview" -orient horizontal
            pack $name_t $name_s -side top -fill x
            set GUI_ID($g_id) $name_t
            return $name_f
        } else {
            set name_f1 $parent.[new_value frame]
            frame $name_f1
            set name_f2 $name_f1.[new_value frame]
            frame $name_f2
            set name_t $name_f2.[new_value textW]
            set name_s1 $name_f2.[new_value scroll]
            set name_s2 $name_f1.[new_value scroll]
            text $name_t -width $width -height $height -font $font \
                    -xscrollcommand "$name_s1 set" -yscrollcommand "$name_s2 set"
            scrollbar $name_s1 -command "$name_t xview" -orient horizontal
            pack $name_t $name_s1 -side top -fill x

            scrollbar $name_s2 -command "$name_t yview" -orient vertical

            pack $name_f2 $name_s2 -side left -fill y
            set GUI_ID($g_id) $name_t
            return $name_f1
        }
    }
}
```

最後のカウンタを作る部分ではGUI というカウンタも作るようにしています.

```
foreach type {frame scroll button textW GUI} {
    create_counter $type
}
```

これですべての定義ができました. ただ, 最初の例に示したような図を描いて Load, Save, Exit, clear_text などを実行できるようにするためにはこれらの手続きを定義しなくてはなりません. これらの手続きを例えば次のように定義してみましょう.

```
proc get_text_widget {parent} {
    global TextWidgetPath

    foreach child [winfo children $parent] {
        set class [winfo class $child]
        if {$class == "Text"} {
            set TextWidgetPath $child
            return
        } elseif {$class == "Frame"} {
            get_text_widget $child
```

```
        }
    }
}


proc Save {} {
    global TextWidgetPath

    if {![info exists TextWidgetPath]} {
        if {[winfo exists .top]} {
            get_text_widget .top
        } else {
            get_text_widget .
        }
    }

    set fd [open editor_data w]
    puts $fd [$TextWidgetPath get 0.0 end]
    close $fd
}

proc Load {} {
    global TextWidgetPath

    if {![info exists TextWidgetPath]} {
        if {[winfo exists .top]} {
            get_text_widget .top
        } else {
            get_text_widget .
        }
    }

    $TextWidgetPath delete 0.0 end
    set fd [open editor_data r]
    while {[gets $fd line] >= 0} {
        $TextWidgetPath insert end $line\n
    }
    close $fd
}

proc clear_text {} {
    global TextWidgetPath

    if {![info exists TextWidgetPath]} {
        if {[winfo exists .top]} {
            get_text_widget .top
        } else {
            get_text_widget .
        }
    }

    $TextWidgetPath delete 0.0 end
}

proc Exit {} {
```

```
    global TextWidgetPath

    if {[info exists TextWidgetPath]} {
        unset TextWidgetPath
    }
    if {[winfo exists .top]} {
        destroy .top
    } else {
        destroy .
    }
}
```

get_text_widget はこのアプリケーションの中にテキストウィジェットが一つしかないと仮定してそのテキストウィジェットのパス名を求める手続きです.

これらの手続きを初期化手続きのファイルの中に書き込むことで，実際この動作をするアプリケーションを作ることができます.

一度解析されるとファイルGUI_FILE が作られ，そこにこの GUI のデータが格納されます．このデータを使って恵比寿を介さずにこのアプリケーションを立ち上げることができます．方法は次の通りです.

1. wish を立ち上げる．原因はわかりませんがバージョン 8.0 でないと駄目なようです.
2. source gui.tcl を実行する．gui.tcl はこれまでに作った初期化手続きのファイル名です.
3. create_gui GUI_FILE を実行する.

なお，この例は，ディレクトリexamples の下にあります．ファイル名は以下の通りです.

- gui.rule ……ルール
- gui.tcl ……初期化手続き

A.5.3 GIF イメージを使う

ここではパイプの機能を備えたシェルの視覚化を例に GIF イメージを使う方法について簡単に述べます．繰返しや分岐などの制御構造については考えないこととします．コマンド（command）は 0 個以上の引数，またはオプション（argument）をとり，コマンドの出力をパイプでつなぐことによって次のコマンドの入力として使うことができます．command の視覚化表現は GIF イメージか，もしくは長方形の中に文字列が書かれたものとして定義します．argument の視覚化表現は楕円の中に文字列が書かれたものとして定義します．

この例はディレクトリexamples の下にあります．ファイル名は以下の通りです．

- vsh.rule ……ルール
- vsh.can ……例となる図
- images/names ……ImageFile

恵比寿の起動前に，設定ファイルのImageFile の行を次のように書き換えておいて下さい．

```
set ImageFile $env(VPG_DIR)/examples/images/names
```