筑波大学大学院博士課程

工学研究科修士論文

例示入力図を用いた Spatial Parser Generator

電子・情報工学専攻

著者氏名　藤山 健一郎
指導教官　電子・情報工学系 田中 二郎

平成 13 年 2 月

要旨

本論文では既存の Spatial Parser Generator が、２次元的な情報をもつ図形言語を１次元的なテキストで記述するために生じるいくつかの問題点を挙げ、これらの問題が、図形言語を図形を用いて直接定義するということで解決できる事について述べる。具体的には恵比寿という Spatial Parser Generator の定義方法について考察を行い、例示入力図という視覚的な表現を利用したインタラクティブな生成規則の編集を行う、より分かりやすい入力インターフェイスを提案する。また、提案インターフェイスを持つ Spatial Parser Generator VIC を作成し、実際に VIC を持ちいてビジュアルシステムを作成する例を示す。最後に評価として VIC の有効性を示すため、既に挙げたビジュアルシステムの作成課程を通して恵比寿と作業時間の比較実験を行う。

目次

1 序論 **3**

2 準備 **5**

2.1 用語の定義 5

2.2 Spatial Parser Generator 5

2.3 従来の生成規則の定義方法について 7

3 生成規則定義方法の考察 **10**

3.1 基本方針 10

3.2 構成要素 11

3.2.1 構成要素の入力 11

3.2.2 構成要素の種類の表示 11

3.2.3 構成要素の種類の変更 12

3.3 制約 13

3.3.1 制約の入力 14

3.3.2 制約の削除 16

3.3.3 制約の視覚化 16

3.4 属性 17

3.5 アクション 19

3.5.1 delete 20

3.5.2 alter 20

3.5.3 create 21

4 Spatial Parser Generator 「VIC」 **23**

4.1 システム VIC 23

4.2 実装 24

4.2.1 使用言語 24

4.2.2 制約解消系 24

4.2.3 各機能の詳細 24

5 ビジュアルシステムの作成例 **28**

5.1 データ構造１：スタック 28

5.1.1 生成規則１の定義 31
5.1.2 生成規則３の定義 32
5.1.3 生成規則２の定義 34
5.1.4 生成規則４の定義 36
5.2 データ構造２：二分木リスト 39
5.2.1 生成規則１の定義 40
5.2.2 生成規則２の定義 42
5.3 描き換えアルゴリズム：フラクタルの樹 45
5.3.1 生成規則の定義 46
5.4 アプリケーション：計算の木 48
5.4.1 生成規則１の定義 50
5.4.2 生成規則２の定義 52

6 評価 **57**
6.1 実験内容 57
6.1.1 実験方法 57
6.1.2 実験環境 57
6.1.3 被験者 57
6.1.4 学習効果への対応 58
6.2 結果 58
6.3 考察 59

7 関連研究 **61**

8 結論 **63**

謝辞 **64**

参考文献 **65**

第 1 章

序論

絵やアニメーションはテキストよりもより豊富な情報を表示でき、かつユーザはより容易にその情報を認識できる。コンピュータの普及に従い、今までコンピュータに馴染みのないエンドユーザもコンピュータを使うようになってきた。それに伴い「多くの情報を分かりやすく」ということはインターフェイスとして重要なポイントとなる。例えば新たなプログラミングとして、人間が理解しやすい図形などの視覚的な表現を用いたビジュアルプログラミング [1][2] などが注目を集めている。このように今後のソフトウェアはこうした視覚的でインタラクティブな要素を含むビジュアルシステムへと発展していくと考えられる。既にいくつかのビジュアルシステム [3][4][5][6][7][8] が提供されており、また研究されている。

ビジュアルシステムの研究では対象となるものが図形を用いた言語 — 図形言語であるため、これを実装する際必要となるのが図形言語の解析をおこなう Spatial Parser である。これはテキスト言語の処理系における字句解析器・構文解析器にあたる。通常、 Spatial Parser は特定の図形言語に特化した、すなわちある一定の図形の形状や配置のみを効率良くを解析する形で実現される。しかし図形言語の研究における試験的な実装の際には、試行錯誤を繰り返すため、様々な仕様変更が頻繁に起こるので、始めから単一の仕様に特化して Spatial Parser を作成するのは逆に効率が悪い。そこで、ある図形言語の図形の形状や配置などに関しての仕様を定義することにより、その仕様に対する Spatial Parser を自動的に生成する（テキスト言語における） lex や yacc のようなもの、すなわち Spatial Parser Generator が必要となってくる。

我々の研究室でも恵比寿 [9][10][11][12][13] という Spatial Parser Generator を研究してきた。恵比寿は拡張 CMG というメタ言語で図形言語の仕様を定義する。図形言語の仕様を定義するということは、すなわち図形間の関係を定義することであるが、これは扱う情報が 2 次元上の図形的なものであることを意味する。しかしながら従来の Spatial Parser Generator ではこれを 1 次元的なテキストで定義するため直感的に理解しにくいという欠点があった。またテキストで記述するため、定義するためのメタ文法を十分知っていなくては定義できない。そこで我々は図形言語の定義をテキストではなく、図形を用いて視覚的に、かつ直感的に行うことにより、これらの欠点を解消できると考えた。 [14]

本論文では例示入力図という視覚的な表現を利用したインタラクティブで、よりわ

かりやすい図形言語の仕様を定義する手法を提案する。提案手法では図形を描くことにより直接図形言語を定義する。このとき描かれる図形を例示入力図と呼ぶ。また提案した手法を実現したインターフェイスをもつ Spatial Parser Generator 「VIC」[15][16][17] を作成し、その評価について述べる。

本論文の構成は次のとおりである。まず第 2 章では本論文で用いる用語の定義や、Spatial Parser Generator についての基本的な知識とその問題点について述べる。第 3 章ではその問題点を解消すべく、図形言語の仕様の新しい定義方法についての考察を行う。第 4 章では考察に基づいたインターフェイスを実装した Spatial Parser Generator 「VIC」について述べ、次の第 5 章で「VIC」を用いて実際にビジュアルシステムを作成する例を示す。第 6 章では第 5 章で行ったビジュアルシステム作成の過程をとおして VIC の評価を行う。第 7 章では関連研究について述べ、最後に第 8 章で結論を述べる。

第 2 章

準備

2.1 用語の定義

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

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

本論文では図形言語を処理するシステムをビジュアルシステムと定義する。

2.2 Spatial Parser Generator

Spatial Parser Generator とは、ある図形言語の図形の形状や配置などに関しての仕様を定義することにより、その仕様に対する Spatial Parser を自動的に生成する（テキスト言語における） lex や yacc のようなものである。 Spatial Parser Generator として「恵比寿」 [9][10][11][12] が開発されている。恵比寿は図形言語の仕様を定義することで、 Spatial Parser を自動生成し、それを用いて描画された図形の意味を解析し、さらに処理を行うことができる。（図 2.1）

恵比寿ではメタ言語である拡張 CMG[9] を用いて図形言語の仕様を定義する。拡張 CMG とは、図形間の関係を記述する CMG[18][19] をベースに、生成規則が適用された時に図形の書き換えなどといった action も定義できるようにしたものである。これ

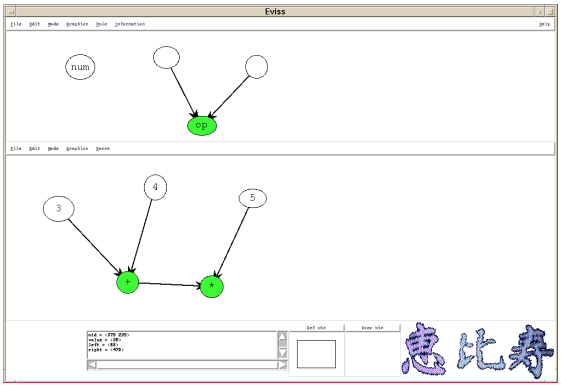

図 2.1: 恵比寿の実行画面

によって、図形言語の解析だけではなく、その結果を利用してなんらかの処理を行うことも可能としている。 CMG で定義した図形言語の仕様を生成規則という。

文献 [18] における CMG の生成規則は以下のようになる。

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

そして拡張 CMG の生成規則は以下のようになる。

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

ここで $T(\vec{x})$ は生成される図形単語の名前（name）である。また $T_1(\vec{x_1}), ..., T_n(\vec{x_n})$ は n 個の図形単語、もしくは図形文字であり、 normal の構成要素と呼ぶ。実際の非終端記号を構成する部品となる、すなわち書き換えされる対象となるものである。 $T'_1(\vec{x'_1}), ..., T'_m(\vec{x'_m})$ も m 個の図形単語、もしくは図形文字であり、 exist の構成要素と呼ぶ。図のどこかに存在する他のトークンと normal の構成要素との間になりたつ制約を記述するために用いられる。すなわち、このトークンが存在しなければ生成規則は適用されない。 $T''_1(\vec{x''_1}), ..., T''_l(\vec{x''_l})$ は l 個の図形単語、もしくは図形文字であり、 not_exist の構成要素と呼ぶ。 exist と同じく、生成規則の決定性を増すために用いられる。このトークンが存在するときは生成規則は適用されない。 $T'''_1(\vec{x'''_1}), ..., T'''_k(\vec{x'''_k})$ は k 個の図形単語、もしくは図形文字であり、 all の構成要素と呼ぶ。これは図形文の中にある指定された

種類の全てのトークンからなるマルチセットを作るために用いられる。つまり同一種類の複数のトークンを指定する際に使用する。 C は属性 $\vec{x_1}, \cdots, \vec{x_n}$、 $\vec{x'_1}, \cdots, \vec{x'_m}$、 $\vec{x'''_1}, \cdots, \vec{x'''_l}$ に関する制約の連接である。制約（constraints）とは、２つの属性間、もしくは１つの属性と定数の間になんらかの条件を課すことである。 F は属性 $\vec{x_1}, \cdots, \vec{x_n}$、 $\vec{x'_1}, \cdots, \vec{x'_m}$、 $\vec{x'''_1}, \cdots, \vec{x'''_l}$ を引数とする関数であり、生成される非終端記号 $T(\vec{x})$ の属性（attributes）$\vec{x}$ を定義している。 C と F に not_exist の属性 $\vec{x''_1}, \cdots, \vec{x''_l}$ が無いのは、生成規則が適用されるためには、 not_exist の構成要素が在ってはならないからである。 *Action* はアクション（action）を記述したものである。アクションとは「生成規則が適用されたときにスクリプト言語のプログラムとして実行される文字列」と定義される。たとえば値の計算、新たな図形の生成、変更、削除などが可能である。

2.3 従来の生成規則の定義方法について

恵比寿で図形言語の仕様、すなわち生成規則を定義するには、前述の拡張 CMG を用いて

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

を定義する必要がある。

具体的には以下のような手法を用いる。まず、図 2.2 のように図形を描くことによって大まかな文法と構成要素を与える。

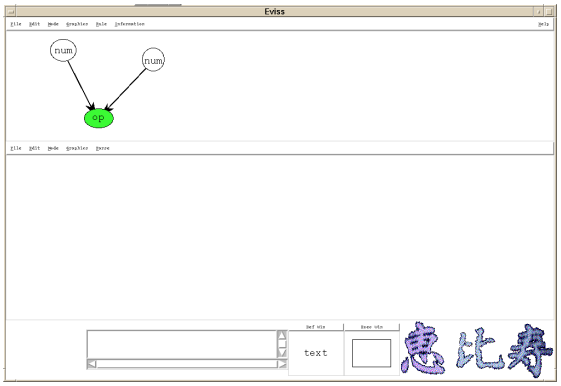

図 2.2: 恵比寿の CMG 入力法１

同時に恵比寿がその描画された図から、簡単な制約 — 図形が重なっている、等 — を自動生成する。このあと CMG 入力ウィンドウが開かれるが、ここには入力された構

成要素と自動生成された制約が既にテキストで書き出されている。（図 2.3）

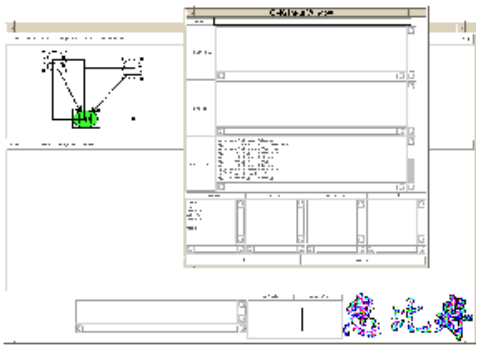

図 2.3: 恵比寿の CMG 入力法 2

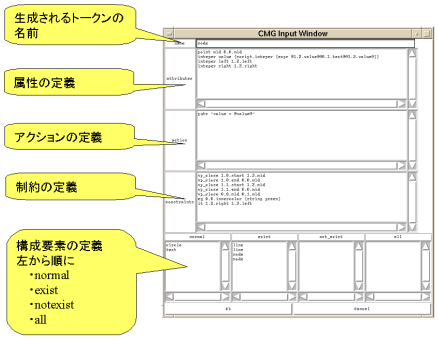

図 2.4: 恵比寿の CMG 入力法 3

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

しかし、この手法では、最初の入力で図形を利用するものの、その後の編集ではこの図が利用されず、結果としてほとんどテキストによって CMG を編集していた。 CMG は図形間の関係を記述するものであるということは既に述べたが、図形関係を扱うには１次元的なテキストの場合と異なり多次元（恵比寿では２次元）な位置関係を扱うので

テキストにくらべより多くのパラメータを指定する必要があるため、文法の記述が非常に複雑である。そもそも多次元的な情報を１次元的なテキストだけで編集することは、次元のギャップがあるため、ユーザの目的とそれを行うための操作に隔たりがあり、直感的にわかりにくく、ユーザに多大な負担をかけることとなる。

またユーザの負担を軽くするため、描画された図形を元に単純な生成規則をシステムが自動生成する機能があるが、実際にはユーザが望んでいる生成規則が生成されることはあまりない。さらにこの生成規則は構成要素の種類と順番に基づいた識別 ID を用いて記述されるが、この識別 ID は順序に基づいたもので、構成要素の種類を１つでも変更すると、生成規則の大部分が無意味となってしまう。結局白紙からテキストで書き起こすのとほとんど変わらない努力を要していた。

第 3 章

生成規則定義方法の考察

3.1 基本方針

従来の Spatial Parser Generator では 2 次元的な情報を持つ図形言語の仕様を 1 次元的なテキストで定義するため直感的に理解しにくいという欠点があった。またテキストで記述するため、定義するためのメタ文法を十分知っていなくては定義できない。そこで我々は図形言語の定義をテキストではなく、図形を用いて視覚的に、かつ直感的に行うことにより、これらの欠点を解消できると考えた [14]。例えば 1 本の線を定義する場合でも、テキストで定義する場合、線の太さ、色、始点の座標、終点の座標、といった非常に複雑な記述を行わなくてはならない。また実際に行いたいことと定義の操作がかけ離れているため、直感的に理解できない。一方図形で定義する場合、要求される線を 1 本実際に描画するだけで、難解なメタ文法に対する細かい知識がなくても、必要な情報を容易に定義できる。

この点に着目し、図形を描くことによりその図形の特徴を抽出、汎化することで直接図形言語を定義する手法を提案する。このとき最初に描かれる図形を例示入力図と呼ぶ。例示入力図はその名の通り、図形言語の構成要素を例示的に入力、すなわち描画したものである。以降の全ての定義も、基本的にこの例示入力図に対するマウス操作だけで行えるようにする。これによりユーザは煩わしいキーボード操作を最小限に留めることにより、ユーザへの余計な負担を少なく出来る。またテキスト記述を行わないので、メタ文法に対する厳密な知識 — 構文などを 1 文字の間違いもなく正確に知っていること — がなくても定義可能である。さらに例示入力図を参照しながら作業を行うことにより、ユーザは自分の作業が図形言語のどの部分を定義しているのか把握しやすく、かつ操作結果を視覚的にフィードバックすることが可能となる。

以下で生成規則定義に必要な

- 構成要素 （normal, exist, not_exist, all）
- 制約 （constraints）
- 属性 （attributes）
- アクション （action）

の順で従来の定義手法の問題点と提案する定義方法について説明する。

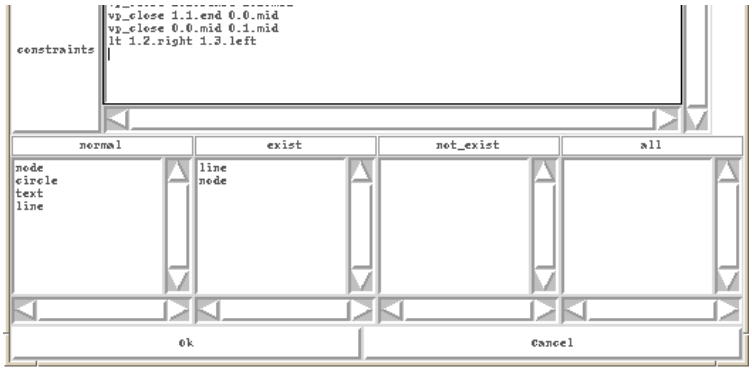

図 3.1: 構成要素表示部

3.2 構成要素

3.2.1 構成要素の入力

生成規則を定義する際、その構成要素となる部品についての情報がまず必要となる。どのようなタイプの構成要素が必要となるかは例示的に入力、すなわちキャンバスに実際に描画することによって指定する。ここまでは恵比寿も同じであるが、構成要素が終端記号だけでなく、他の生成規則によって定義された非終端記号である場合、問題が生じる。非終端記号は最終的には終端記号の集まりであるので、これを入力する際には終端記号を組み合わせて描画する。しかし恵比寿ではこのようにして新しく生成規則を定義しようとしても、全体を非終端記号と認識せず、終端記号と認識される。そのため、CMG 入力ウィンドウ上で終端記号を削除し改めて非終端記号に手動で書き直さなくてはならない。このような方法では、構成要素中に非終端記号が占める割合が増加するにつれ、その修正の手間も比例して増えるので、大規模なビジュアルシステムを構築することは困難となる。したがって、他の生成規則を適用して非終端記号を直接認識するべきである。描かれた図形を 1 つの意味のある非終端記号として認識するためには、他の生成規則を利用してその図形が一度解析される、すなわち Spatial Parsing が行われる必要がある。

またこれら構成要素について削除や、新規に追加することもやはり例示的に、すなわちキャンバス上に描いたり削除したりすることによって行うことができるべきである。

3.2.2 構成要素の種類の表示

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

しかし、このような表示法では構成要素間の図形的位置関係がわかりにくく、全体的な把握が困難である。たとえば図 3.1の例でいえば、 normal 欄、 exisit 欄にそれぞれ

図 3.2: 構成要素の表示法

「node」という構成要素があるが、例示入力図と見比べて、どちらの「node」が normal が exist 要素かわかりにくい。

そこでこのテキストによる表示法を改め、例示入力図を利用した視覚的な表現を考える。表示法には構成要素を指定するとその種類を表示するといった手法や、構成要素ごとに別々のキャンバスに表示するといった手法などが考えられるが、これらはいずれも図形的位置関係は把握できるが、一度に表示できる種類が限られているため、構成要素全体の把握が困難である。そこですべての構成要素の種類を同時に表示するための方法として、構成要素の種類ごとに色分けをして種類を明示し、一緒に表示する方法を考える。（図 3.2）

この方法ならばすべての構成要素の種類を同一キャンバスに同時に表示できるため、全体的な把握が容易である。また前述の 2 つの手法と同じく、図形間の位置関係を保存したまま表示するため直感的にわかりやすい。ただし、この表示法の場合、構成要素のもともとの色情報が失われてしまうという欠点がある。そこで「種類によって色分けで表示」「構成要素の本来の色で表示」の 2 種類のモードを瞬時に切り替えれるようにすべきである。

また、構成要素のタイプ（円、直線、 node、 etc...）については描かれた形状から視覚的に判別できるが、さらに構成要素にポインタを当てることにより、ティップスウィンドウなり、固定のメッセージウィンドウなりにそのタイプなどが表示されればより確実に識別ができると思われる。ティップスウィンドウは視点移動が少ないというヒューマンインターフェイス上の利点はあるが、構成要素上をポインタが動くたびにティップスウィンドウが次々と飛び出すのは煩わしい上、キャンバス上の情報を隠してしまうという欠点もある。そもそもここで表示される情報はあくまで補助的なものなので固定メッセージウィンドウ表示方式を採用する。

3.2.3 構成要素の種類の変更

入力された時点で構成要素の種類はすべて normal となっている。そこでこれを必要に応じ exist、 not_exist 等に変更する。恵比寿では構成要素表示部に書き出された構成要素の名前を編集することで変更していた。たとえば図 3.1の normal の構成要素の

「node」を not_exist に変更したい場合、まず normal 欄の 「node」 の名前を削除する。そのあと not_exist 欄に 「node」 と書きこむ。しかしこの方法では変更という本来一つの操作を、削除と追加という 2 つの操作で実現しているため操作と実体が一致していなく、ユーザの短期記憶に依存する非直感的な操作であり、構成要素の名前が長い場合、名前を間違える恐れもある。

またテキストで種類変更をする場合、 CMG の整合性の問題も無視できない。恵比寿では構成要素は構成要素表示部における位置によってナンバリングがされている。構成要素の識別 ID は kind.number という形式を取る。 kind は構成要素の種類で normal なら 0、 exist なら 1、 not_exist なら 3 となる。 number は 各構成要素欄における順序で 0 からはじまる整数である。図 3.1では, たとえば normal 欄の node の識別 ID は 1.0、 exist 欄の node の識別 ID は 1.1 となる。ここで問題となるのは number の部分が順序でつけられているため、ある欄からひとつ構成要素を削除してしまうと、その構成要素以降の構成要素すべての number がずれてしまうということである。図 3.1でいえば exist 欄の line（識別 ID 1.0）を削除するとそれ以降の構成要素である node の識別 ID は 1.1 から 1.0 となってしまう。恵比寿ではこの識別 ID で構成要素を特定して生成規則を記述しているため、種類を変更した構成要素だけでなく、それ以外の識別 ID もずれてしまうと、生成規則全体を書きなおす必要がでてくる。恵比寿ではこの整合性を保つのもすべてユーザに任せられている。大規模な生成規則にになるほどその整合性を保つのは人手ではむずかしくなる。したがってこのような識別 ID 管理という単調な作業ははシステム側に一括して任せるべきである。

さて、 3.2.2節で構成要素の種類の表示法について、例示入力図を利用したが、ここでもその図を利用する方法を用いる。その理由はやはり図を用いたほうが、その構成要素の図形間の位置関係を確認したまま変更作業が行えるからである。また色を用いて構成要素の種類を表示してあるということは、その種類を変更するとその結果として描かれた構成要素の色が変わるというフィードバックが得られるので、よりインタラクティブな作業ができると考えられるからである。以上のことを考慮すると 図 3.3 のような方法が良いと考えられる。まず変更したい構成要素にポインタをあてる。このときメッセージウィンドウにはこの構成要素のタイプが表示される。構成要素をクリックするとポップアップメニューが表示され、そこで構成要素の種類（normal、 exist、 not_exist、 all）を選択する。選択すると構成要素の色が選択された種類の色に変わる。といった具合である。なおこの際ユーザは構成要素の識別 ID の整合性を考慮しなくてもよい。直接例示入力図で指定するためユーザが識別 ID を意識することは無く、先ほど述べたように、内部的な識別 ID の管理はシステム側がすべて行ってくれるからである。

3.3 制約

生成規則の適用条件は、制約を満たす構成要素が存在することであり、この制約が生成規則の重要な役割を占めている。制約は構成要素の属性間の関係を示すのであるが、例示入力図から、制約となる構成要素間の関係をある程度推測することが可能である。

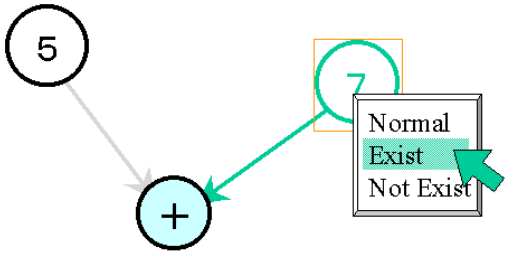

図 3.3: 構成要素の種類の変更

システム側がある程度推測して制約を自動生成することによって、ユーザの入力を簡単化することができる。自動生成機能によってある程度は制約が生成されるが、一般にこれだけでは生成規則を定義するのに十分とはいえない場合が多い。そこで生成された制約のうち不必要な制約を削除したり、足りない制約を追加したりして生成規則を完成させる。恵比寿では制約を追加、削除するには新たにテキストで記述するしか方法がなかった。しかし単純にテキストで記述する場合、新たに制約を追加するには恵比寿における CMG の文法について正確な知識を持ち合わせていないと記述できず、また文法を知っていても構文エラーを犯す場合がある。削除する場合にも、恵比寿の CMG 入力ウィンドウには Undo 機能がついていないので、誤って削除してしまった場合、書き直す以外に復元させることはできない。また 3.2.3節に述べたように識別 ID の整合を保つのも大変である。

そこでテキストによる記述以外の方法で制約の編集を行う方法を考える。

3.3.1 制約の入力

制約は

$$const\ Arg1\ Arg2$$

の形式で示すことができる。ここで const は制約の種類、 Arg1、 Arg2 は「構成要素（の識別 ID）. 属性名」、もしくは「{ 型　定数 }」である。つまり制約を定義するということは制約の種類と属性名もしくは定数を一組にして指定するということである。そこで、図 3.4のような制約定義ウィンドウを設け、その各フィールドに制約の種類、属性名、定数を何らかの方法で入力することによって制約を定義する。

まず制約の種類の入力方法について考える。制約の種類自体は多くない。恵比寿でも扱える制約は 8 個である。したがって選択肢の少ない制約の種類は制約種類入力フィールドにメニューを設け、そこから選ぶようにすれば良いと考えられる。

次に「構成要素. 属性名」の入力方法である。順番としては、まず構成要素を選んでから、その構成要素中にある属性名を選択するというのが自然であり、直感的に理解しやすい。そこでまず構成要素の指定法から考えるが、これは構成要素の種類を変更する際に構成要素を選択する手法と同じ状況である。したがってヒューマンインターフェイ

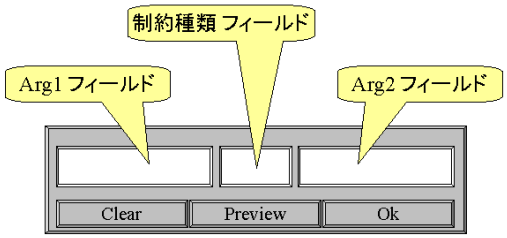

図 3.4: 制約定義ウィンドウ

スの観点から見て、統一的な手法を用いたほうが良いので、同様に例示入力図をポインタで指し示すことにより指定する方法が良いと思われる。そして属性名の選択であるが、一つの構成要素が持つ属性の個数は一般にさほど多くないので、リストメニューから選ぶ方式が良いと思われる。すなわち構成要素を指定をすると、属性名のリストがメニューとして現れ、その中から選んで指定するという方法である。ところでその属性名のリストメニューだが、これは制約の種類とは異なり、制約生成ウィンドウの Arg フィールドに表示させ選択するのではなく、選んだ構成要素の近くにポップアップメニューとして表示させるのが良いと思われる。なぜなら、構成要素の属性は普段は表示されないので、場合によっては複数の構成要素の属性リストを同時に表示して見比べたりすることもあるからである。ただしリストが複数出現する場合、どのリストがどの構成要素と対応しているのかが明らかでなくてはならない。そこであるリストの上にポインタが乗ると、そのリストに対応した構成要素がセレクト状態 — 例えばバウンディングボックスで囲まれる — になって対応していることを明示すべきである。そしてポップアップメニューに表示された属性名を Cut&Paste の要領で制約生成ウィンドウの Arg フィールド入力する。（図 3.5）

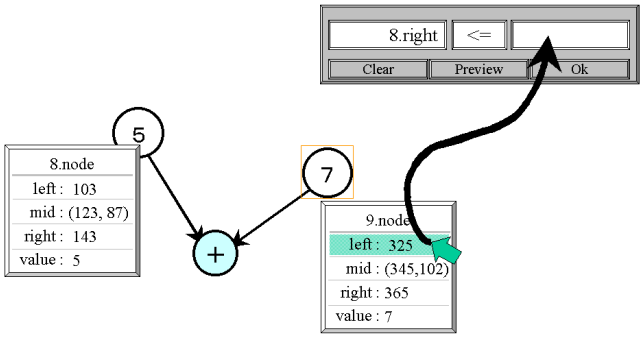

図 3.5: 制約定義：属性名の入力

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

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

3.3.2 制約の削除

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

3.3.3 制約の視覚化

制約の有効 / 無効を選択する際には当然その対象となる制約を選ばなくてはならない。制約を選ぶ方法としては単純にリストに表示してそこから選択するという手法を持ちいる。制約を完全に視覚化して図から選ぶという手法も考えられるが、定数値や「ある図形単語がある図形単語より右にある」などといった視覚化するのが困難、もしくはかえって分かりにくくなる恐れの制約も存在するので、基本的には前述のとおりリストから選択というほうが良いと思われる。ただ、リストに書かれた制約を見ただけでは、

これがどの制約か直観的に理解する事は難しい。そこで選んだ制約がどの制約かを理解するための補助情報として、ある程度の制約の視覚化を考えてみる。

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

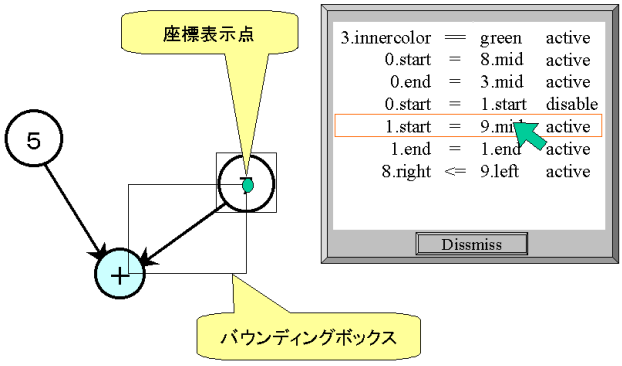

図 3.6: 制約の視覚化

具体的には図 3.6のようになる。

制約が表示されたリストの上にポインタを持ってくると、そのポインタの指す制約の対象となっている属性を持っている構成要素がバウンディングボックスで囲まれる。またその属性が視覚化するのに向いていればさらにそれを視覚的に明示する。図 3.6なら座標情報をその位置に点を表示することにより示している。視覚化に向いている属性としては、座標（点）、図形の色、線の太さなどが考えられる。

3.4 属性

ここで定義する属性は生成された非終端記号が持つ値である。属性は

$$type Attr\_name \ = \ Function$$

の形式であらわせる。 type は定義する属性の型、 Attr_name は定義する属性名、 Function は、構成要素の属性を引数とする任意のスクリプトである。そこで 3.7のような属

性定義ウィンドウを用いて、その各フィールドに型、属性名、スクリプトを何らかの方法で入力することによって制約を生成する。

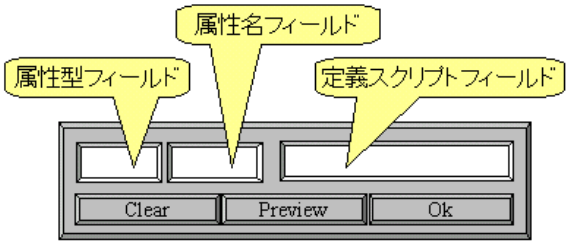

図 3.7: 属性定義ウィンドウ

まず、 type だが、これは恵比寿でも

- integer
- string
- point

の 3 種類のみである。したがって選択肢が少ないので、属性型フィールドにメニューを設け、そこから選ぶようにすれば良いと考えられる。 Attr_name は属性名、すなわち任意の文字列なので、これはフィールドに直接入力する。 Function はスクリプトなので、一般的には従来のテキストによる編集が比較的向いていると考えられる。しかし多くの場合は複雑なスクリプトを記述せず、単純に構成要素の属性そのままだったり、せいぜい四則演算の組み合わせである。したがって属性定義の際には、構成要素の属性を引数として参照する記述がその多くを占める。。

恵比寿では参照する属性を指定する場合、その属性名を全てテキストで記述していた。そのためユーザは長い属性名を一文字も間違えずに打つという操作を強要されていた。またどの構成要素がどの属性を持っているかは表示されないか、表示されていてもテキストで記述された複雑な生成規則の中に埋もれていて、実際にあつかう図形の、どの構成要素のどの属性か把握することが非常に困難であった。

そこで参照する属性を例示入力図を利用して選択し入力する方法を考える。この状況は制約定義の際の属性選択と似ている。したがってヒューマンインターフェイスの観点から見て、統一的な手法を用いたほうが良いので、同様に例示入力図からポインタで参照したい属性をもつ図形を選択、そして表示される属性のリストから参照したい属性をさらに選択し、それを Cut&Paste の要領で入力する。このようにすることにより、その属性のもつ視覚的な情報（線の太さ、色など）や空間的な情報（座標など）を把握しやすく、より直感的に属性定義を行うことができる。

図 3.8では定義する属性の型と属性名をそれぞれのフィールドに入力したあと、継承したい属性をもつ構成要素の属性リストから、その属性を選択し、スクリプトフィールドに入力している。

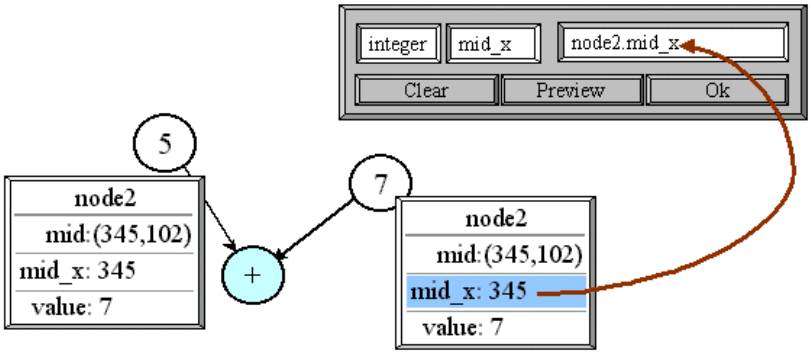

図 3.8: 属性の定義

3.5 アクション

アクションとは生成規則が適用され図形が認識された際、スクリプト言語のプログラムとして実行さえる文字列である。スクリプト言語なので、一般的には従来のテキストによる編集で十分と思われる。しかし、図形言語特有の操作である図形の書き換えなどを行う delete（図形の削除）、alter（図形の属性値の変更）、create（図形の生成）については、やはり扱う情報が二次元的なものであるため、一次元的なテキストで記述するのは直感的ではない。そこで、この３つの図形操作について例示入力図を利用した視覚的な定義方法を考える。

また図形で定義する際、図形操作が行われる前の図と後の図を同時に表示した方が図形書き換えの前後の変化がより具体的に把握できると思われる。したがって図形操作のアクションを定義する際は、図 3.9のように例示入力図のコピーを例示入力図の隣に描画し、それに対して実際に図形操作を行うことで定義を行うほうが良いと考えられる。

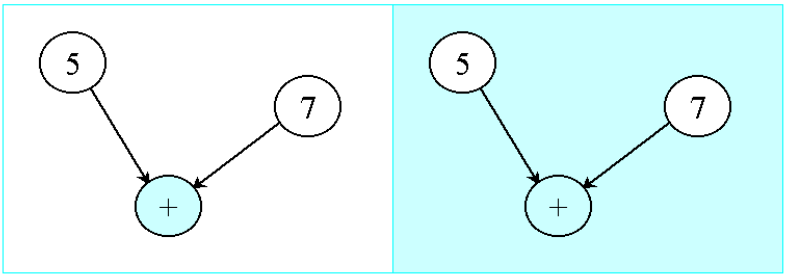

図 3.9: アクションの定義

3.5.1 delete

delete、すなわち図形の削除を定義するには、対象となる図形を実際に削除する、という操作自体が定義方法として最も直感的である。

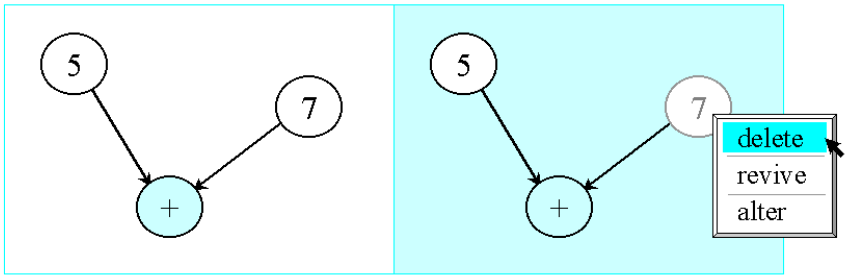

図 3.10: delete の定義

図 3.10のように、削除したい図形を例示入力図より選択しクリックする。するとアクション定義に関するポップアップメニューが表示される。そこから削除を選択することにより、実際に図形の削除をもって delete の定義とする方法が良いと思われる。

ところで、生成規則を定義する際には割と試行錯誤を繰り返すことになる。よって不必要と思って削除した図形がやはり必要となることもある。そこで削除指定された図形は実際には完全に消えるわけではなく、元より薄い色で表示するようにする。削除を取り消したい場合は、その薄く表示された図形を再びクリックしてポップアップメニューを表示させ、そこから削除取り消しを選択することにより、元の状態に戻すことができる。このようにすることにより、削除という操作に可逆性を持たせることが出来る。

3.5.2 alter

alter は既に存在している図形の属性値を変更する操作である。したがってこれも実際に図形を書き換えをもって操作の定義とするのが最も直感的である。しかし実際に変更するのは図形のもつ多くの属性のうちのごくわずか、大抵１つである。この一つの変更を表すために、書き換える図形がもつ全ての属性を指定して、その図形を１から書きなおすのは効率が悪い。従って図形を直接書き換えるのではなく、変更する属性を指定し、それだけを書き換え、その変更の結果として図形をシステムが自動的に書き換えることでユーザにフィードバックし、その一連の操作をもって alter の定義とするのが良いと思われる。

変更する属性を選択するには、まず変更したい図形を選んでから、その図形中にある属性を選択するという順番が自然であり、直感的に理解しやすい。まず構成要素の指定であるが、これはこれまでにあるように、例示入力図からポインタで指し示すことにより指定する方法を用いる。統一的な手法を用いたほうが、ヒューマンインターフェイスの観点から見て良いからである。次に属性の選択であるが、変更する属性は一般には複

数である。従って選択された図形がもつ全ての属性の属性名と属性値の組をリストを、選んだ構成要素の近くにポップアップメニューとして表示させる手法が良いと思われる。そしてリストメニュー上で、変更したい属性の属性値を直接変更する。具体的には図 3.11のようになる。

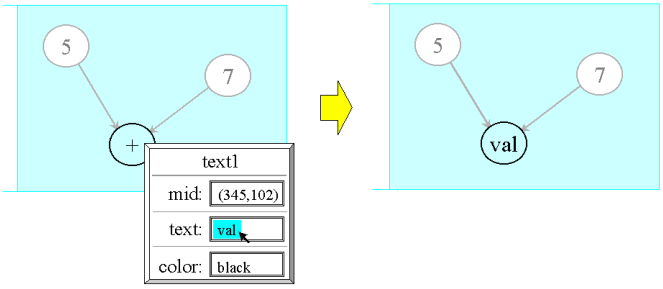

図 3.11: alter の定義

このように属性を変更すると、その変更した属性にしたがい、例示入力図の図形が自動的に書き換えられ、定義が行われたフィードバックを示す。また書き換えられることで、隣にある書き換えられる前の例示入力図との差分により、定義した属性変更を視覚的に表現できる。

なお、属性定義と同じく、属性の値として他の図形の属性などを参照することもできる。参照したい属性を、やはり例示入力図から図形を選択→表示される属性リストから属性を選択し、それを変更したい属性の値として Cut&Paste の要領で入力する。更に参照した属性を元に、演算を行うことも、同様に可能である。

3.5.3 create

create は新たに図形を生成する操作である。これもまた、実際に生成したい図形を例示入力図に実際に描き、例示することによって定義を行うのが最も直感的で分かりやすい。 create は生成する図形の全ての属性を定義する必要があるので、テキストでそれを定義する場合は、座標、色、長さなど、全ての属性について CMG に基づいて非常に複雑な記述を行わなくてはならない。また実際に望む図形を生成するという行為と、それを行うために必要な操作がかけ離れているため、直感的に理解できない。一方実際に描画という操作は、当然図形生成に必要な属性を全てを容易に、かつ直感的に定義することができる。さらに定義後も、その create がどのような属性を指定したのかを、テキストよりもより容易に、それこそ一目みただけで把握できるという利点がある。

図 3.12は左右の円の中心を結ぶ線を生成するということを定義してる。図形で定義する場合は、実際に円の中心を結ぶ線を描画するだけで定義が完了するが、これをテキストで定義する場合、始点、終点となる円の中心点の座標、線の色、太さ、タイプ（矢印着きかそうでないか）、といった属性を全て、 CMG の文法に則って記述しなくてはならない。

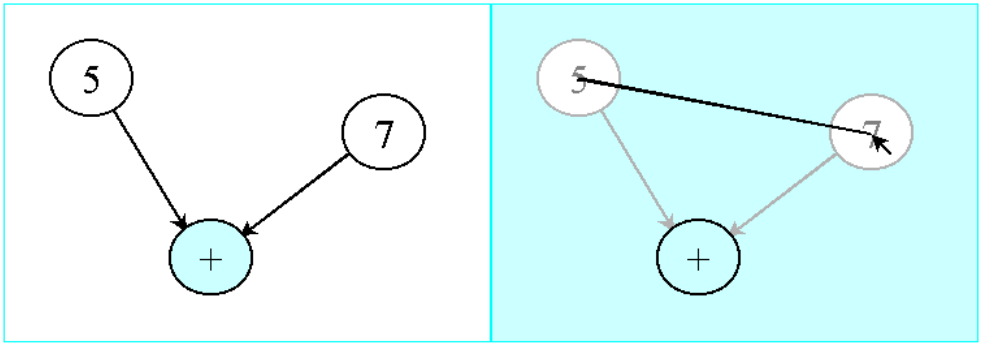

図 3.12: create の定義

第 4 章

Spatial Parser Generator 「VIC」

本章では、前章の考察に基づいた定義インターフェイスをもつ Spatial Parser Generator VIC[15][16][17] について述べる。

4.1 システム VIC

VIC の実行画面は図 4.1のようになる。

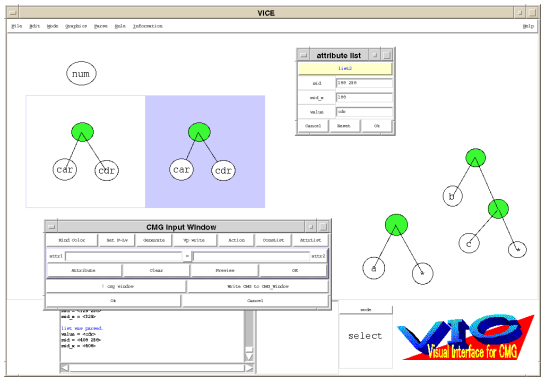

図 4.1: 実行画面

ユーザはメタ文法である拡張 CMG により図形言語の文法を定義することにより、Spatial Parser を生成し、これを用いて、描画された様々な図形言語の解析し、処理を行うことができる。またこれらの作業が一貫して一つのウィンドウで行える。定義と実行の境界がなくなることにより、スムーズな操作と、インタラクティブな図形言語の定義を可能としている。

また、テキスト編集を行わなず、ほぼ例示入力図に対するマウス操作のみで生成規則を定義できる。そのため拡張 CMG における文法をほとんど意識する必要はない。したがって拡張 CMG に対する十分な知識がないユーザでも、直感的に図形言語を定義できる。

4.2 実装

4.2.1 使用言語

実装には Tcl/Tk[20] を用いた。

4.2.2 制約解消系

VIC では図形の解釈が行われたあと、生成規則中で定義した制約をそのトークンに課す。これにより、解釈された図形同士の空間的な意味の繋がりを保存できる。これらの制約を常に成り立たせるための機構を制約解消系と呼ぶ。VIC では制約解消系として SkyBlue[21] を用いた。SkyBlue 自体は C 言語で実装されている。そこで SkyBlue を Tcl から呼び出すインターフェイスを作成しこれを介して用いた。インターフェイスでは 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、制約を課している変数の ID を得るなどの操作ができる。

4.2.3 各機能の詳細

構成要素の定義

構成要素の例示入力には VIC に付属する図形エディタを用いる。この図形エディタは VIC で用いられる終端記号の描画と、図形エディタとして一般的な操作（cut, copy, paste 等）が可能である。終端記号としては

- rectangle ··· 矩形
- oval ··· 楕円
- line ··· 直線
- arc ··· 円弧
- text ··· 文字列
- image ··· GIF 画像

を用いることができる。

構成要素の種類は 4 種類あり、それぞれ

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

で表現した。また構成要素のタイプの表示であるが、補助情報として本来のシステムメッセージウィンドウの横に図 4.2のように専用のインフォメーションウィンドウを設け、そこに表示するようにした。この時その構成要素が非終端記号の場合、さらにその構成要素が終端記号まで同時に表示される。

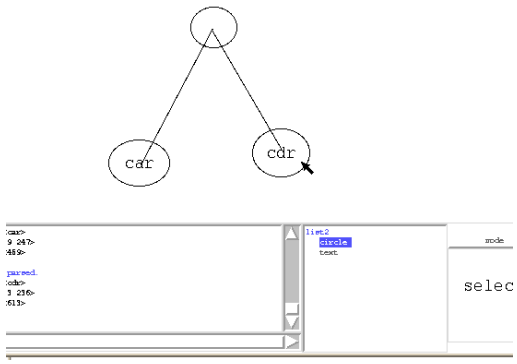

図 4.2: 構成要素のタイプ表示

制約の定義

制約は以下の 7 種類を用いることができる。

- eq ··· equal
- neq ··· not equal
- gt ··· greater than
- ge ··· greater or equal
- lt ··· less than
- le ··· less or equal
- vp_close

vp_close 制約は eq 制約の変形で、いわば「やや近い」という制約である。対象となる変数同士がある程度近い値なら、その変数間に eq 制約が課せられる。

制約の編集方法については考察で述べた通りである。ただ制約の自動生成において、座標が一致するという制約については、例示入力図に視覚的に表現し、かつそこで制約の有効 / 無効を選択できるようにする手法を試験的に実装してみた。具体的にはメインメニュー上の「制約表示ボタン」を押すと、 vp_close 制約がかかっている点に円が表示される。その円にポインタを合わせクリックするとメニューが表示され（図 4.3）そこでその vp_close 制約を無効にするかを選ぶ。このときインフォメーションウィンドウには vp_close 制約がかかっている 2 つのトークンが表示される。

属性の定義

属性の型としては以下の 3 種類を用いることが出来る。

- integer ··· 数値
- string ··· 文字列

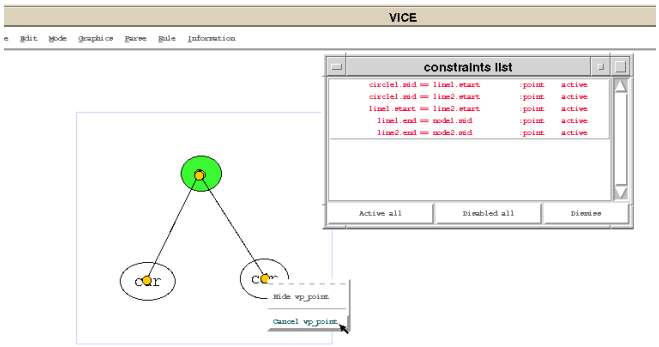

図 4.3: 制約の視覚化

- point　　・・・ 2 次元座標

script 命令の処理系としては実装に用いた Tcl の処理系をそのまま用いる。

また考察で述べた定義方法の他に、定義しおえた属性の有効／無効を選択できる機能を実装した。具体的には制約の項で述べたように、定義した属性のリストから、変更したい属性を選択し、有効／無効を切替える。無効にした属性は実際に生成規則には反映されない。

アクションの定義

delete、 alter については考察で述べたとおりである。

create に関しては、新たに生成する図形の位置座標は、既にある構成要素との相対的な座標になるようにした。具体的には以下のアルゴリズムで例示入力図より汎化する。

1. 描画された図形の位置座標が、既にある構成要素の座標型の属性に近い場合、

　1-1. その座標型属性を位置座標とする。

2. 1. でない場合、

　2-1. 定義されたトークンが、座標型の属性を 2 つ以上持っている場合、

　　2-1-1. 属性型の属性の示す任意の 2 点を A, B、描画された図形の位置座標の示す点を X とする。

　　2-1-2. 座標からベクトル $\vec{AB}$、 $\vec{AX}$ を計算する。また $a = \frac{||\vec{AX}||}{||\vec{AB}||}$ とする。これらはそれぞれ定数となる。

　　2-1-3. Parsing された図形の A, B に相当する点を A', B' とすると、生成される図形の位置座標 X' は、 $X' = A' + a\frac{||\vec{A'B'}||}{||\vec{AB}||}\vec{AX}$ となる。

　2-2. 定義されたトークンが、座標型の属性を 1 つしか持っていない場合、

2-2-1. 属性型の属性の示す点を A、描画された図形の位置座標の示す点を X とする。

2-2-2. 座標からベクトル $A\vec{X}$ を計算する。これは定数となる。

2-2-3. Parsing された図形の A に相当する点を A' とすると、生成される図形の位置座標 X' は、 $X' = A' + A\vec{X}$ となる。

2-2. それ以外、つまり定義されたトークンが、座標型の属性を持っていない場合、

2-2-3. 描画された図形の位置座標が、生成される図形の位置座標となる。

第 5 章

ビジュアルシステムの作成例

本章では VIC の実行例として、 VIC を用いて実際にビジュアルシステムを作成する例をあげる。

5.1 データ構造１：スタック

基本的なデータ構造であるスタックを処理するビジュアルシステムを作成する。実行例は図 5.1のようになる。またスタック特有の操作である push や pop といった機能も実装する。

スタックは以下の 2 つの生成規則より再帰的に定義される。

生成規則１. line は stack（ベースライン）である。

生成規則２. stack の上に node が在れば、 stack である。

また node を定義するために、以下のルールが必要となる。

生成規則３. rectangle の中央に text があれば、それは node である。

さらに pop 操作を行うために、以下のルールも定義する。

生成規則４. stack の上に pop 記号（circle）が置かれたら、 stack の最上段の値を返す。

ちなみに push 操作は生成規則 2 がそれにあたる。

これらの生成規則をより厳密に拡張 CMG の生成規則に従って記述すると以下のようになる。

```
01:  stack(integer lu_y, integer rl_y, integer mid_x,
02:                            string value, string stack_value) ::=
03:     L:line
04:     where {
05:     } and {
06:        integer lu_y  = L.mid_y
07:        integer rl_y  = L.mid_y
```

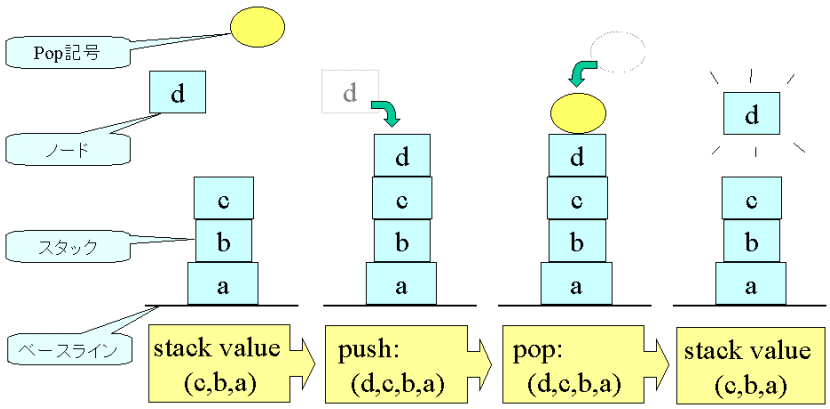

図 5.1: ビジュアルシステムの例：スタック

```
08:        integer mid_x = L.mid_x
09:        string  value = {string ‘‘’’}
10:        string  stack_value = {string ‘‘’’}
11:     } and {
12:     }
13:
14:  stack(integer lu_y, integer rl_y, integer mid_x,
15:                           string value, string stack_value) ::=
16:     N:node
17:     exists, S:stack
18:     where {
19:        N.rl_y  == S.lu_y
20:        N.mid_x == S.mid_x
21:     } and {
22:        integer lu_y  = N.lu_y
23:        integer rl_y  = N.rl_y
24:        integer mid_x = N.mid_x
25:        string  value = N.value
26:        string  stack_value = {script.string {set x
27:                    [concat [list @N.value@] [list @S.stack_value@]]}}
28:     } and {
29:     }
30:
31:  node (integer lu_y, integer rl_y, integer mid_x, string value) ::=
32:     R:rectangle, T:text
33:     where {
34:       R.mid == T.mid
35:     } and {
36:        integer lu_y  = R.lu_y
37:        integer rl_y  = R.rl_y
38:        integer mid_x = R.mid_x
39:        string  value = T.text
```

```
40:     } and {
41:     }
42:
43:  node (integer lu_y, integer rl_y, integer mid_x, string value) ::=
44:     C:circle S:stack
45:     where {
46:        C.mid_x == S.mid_x
47:        C.rl_y  == S.lu_y
48:     } and {
49:        integer lu_y  = C.lu_y
50:        integer rl_y  = C.rl_y
51:        integer mid_x = C.mid_x
52:        string  value = S.value
53:     } and {
54:        delete {C S}
55:        create rectangle @C.lu_x@ @C.lu_y@ @C.rl_x@ @C.rl_y@
56:                                          -fill white -outline black
57:        create text @C.mid_x@ @C.mid_y@ -text @S.value@
58:     }
```

01 行から 12 行までが生成規則 1 の定義である。 01,02 行は生成されるトークンの名前「Name」と属性を定義している。名前が stack であり、 integer 型の lu_y, rl_y, mid_x、string 型の value, stack_value という属性をもつ。 03 行ではどのような構成要素からできているかを定義している。ここでは normal 要素の一本の線（line） L のみから成るということを示している。 04 行は制約「Constraint」の定義だが、生成規則 1 では特に定義していない。 05 〜 10 行は属性「Attribute」の定義である。例えば 06 行では integer 型の lu_y という属性（left-upper-y：左上隅ポイントの y 座標）の値が line：L の mid_y （middle-y：中心ポイントの y 座標）という属性の値と同じであると定義している（属性値の継承）。その他の属性の意味だが、 07 行の rl_y は right-lower-y で右下隅ポイントの y 座標、 08 行の mid_x は middle-x で中央ポイントの x 座標、value は stack 自身の値、 stack_value はスタック全体の値である。 11 行はアクション「Action」の定義だが、これも生成規則 1 では特に定義していない。

14 行から 29 行までが生成規則 2 の定義である。 14,15 行は生成されるトークンの名前と属性を定義している。 16,17 行は構成要素の定義である。ここでは normal 要素として非終端記号 node 、そして exist 要素として非終端記号 stack を必要とすることを示している。 19 〜 20 行は制約「Constraint」の定義である。 19 行は node： N の rl_y が stack： S の lu_y に、 20 行は node： N の mid_x が stack： S の mid_x に、それぞれ一致していなくてはならないという意味である。具体的には node： N の上辺と stack： S の下辺が触れているという意味である。 22 〜 27 行は属性「Attribute」の定義である。生成規則 1 とほぼ同じであるが、 26 行で script 指令を用いている。 script 指令とは書かれた文字列をスクリプト処理系に渡し実行させ、図形単語の属性値を生成するのに用いられる。 script 指令は次のような形式である。

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

ただし *type* は得られた結果をどのような型として扱うかを示し、 *script* はスクリプト

処理系に仮借させる文字列である。なお *script* 内では @ マークに囲まれた属性は、その属性値に変換されて処理系に渡される。 script 指令が無い場合 CMG の処理系があらかじめ持っている関数を用いて値を生成するしかないが、 script 指令を用いることによって、スクリプト言語によってより高級な演算が可能となる。ここでは node： N の value という属性値と stack： S の stack_value という属性値をそれぞれリストに変換し、その 2 つのリストを結合するという操作を定義している。

31 行から 41 行までが生成規則 3 の定義である。特に変わった記述はないので説明は省略する。

43 行から 58 行までが生成規則 4 の定義である。生成規則 4 では 54 〜 57 行でアクション「Action」の定義をしている。 54 行では構成要素である circle： C と stack： S の delete を定義している。 55,56 行では新たに rectangle の、 57 行では text の create を定義している。

5.1.1 生成規則１の定義

生成規則 1 を定義する過程を述べる。

1 構成要素の定義を行う。

1-1 構成要素の例示入力、すなわちキャンバス上に実際に line を描画する。

1-2 直線を選択し、 Rule メニューから VCW を選択する。

1-3 図 5.2のように例示入力図周辺がボックスで囲まれ、メインメニューが現れる。

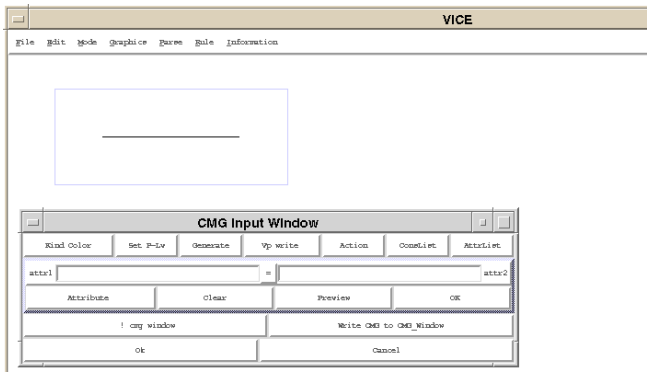

図 5.2: 構成要素の定義

2 属性の定義を行う。

2-1 まず、 lu_y = L.mid_y という属性の定義を行う。

2-2 定義ウィンドウを属性定義モードにし、第 1 フィールドに属性名 lu_x と入力する。（図 5.3A）

2-3 例示入力図の line をクリックし、メニューから属性リストを選択して表示させる。

2-4 属性リストから継承させたい属性 — 図 5.3 B の場合 mid_y — を選択し、これをダブルクリックで定義ウィンドウの第 2 フィールドに入力する。

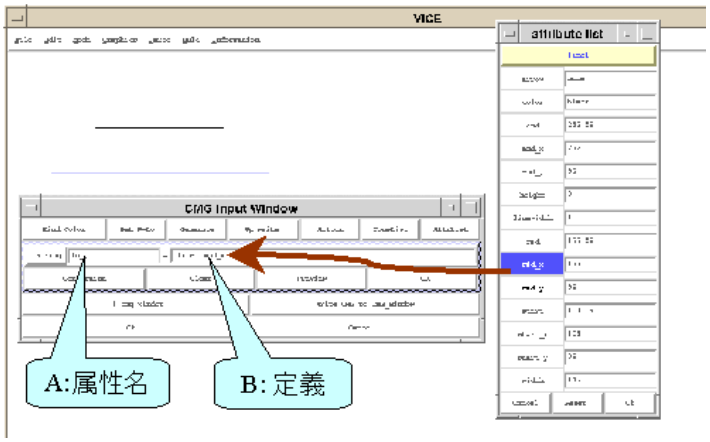

図 5.3: 属性の定義

2-5 OK を選択し、定義完了。

2-6 他の属性も同様に定義する。図 5.4は全ての属性を定義したところである。

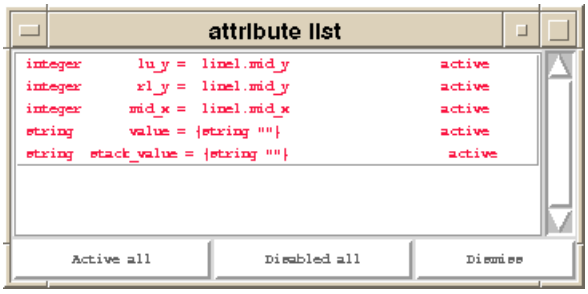

図 5.4: 定義した属性

制約、アクションは定義する必要がないので、以上で生成規則 1 の定義は終了である。

5.1.2 生成規則 3 の定義

生成規則 2 の前に、生成規則 2 で必要な node を定義する生成規則 3 を先に定義する。

1 構成要素の定義を行う。

1-1 図 5.5のように、キャンバス上に rectangle を描き、さらにその中心付近に text を描画する 。

1-2 全体を選択し、 Rule メニューから VCW を選択すると、例示入力図周辺がボックスで囲まれ、メインメニューが現れる。

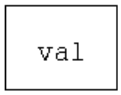

図 5.5: 構成要素の定義

2 制約の定義を行う。

2-1 R.mid == T.mid という制約の定義を行う。

2-2 定義ウィンドウを制約定義モードにし、制約メニューから「==」という制約を選択する。（図 5.6）

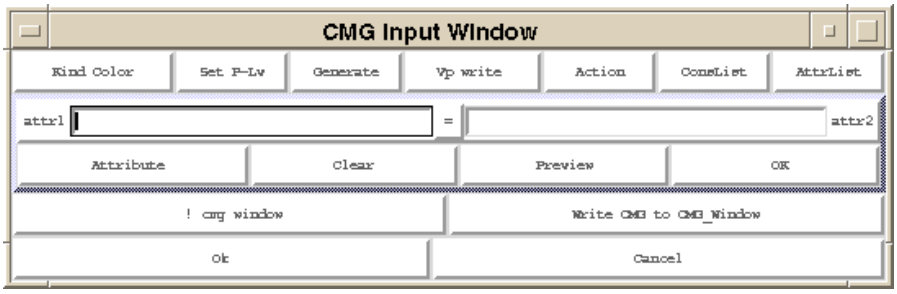

図 5.6: 制約の定義 1

2-3 制約を課したい属性をもつ構成要素をクリックし、メニューから属性リストを選択して表示させる。

2-4 属性リストから制約を課したい属性を選択し、これを定義ウィンドウのフィールドに入力する。図 5.7は rectangle： R の中心座標： mid という属性を選択、第 1 フィールドに入力している。

2-5 第 2 フィールドにも属性： T.mid を入力し、 OK を選択して定義完了。

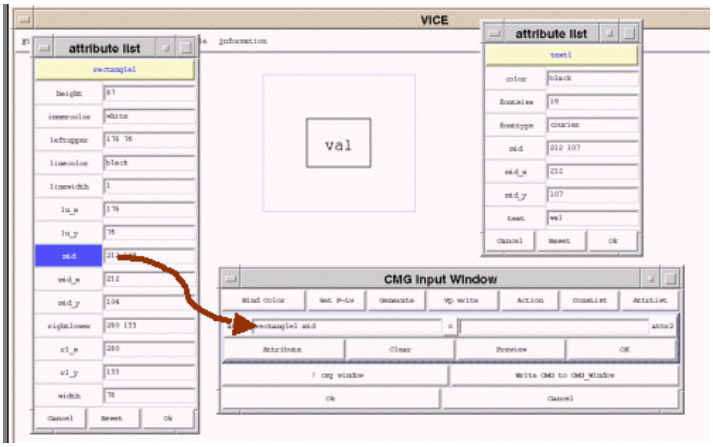

図 5.7: 制約の定義 2

3 属性の定義を行う。

3-1 生成規則 1 とほぼ同じなので省略。

アクションは定義する必要がないので、以上で生成規則 3 の定義は終了である。

5.1.3 生成規則 2 の定義

生成規則 2 を定義する過程を述べる。

1 構成要素の定義を行う。

1-1 構成要素の例示入力を行う。まず stack は生成規則 1 に基づき line を描く。そして node は生成規則 3 に基づき rectangle とその中央に text を描画する。（図 5.8）

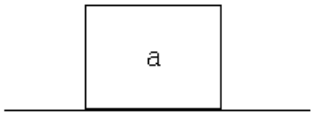

図 5.8: 構成要素の定義 1

1-2 全体を選択し、 Rule メニューから VCW を選択すると、例示入力図周辺がボックスで囲まれ、メインメニューが現れる。この際、既に定義された生成規則を用い

て Spatial Parsing が行われる。すなわち生成規則 1 , 3 を用いて stack と node が認識される。 図 5.9は、 rectangle と text が node と認識されていることを示している。

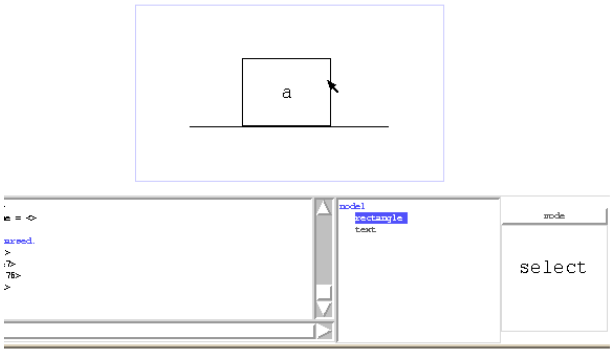

図 5.9: 構成要素の定義 2

1-3 構成要素 stack を exist 要素と定義する。図 5.10のように、例示入力図から stack をクリックし、出現したメニューから exist を選択する。

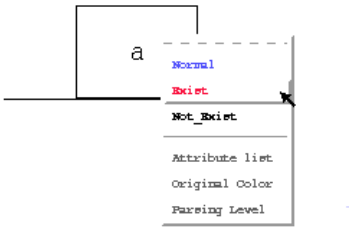

図 5.10: 構成要素の定義 3

2 制約の定義を行う。

2-1 生成規則 3 と同様に、制約を課したい属性を属性リストより選択し、定義ウィンドウを用いて定義する。詳しくは省略する。

3 属性を定義する。生成規則 1 とほぼ同じであるが、 stack_value の定義で script 指令を用いているので、これについて説明する。

3-1 定義ウィンドウを属性定義モードにし、第 1 フィールドに属性名 stack_value と入力する。

3-2 第２フィールドにスクリプト命令を入力する。スクリプト中で参照したい属性がある場合、図 5.11のように、既に表示されている属性リストから選択することで入力できる。

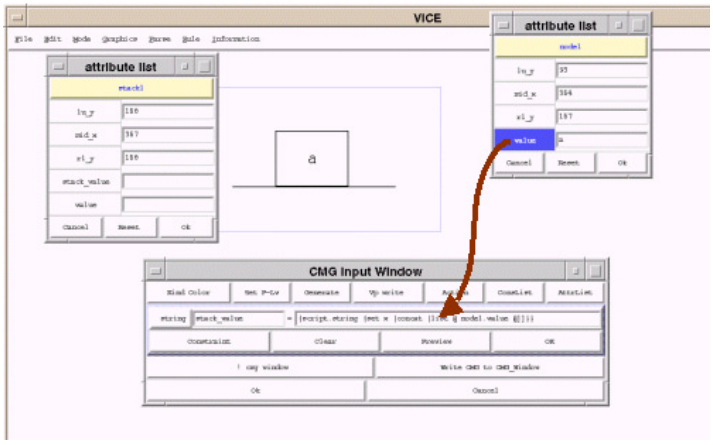

図 5.11: 属性の定義

3-3 OK を選択し、定義完了。

アクションは定義する必要がないので、以上で生成規則２の定義は終了である。

5.1.4 生成規則４の定義

最後に生成規則４を定義する過程を述べる。

1 構成要素の定義を行う。

1-1 構成要素の例示入力を行う。まず stack は生成規則１に基づき line を描く。そしてその上に circle を描画する。（図 5.12）

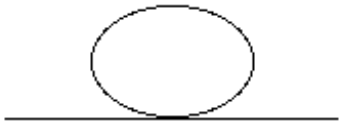

図 5.12: 構成要素の定義１

1.2 全体を選択し、 Rule メニューから VCW を選択すると、例示入力図周辺がボックスで囲まれ、メインメニューが現れる。この際、既に定義された生成規則を用いて Spatial Parsing が行われる。すなわち生成規則 1 を用いて line が stack と認識される。

2 制約の定義を行う。

2-1 生成規則 2 とほぼ同じなので、省略する。

3 属性の定義を行う。

3-1 生成規則 3 とほぼ同じなので、省略する。

4 アクションの定義を行う。

4-1 アクションの定義は、図 5.13のように、例示入力図の横に表示される図形に対する操作で行う。

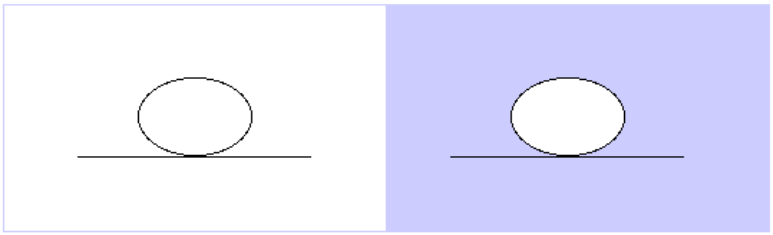

図 5.13: アクションの定義

4-2 circle： C、 stack： S を delete するアクションを定義する。図 5.14のように、delete したいトークンを例示入力図から選択し、メニューで delete を選択する。delete 選択されたトークンは薄い色で表示される。

4-3 rectangle を create するアクションを定義する。図 5.15のように、 create したい図形を実際に描画する。

4-4 text を create するアクションも同様に定義する。 text の文字として stack： S の属性 value を参照したいので、属性リストより選択して入力する。

4-5 アクション定義後の左右の例示入力図の差分でアクションが視覚的に把握できる。（図 5.16）

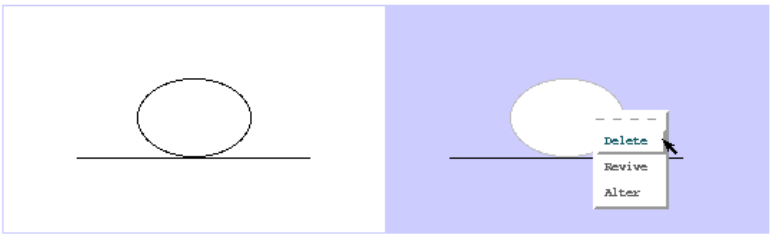

図 5.14: delete の定義

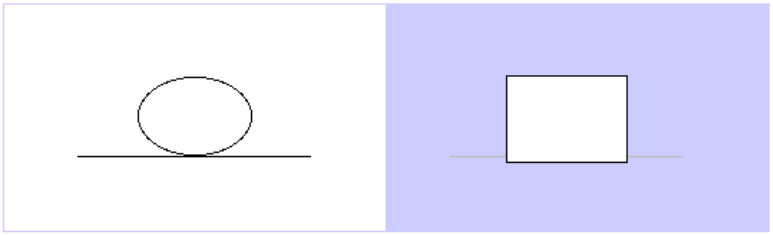

図 5.15: create の定義 1

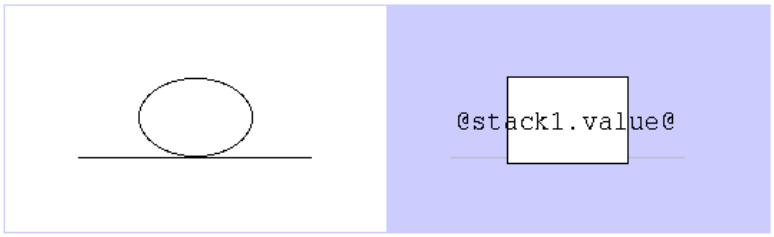

図 5.16: アクションの定義 2

以上で生成規則４の定義は終了である。
　これでスタックの定義は完成である。

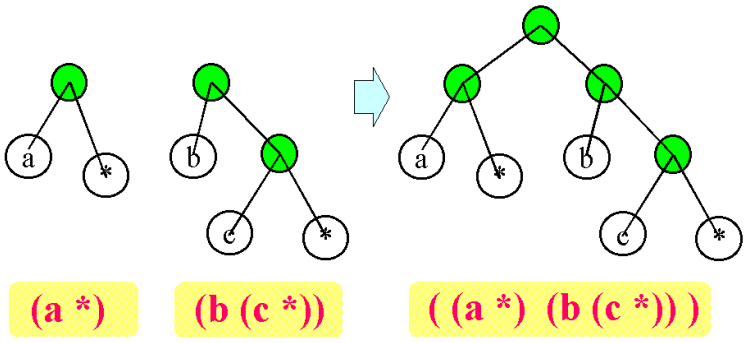

図 5.17: ビジュアルシステムの例：二分木リスト

5.2 データ構造２：二分木リスト

基本的なデータ構造である二分木リストを処理するビジュアルシステムを作成する。実行例は図 5.17のようになる。また二分木リスト同士の結合の操作である append 機能も実装する。

二分木リストは以下の２つの生成規則より再帰的に定義される。

生成規則１． circle の中心に text があれば、それは list である。

生成規則２． ２つの list が１つの circle に、それぞれ line で結ばれていたら、それは list である。

生成規則２が append 機能に相当する。

これらの生成規則をより厳密に拡張 CMG の生成規則に従って記述すると以下のようになる。

```
01:  list(point mid, integer mid_x, string value) ::=
02:     C:circle, T:text
03:     where {
04:        C.mid == T.mid
05:     } and {
06:        mid = C.mid
07:        mid_x = C.mid_x
08:        value = T.text
09:     } and {
10:        puts ‘‘value = @value@’’
11:     }
12:
13:  list(point mid, integer mid_x, string value) ::=
14:     C:circle
15:     exists S1:list, S2:list, L1:line, L2:line
16:     where {
```

```
17:        S1.mid == L1.end
18:        S2.mid == L2.end
19:        C.mid == L1.start
20:        C.mid == L2.start
21:        C.mid == T.mid
22:        S1.mid_x < S2.mid_x
23:     } and {
24:        mid = C.mid
25:        mid_x = C.mid_x
26:        value = {script.string {concat [list @S1.value@][list @S2.value@]}}
27:     } and {
28:        puts ‘‘value = @value@’’
29:     }
```

01 行から 11 行までが生成規則 1 の定義である。 01 行は生成されるトークンの名前「Name」と属性を定義している。名前が list であり、 mid, mid_x, value という属性値を持ち、それぞれの型は point 型、 integer 型、 string 型である。 02 行では図形単語がどのような構成要素からできているかを定義している。ここでは全て normal 要素で、一つの円（circle） C と文字列（text） T から成るということを示している。 04 行は制約「Constraint」を定義している。ここでは円 C の中心座標： mid という属性と文字列 T の中心座標： mid という属性がほぼ一致しているという制約を定義している。 06 行 〜 08 行は属性「Attribute」の定義である。例えば 06 行は mid という属性が、円 C の中心座標： mid と常に等しいということを示している。 09 行は value という属性の値を定義している。ここでは text： T の文字列： text をリストとして扱うことを示している。 10 行目は「Action」を定義している。 puts は tcl の出力コマンドで、この生成規則が適用された際、 `value = 3` などと表示される。なお @ マークで囲まれた属性は、その属性値に変換される。

13 行からが生成規則 2 の定義である。生成規則 1 とほとんど同じであるので異なる点のみを説明する。まず 15 行であるが、これは構成要素として exist の構成要素を必要とすることを示している。 22 行の制約は list： S1 の中心の x 座標： mid_x が list： S2 の中心の x 座標： mid_x より小さい、すなわち list： S1 が list： S2 より左にあるということを示す制約であるこの制約は左右の list を区別するためにある。この制約が無ければ、 list の car 部と cdr 部を決定できない。 26 行では 2 つの list の連結を指定している。

5.2.1 生成規則１の定義

生成規則 1 を定義する過程を述べる。

1 構成要素の定義を行う。

1-1 構成要素の例示入力、すなわちキャンバス上に実際に circle と text を描画する。

1-2 全体を選択し、 Rule メニューから VCW を選択する。すると図 5.18のように例示入力図周辺がボックスで囲まれ、メインメニューが現れる。

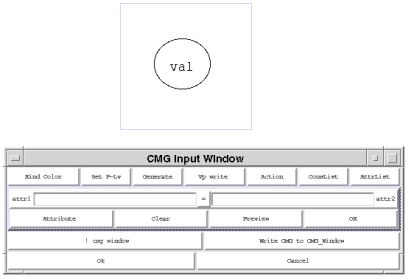

図 5.18: 構成要素の定義

2 制約の定義を行う。

2-1 C.mid == T.mid という制約の定義を行う。

2-2 定義ウィンドウを制約定義モードにし、制約メニューから「==」という制約を選択する。

2-3 制約を課したい属性をもつ構成要素をクリックし、メニューから属性リストを選択して表示させる。

2-4 属性リストから制約を課したい属性を選択し、これを定義ウィンドウのフィールドに入力する。図 5.23は circle： C の 中心座標： mid という属性を選択、第 1 フィールドに入力している。

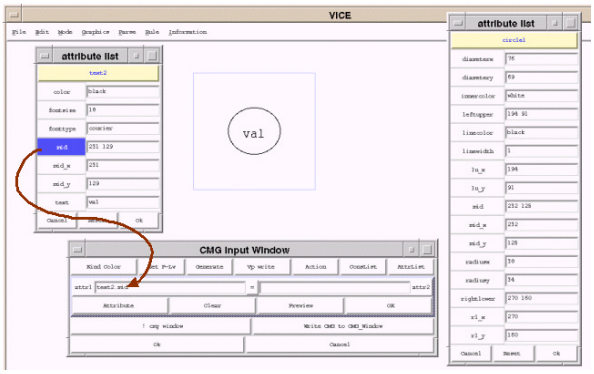

図 5.19: 制約の定義

2-5 第 2 フィールドに属性： T.mid を入力し、 OK を選択して定義完了。

3 属性の定義を行う。

3-1 まず、 mid = C.mid という属性の定義を行う。

3-2 定義ウィンドウを属性定義モードにし、第 1 フィールドに属性名 mid と入力する。

3-3 例示入力図の circle をクリックし、メニューから属性リストを選択して表示させる。

3-4 属性リストから継承させたい属性 — 図 5.20の場合 mid — を選択し、これをダブルクリックで定義ウィンドウの第 2 フィールドに入力する。

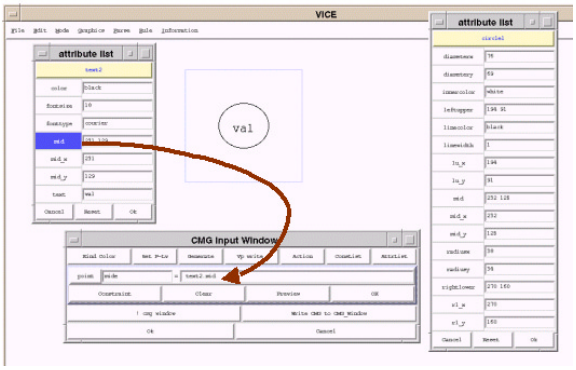

図 5.20: 属性の定義

3-5 OK を選択し、定義完了。

3-6 他の属性も同様に定義する。

4 アクションの定義を行う。

4-1 今回は単なるスクリプト命令なので、テキストで入力する。

以上で生成規則 1 の定義は終了である。

5.2.2 生成規則 2 の定義

生成規則 2 を定義する過程を述べる。

1 構成要素の定義を行う。

1-1 構成要素の例示入力を行う。 node は生成規則 1 に基づき circle とその中央に text を描画する。

1-2 全体を選択し、 Rule メニューから VCW を選択すると、例示入力図周辺がボックスで囲まれ、メインメニューが現れる。この際、既に定義された生成規則を用いて Spatial Parsing が行われる。すなわち生成規則 1 を用いて node が認識される。 図 5.21は、 circle と text が node と認識されていることを示している。

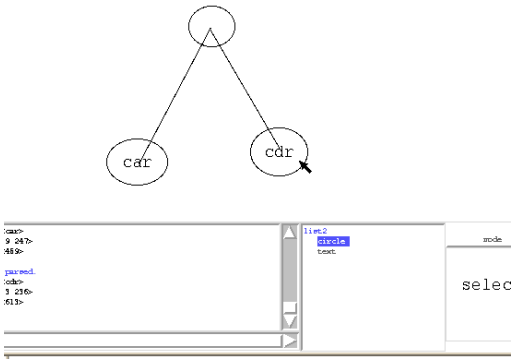

図 5.21: 構成要素の定義 1

1-3 必要な構成要素を exist 要素と定義する。図 5.22のように変更したい構成要素をクリックし、メニューから exist を選択する。

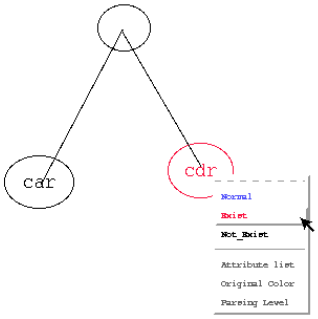

図 5.22: 構成要素の定義 2

2 制約の定義を行う。

2-1 今回は図形的な制約が多く必要なので制約の自動生成機能を使用する。メインメニューの「Generate」ボタンを押すと制約が自動生成される。

2-2 どんな制約が生成されたかは、図 5.23のような制約リストで確認できる。

2-3 制約リストで不必要な制約を無効にする。この際、現在ポインタが指しているリスト上の制約の対象となっている構成要素がバウンディングボックスに囲まれ例示入力図上に表示されるので、これを参照しながら行うと楽である。

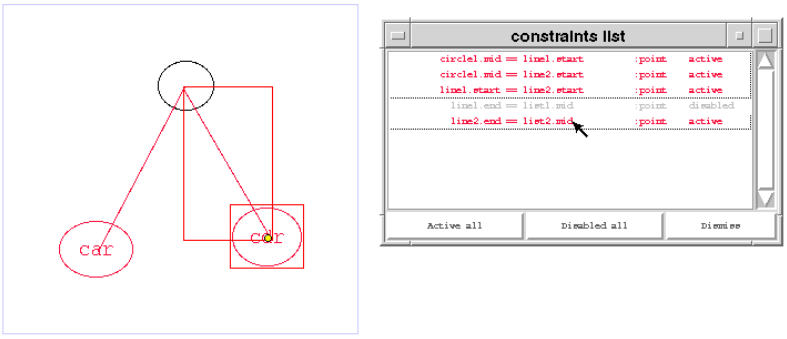

図 5.23: 制約の定義

2-4 足りない制約は定義ウィンドウを用いて手動で定義する。

3 属性の定義を行う。生成規則 1 とほぼ同じであるが、 value の定義で script 指令を用いているので、これについて説明する。

3-1 定義ウィンドウを属性定義モードにし、第 1 フィールドに属性名 value と入力する。

3-2 第 2 フィールドにスクリプト命令を入力する。スクリプト中で参照したい属性がある場合、図 5.24のように、既に表示されている属性リストから選択することで入力できる。

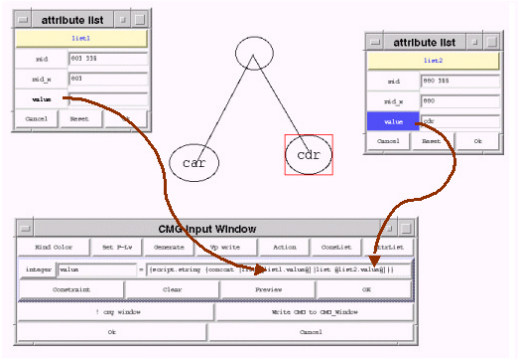

図 5.24: 属性の定義

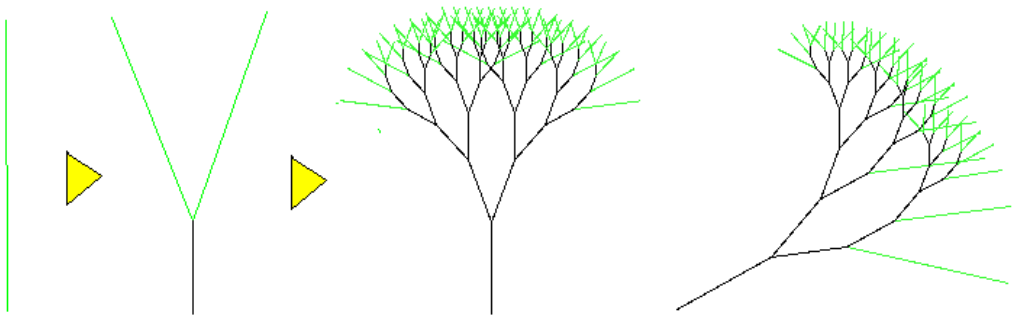

図 5.25: ビジュアルシステムの例：フラクタルの樹

3-3 OK を選択し、定義完了。

4 アクションの定義を行う。

4-1 今回は単なるスクリプト命令なので、テキストで入力する。

以上で生成規則 2 の定義は終了である。
これで二分木の定義は完成である。

5.3 描き換えアルゴリズム：フラクタルの樹

特定の図形を描き換えるビジュアルシステムを作成する。フラクタルの樹は図 5.25のように、1 本の直線を認識するとそれを Y 字型に描き換える。さらに描き換えた直線に生成規則を適用し、再帰的に描き換え、最終的に樹のような図形を生成するビジュアルシステムである。
フラクタルの樹は以下の生成規則より再帰的に定義される。

生成規則. 緑色の line があれば、それを黒の line （Y 字型の根元の）と、その先端から斜めの方向の緑の line （Y 字型の左右の枝）に描き換える。

この生成規則をより厳密に拡張 CMG の生成規則に従って記述すると以下のようになる。

```
01:  branch()::=
02:     L:line
03:     where {
04:        L.color == {string green}
05:        L.height > {integer -20}
06:     } and {
07:     } and {
08:        set v_x [expr @L.end_x@ - @L.start_x@]
```

```
09:         set v_y [expr @L.end_y@ - @L.start_y@]
10:         set r_x [expr @L.start_x@ + $v_x / 3]
11:         set r_y [expr @L.start_y@ + $v_y / 3]
12:         set d1_x [expr @L.end_x@ - $v_y / 4]
13:         set d1_y [expr @L.end_y@ + $v_x / 4]
14:         set d2_x [expr @L.end_x@ + $v_y / 4]
15:         set d2_y [expr @L.end_y@ - $v_x / 4]
16:         delete {@L@}
17:         create line @L.start_x@ @L.start_y@ $r_x $r_y -fill black
18:         create line $r_x $r_y $d1_x $d1_y -fill green
19:         create line $r_x $r_y $d2_x $d2_y -fill green
20:     }
```

01 行は生成されるトークンの名前「Name」を定義している。 02 行ではどのような構成要素からできているかを定義している。ここでは normal 要素の一本の線（line） L のみから成るということを示している。 04,05 行は制約「Constraint」の定義である。 04 行は書き換えがなされた後、根本の枝に再び生成規則が適応されないための制約、 05 行は再帰が終了するための制約である。 06 行は属性「Attribute」の定義だが、本生成規則では定義していない。 08 行からはアクション「Action」の定義である。 08 ～ 15 行は create のための座標計算で、実際には 16 行で構成要素の line を delete し、 17 行で Y 字の根本を、 18,19 行で Y 字の枝の部分の line を create している。

5.3.1 生成規則の定義

生成規則を定義する過程を述べる。

1 構成要素の定義を行う。

1-1 構成要素の例示入力、すなわちキャンバス上に実際に緑色で line を描画する。

1-2 line を選択し、 Rule メニューから VCW を選択すると、図 5.26のように例示入力図周辺がボックスで囲まれ、メインメニューが現れる。

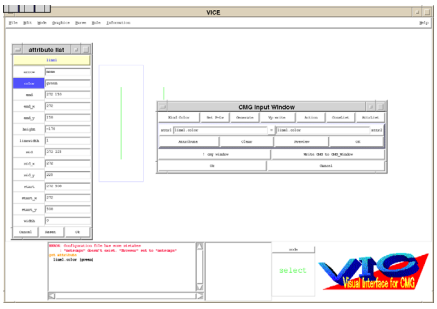

図 5.26: 構成要素の定義

2 制約の定義を行う。

2-1 L.color $==$ $\{stringgreen\}$ という制約の定義を行う。

2-2 定義ウィンドウを制約定義モードにし、制約メニューから「==」という制約を選択する。

2-3 構成要素をクリックし、メニューから属性リストを選択して表示させる。

2-4 属性リストから制約を課したい属性 — この場合 color — を選択し、これを定義ウィンドウのフィールドに入力する。 line は緑で描いたので 属性 color の値は green である。よって図 5.27のように第 1 、第 2 両方のフィールドに 属性 color を入力することで制約が定義できる。

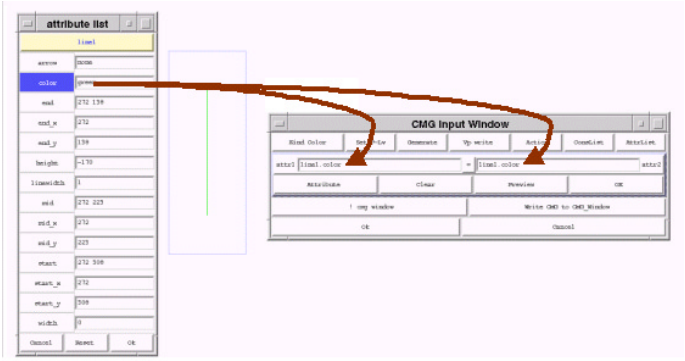

図 5.27: 制約の定義

2-5 L.height $>$ $\{integer - 5\}$ という制約の場合、第 2 フィールドには、直接「-5」と入力する。

3 アクションの定義を行う。

3-1 アクションの定義は、図 5.28のように、例示入力図の横に表示される図形に対する操作で行う。

3-2 line： L を delete するアクションを定義する。 delete したい line を例示入力図から選択し、メニューで delete を選択する。 delete 選択されたトークンは薄い色で表示される。

3-3 line を create するアクションを定義する。実際にキャンバスに、 create したい図形を実際に描画することにより、細かい属性などを簡単に定義できる。

3-4 アクション定義後の例示入力図は図 5.29のようになる。左右の図の差分でアクションが視覚的に把握できる。

属性は定義する必要がないので、以上で生成規則の定義は終了である。
これでフラクタルの樹の定義は完成である。

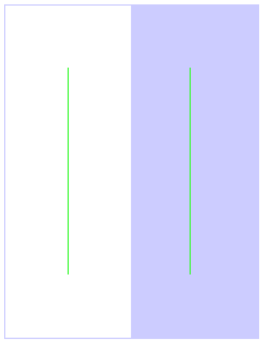

図 5.28: アクションの定義

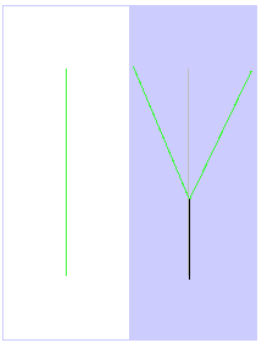

図 5.29: アクション定義後

5.4 アプリケーション：計算の木

数式を視覚的に表現し、処理するビジュアルシステムを作成する。計算の木とは図5.30のように２つのノードの値を引数とし、それらのノードが結線された円の演算子を作用させた結果を返すビジュアルシステムである。

計算の木は以下の生成規則より再帰的に定義される。

生成規則１． crcle の中心に text があれば、それは node である。

生成規則２． ２つの node が、中央に text をもつ circle に、それぞれ line で結ばれていたら、それは node である

これらの生成規則をより厳密に拡張 CMG の生成規則に従って記述すると以下のようになる。

```
01:  node(point mid, integer mid_x, integer value) ::=
```

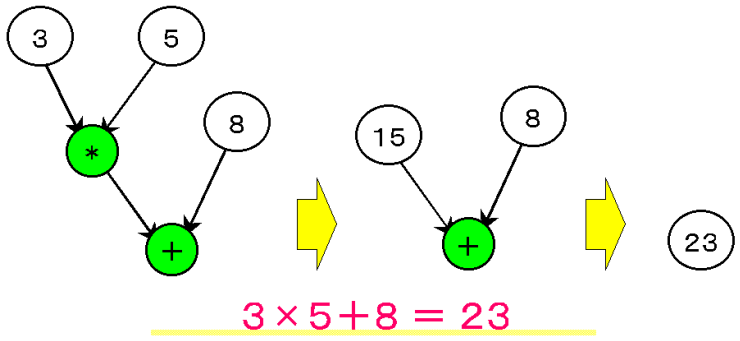

図 5.30: ビジュアルシステムの例：計算の木

```
02:     C:circle, T:text
03:     where {
04:        C.mid == T.mid
05:        C.innercolor == {string white}
06:     } and {
07:        mid = C.mid
08:        mid_x = C.mid_x
09:        value = {script.integer {set x @T.text@}}
10:     } and {
11:     }
12:
13: node(point mid, integer mid_x, integer value) ::=
14:    C:circle T:text
15:    exists N1:node, N2:node, L1:line, L2:line
16:    where {
17:       N1.mid == L1.start
18:       N2.mid == L2.start
19:       C.mid == L1.end
20:       C.mid == L2.end
21:       C.mid == T.mid
22:       C.innercolor == {string green}
23:       N1.mid_x < N2.mid_x
24:    } and {
25:       mid = C.mid
26:       mid_x = C.mid_x
27:       value = script.integer{expr @N1.value@@T.text@@N2.value@}}
28:    } and {
29:       delete {@N1@ @N2@ @L1@ @L2@}
30:       alter @T@ text @value@
31:    }
```

01 行目から 11 行までが生成規則 1 の定義である。 01 行は生成されるトークンの名前「Name」と属性を定義している。名前が node であり、 mid, mid_x, value という属性値を持ち、それぞれの型は point 型、 integer 型、 integer 型である。 02 行では図

形単語がどのような構成要素からできているかを定義している。ここでは全て normal 要素で、一つの円（circle） C と文字列（text） T から成るということを示している。04,05 行は制約「Constraints」を定義している。 04 行では円 C の中心座標： mid という属性と文字列 T の中心座標： mid という属性がほぼ一致しているという制約を定義している。 05 行では円 C の内部の色： innercolor という属性が white という文字列と一致する、つまり円 C の内部が白であるという制約を定義している。 07 行以下は生成規則が適用されたとき実行される部分である。 07 行から 09 行目までは定義中の図形単語にどのような属性値を与えるかという「Attributes」を定義している部分である。07,08 行は mid という属性と mid_x という属性が、それぞれ円 C の中心座標： mid 、中心の x 座標： mid_x と常に等しいということを示している。 09 行は value という属性の値を定義している。ここではテキスト T の文字列： text を integer として扱うことを示している。 10 行はアクション「Action」の定義だが、生成規則 1 では特に定義していない。

13 行からが生成規則 2 の定義である。生成規則 1 とほとんど同じであるので異なる点のみを説明する。まず 15 行であるが、これは構成要素として exist の構成要素を必要とすることを示している。 23 行の制約は ノード N1 の中心の x 座標： mid_x が ノード N2 の中心の x 座標： mid_x より小さい、すなわち ノード N1 が ノード N2 より左にあるということを示す制約である。この制約は左右の node を区別するためにある。この制約が無ければ、例えば「5 - 3」と「3 - 5」の区別が出来ない。 27 行では 2 つのノードの値 value を引数、テキスト T を演算子とした計算結果を、ノードの value とすることを定義している。 29,30 行ではアクションを定義している。 29 行で N1,N2,L1,L2, を delete し、 30 行で text： T の 属性： text を計算結果である value の値に書き換えている。

5.4.1 生成規則１の定義

生成規則 1 を定義する過程を述べる。

1 構成要素の定義を行う。

1-1 構成要素の例示入力、すなわちキャンバス上に実際に circle と text を描画する。

1-2 全体を選択し、 Rule メニューから VCW を選択する。すると図 5.31のように例示入力図周辺がボックスで囲まれ、メインメニューが現れる。

2 制約の定義を行う。

2-1 制約 C.mid == T.mid の定義を行う。

2-2 定義ウィンドウを制約定義モードにし、制約メニューから「==」という制約を選択する。

2-3 制約を課したい属性をもつ構成要素をクリックし、メニューから属性リストを選択して表示させる。

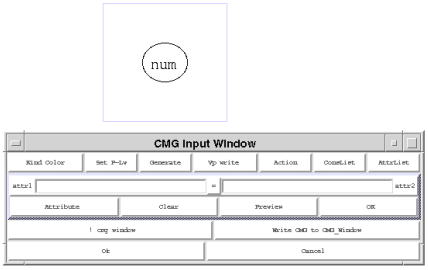

図 5.31: 構成要素の定義

2-4 属性リストから制約を課したい属性を選択し、これを定義ウィンドウのフィールドに入力する。図 5.32は circle：C の 中心座標：mid という属性を選択、第 1 フィールドに入力している。

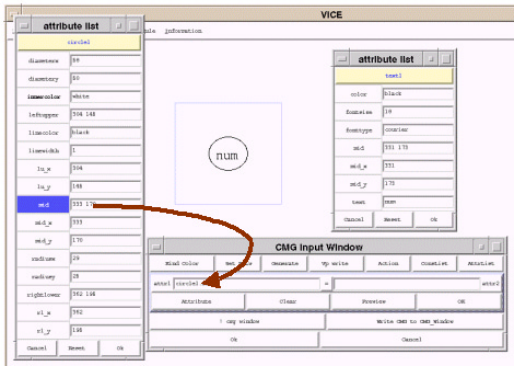

図 5.32: 制約の定義 1

2-5 制約 C.color == string white の定義を行う。

2-6 属性リストから制約を課したい属性 — この場合 color — を選択し、これを定義ウィンドウのフィールドに入力する。 circle は内部の色を白で描いたので 属性 color の値は white である。よって図 5.33のように第 1、第 2 両方のフィールドに 属性 color を入力することで制約が定義できる。

3 属性の定義を行う。

3-1 まず、属性 mid = C.mid の定義を行う。

3-2 定義ウィンドウを属性定義モードにし、第 1 フィールドに属性名 mid と入力する。

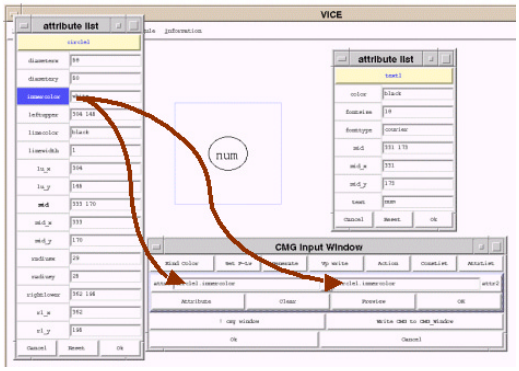

図 5.33: 制約の定義 2

3-3 例示入力図の circle をクリックし、メニューから属性リストを選択して表示させる。

3-4 属性リストから継承させたい属性 — 図 5.34の場合 mid — を選択し、これをダブルクリックで定義ウィンドウの第 2 フィールドに入力する。

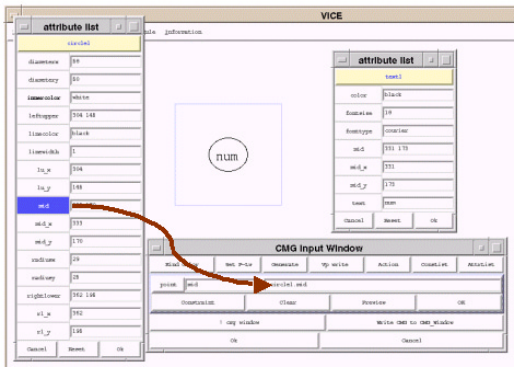

図 5.34: 属性の定義

3-5 他の属性も同様に定義する。

アクションは定義する必要がないので、以上で生成規則 1 の定義は終了である。

5.4.2 生成規則 2 の定義

生成規則 2 を定義する過程を述べる。

1 構成要素の定義を行う。

1-1 構成要素の例示入力を行う。 node は生成規則 1 に基づき circle とその中央に text を描画する。

1-2 全体を選択し、 Rule メニューから VCW を選択すると、例示入力図周辺がボックスで囲まれ、メインメニューが現れる。この際、既に定義された生成規則を用いて Spatial Parsing が行われる。すなわち生成規則 1 を用いて node が認識される。 図 5.35は、 circle と text が node と認識されていることを示している。

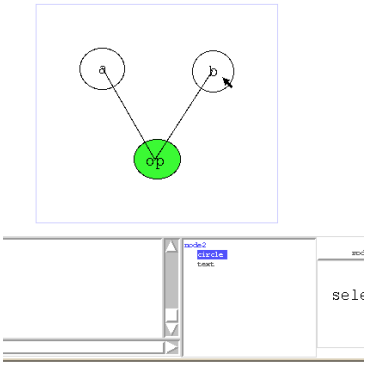

図 5.35: 構成要素の定義 1

1-3 必要な構成要素を exist 要素と定義する。図 5.36のように変更したい構成要素をクリックし、メニューから exist を選択する。

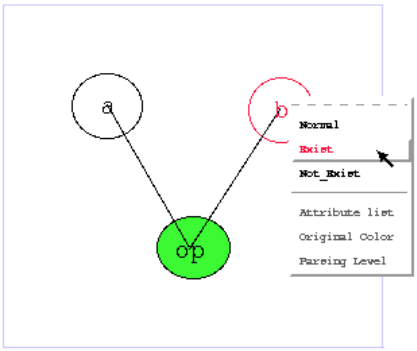

図 5.36: 構成要素の定義 2

2 制約の定義を行う。

2-1 今回は図形的な制約が多く必要なので制約の自動生成機能を使用する。メインメニューの「Generate」ボタンを押すと制約が自動生成される。

2-2 どんな制約が生成されたかは、図 5.37のような制約リストで確認できる。

2-3 制約リストで不必要な制約を無効にする。この際、現在ポインタが指しているリスト上の制約の対象となっている構成要素がバウンディングボックスに囲まれ例示入力図上に表示されるので、これを参照しながら行うと楽である。

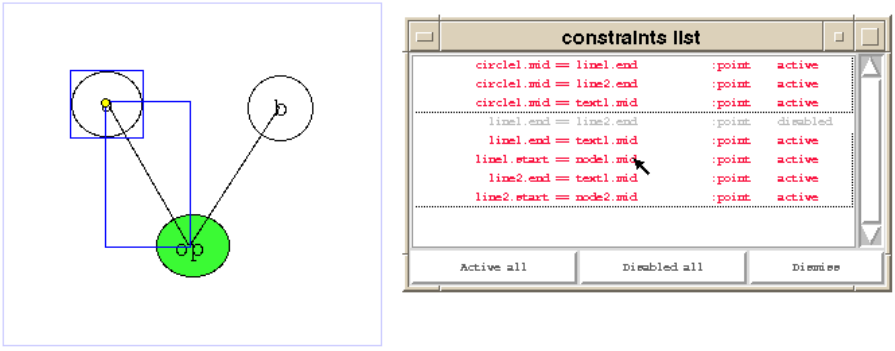

図 5.37: 制約の定義

2-4 足りない制約は定義ウィンドウを用いて手動で定義する。

3 属性の定義を行う。

3-1 生成規則 1 とほぼ同じなので省略。

4 アクションの定義を行う。

4-1 アクションの定義は、図 5.38のように、例示入力図の横に表示される図形に対する操作で行う。

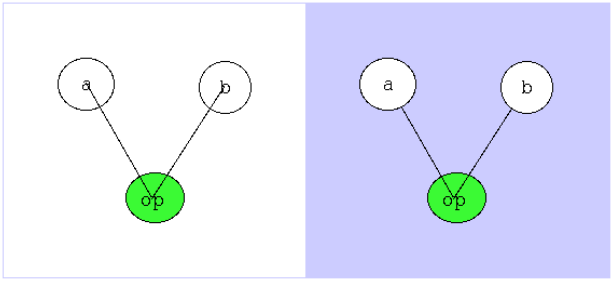

図 5.38: アクションの定義

4-2 N1,N2,L1,L2 を delete するアクションを定義する。図 5.39のように、 delete したいトークンを例示入力図から選択し、メニューで delete を選択する。 delete 選択されたトークンは薄い色で表示される。

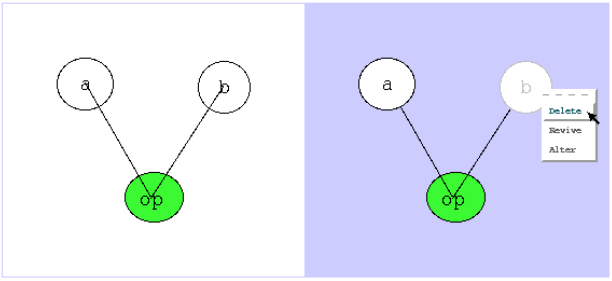

図 5.39: delete の定義

4-3 T.text を alter するアクションを定義する。変更したい属性をもつ構成要素をクリックして、メニューより属性リストを表示される。（図 5.40）

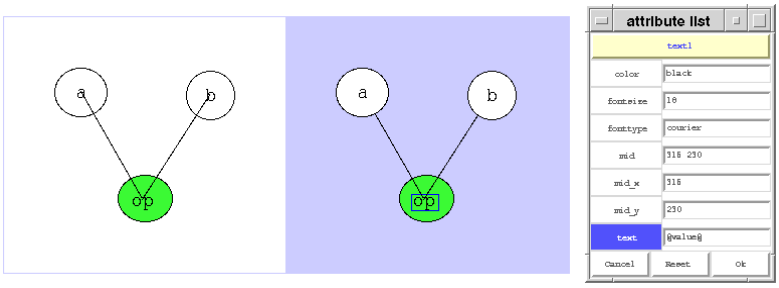

図 5.40: alter の定義

4-4 属性リストより変更したい属性の属性値を選択し、実際に変更する。

4-5 アクション定義後の例示入力図は図 5.41のようになる。左右の図の差分でアクションが視覚的に把握できる。

以上で生成規則 2 の定義は終了である。
これで計算の木の定義は完成である。

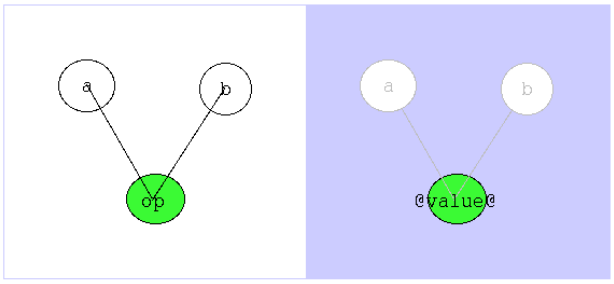

図 5.41: アクション定義後

第 6 章

評価

システム VIC の有効性を示すために評価実験を行った

6.1 実験内容

6.1.1 実験方法

例示入力図を用いた Spatial Parser Generator である VIC と、従来のテキスト記述による Spatial Parser Generator である恵比寿の 2 つのシステムを用いて、実際にビジュアルシステム：スタックの生成規則 4 を定義してもらい、その作業時間を比較する。

なお被験者には図 6.1に示すような生成規則 4 の内容を示したメモを手渡し、既に生成規則 1 、 2 、 3 が定義された状態、すなわち非終端記号の認識ができる状態のシステムで実験を行った。

6.1.2 実験環境

実験環境は以下のとおりである。

Machine: Sun Ultra1 (CPU: UltraSPARC)

OS: Solaris7

Tcl/Tk: wish8.0

これに付属のディスプレイ、キーボード、マウスを使用して表示、操作を行った。

6.1.3 被験者

本研究室の学生 8 人を被験者とした。

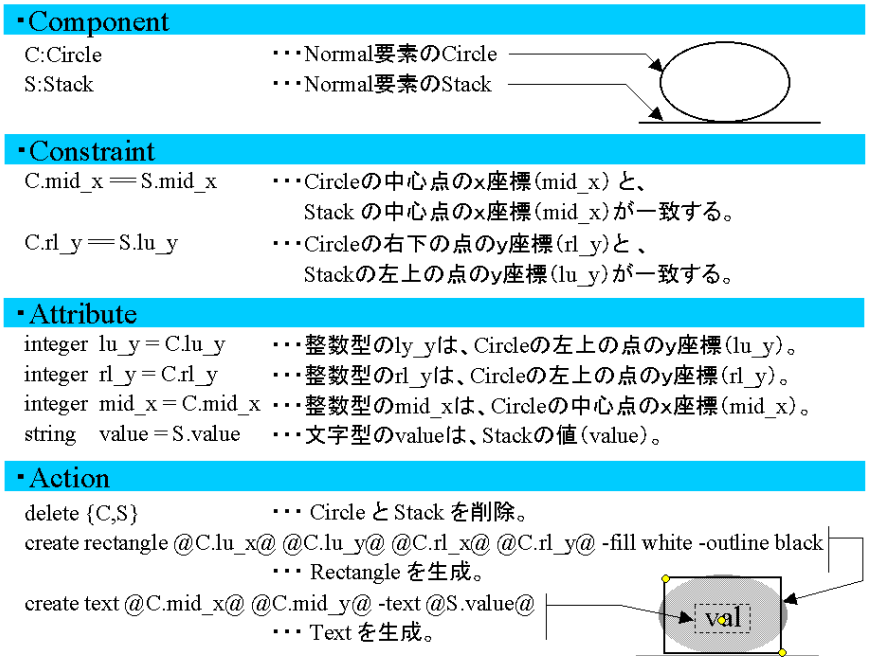

図 6.1: 生成規則４のメモ

6.1.4 学習効果への対応

人間には「慣れ」があるため、複数のシステムを比較評価する場合、より後で行った実験のほうが良い結果を出しやすい。これを学習効果といい、正確な評価を行うのに妨げとなる場合がある。

しかしながら、本実験で用いられる VIC はマウスによる図形操作が主な操作であり、恵比寿ではキーボードによる文字入力が主な操作である。目的は同じであるが、それを実現するための手法におけるコンセプトが全く異なるため、両者に共通する操作はほとんどない。したがって操作による学習効果の影響はほとんどないと思われる。

ただし、図形言語や生成規則に関する理解度などには学習効果が働く。したがって本実験では、あえて提案システムである VIC により不利な条件で、すなわち VIC →恵比寿の順で評価実験を行った。

6.2 結果

評価実験の結果を図 6.2に示す。縦軸が被験者、横軸が作業にかかった時間（単位は秒）である。

これらを VIC ／恵比寿ごとにまとめると表 6.1のようになる。

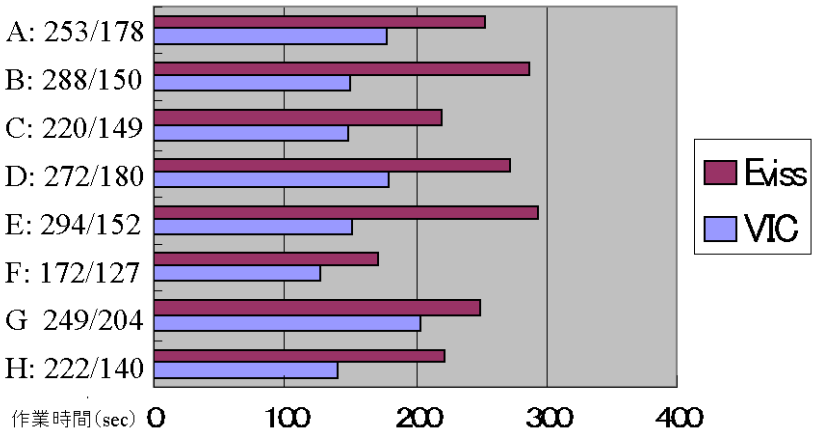

図 6.2: 結果

	Average:$\overline{X}$	Variance:s^2	Std_Deviation:s
VIC	160.00	554.25	23.54
恵比寿	246.25	1441.20	37.96

表 6.1: 結果のまとめ

6.3 考察

6.2の実験結果より、全ての場合において VIC を用いたほうが恵比寿を用いて生成規則を定義するより作業時間が短い、すなわち作業効率が良いと思われる。これを統計的に証明するために、実験結果を t 検定を用いて検定する。

まず平均値の差について検定を行うには、2 つの分散が均質であると言う前提条件が満たされていなければならない。そこでまず帰無仮説として $\sigma_1^2 = \sigma_2^2$ を立て、F 検定を用いて検定を行う。

$$F = \frac{s_2^2}{s_1^2} = \frac{37.96^2}{23.54^2} = 2.60$$

有意水準 5% の対応する自由度の臨界値はは 4.99 なので、$F = 2.60 < 4.99$ より帰無仮説を棄却することはできず、分散は等質であるとみなされる。

分散が均質であるという前提条件が満たされたので、今度は t 検定を用いて実験結果を検定する。まず 2 つの標本の母平均は等しい、すなわち $\mu_1 \geq \mu_2$ という帰無仮説を立てる。対立仮説は $\mu_1 < \mu_2$ として片側検定を行う。

$$t = \frac{\overline{X_2} - \overline{X_1}}{\sqrt{\frac{s_2^2+s_1^2}{n-1}}} = \frac{246.25 - 160.00}{\sqrt{\frac{37.96^2+23.54^2}{8-1}}} = 5.12$$

自由度 $2(n-1)=14$ の片側有意水準 5% の臨界値は 1.76 なので、 $t=5.12>1.79$ より、帰無仮説を棄却し、対立仮説を採択する。

したがって有意水準 5%、すなわち信頼係数 95% で VIC を用いたほうが作業効率が高いと言える。よって VIC の有効性が示された。

第 7 章

関連研究

我々の研究室では Spatial Parser Generator を研究している。我々はまず基本的な機能を持つ Spatial Parser Generator として恵比寿 [9][10][11][12][13] を作成した。しかし、前述の通り、恵比寿は１次元的なテキストで図形言語を定義するため直感的に理解しにくいという欠点があった。またテキストで記述するため、恵比寿におけるメタ文法である拡張 CMG を十分知っていなくては図形文法を定義できない。

そこで恵比寿をさらに発展させ、より使いやすく幅広いシステムに応用できるように２通りのアプローチを試みた。一つは「図形による図形言語の仕様記述」という点をより押し進め、完全に図形によって仕様記述を行うという方向である。もう一つは恵比寿に図形のレイアウト機能を導入し、図形を解釈しながら、繁雑になりがちな図を、よりインタラクティブに分かりやすく扱えるようにするという方向である。前者の考えに基づいたシステムとして、本論文で述べた VIC、後者の考えに基づいたシステムとして Rainbow[22][23] がある。 Rainbow ではレイアウト機能を制約の一部として実現している。制約の一部とすることで、ユーザが任意の形でレイアウトを行なうことが可能となっている。

同じく Spatial Parser Generator をもつシステムとして、 Penguins[24]、 SPARGEN[25] がある。 Penguins は CMG を用いて図形言語を定義しているが、その入力方法はテキストである。したがって恵比寿と同様の欠点をもつ。またアクションに相当する概念も無い。 SPARGEN は PLG の拡張である OOPLG を用いている。アクションのように解析した結果を用いて何らかの処理を行うことは可能であるが、解析後の図形間に意味的な関係を保存する機能はない。

テキストの Parser Generator として YACC(Yet Another Compiler Compiler)[26] がある。 YACC は Lex で字句解析したものを使い、構文解析を行う C のプログラムを自動生成するツールである。文法は BNF によく似た記述になっていて、構文解析のアルゴリズムには LALAR が用いられている。テキスト用のため、当然ビジュアル的な図形言語は扱えないが、アクションに相当するものとして任意の C プログラムを実行することができる。

制約を取り扱ったシステムに ThingLab[27] や TRIP3[28] がある。 ThingLab はオブジェクト指向に基づいて非終端記号を定義し、直接生成できる。しかし逆に複数の非終端記号をまとめて、新しい一つの非終端記号とすることはできない。 TRIP3 はアプ

リケーションのデータと、それを視覚化した図形オブジェクトとの対応関係を記述できる。これによりテキストから図形を生成、またその逆変換も可能としている。しかし、どちらも VIC と異なり、制約を動的に与えることはできない。

アクションのような図形の操作を可視化したシステムに Mondorian[29] や Chimera[30] がある。 Mondorian は基本的な図形を幾つか組み合わせたり引いたりして新しい図形を定義する過程を視覚化している。定義する過程の一番最初と最後を使ったコマンドとして表示する。 Chimera は図形エディタでの作業編集過程を視覚化している。フォントの変更や図形のコピーなどの作業の１つ１つをパネルにして並べて表示する。

第 8 章

結論

本論文では既存の Spatial Parser Generator が、２次元的な情報をもつ図形言語を１次元的なテキストで記述するために生じるいくつかの問題点を挙げ、これらの問題が、図形を用いて直接図形言語を定義するということで解決できる事について述べた。具体的には、まず恵比寿という Spatial Parser Generator の定義方法について考察を行い、その問題点について言及した。そして解決方法として、例示入力図という視覚的な表現を利用したより分かりやすい図形言語定義インターフェイスを提案した。これは図形を描くことによりその図形の特徴を抽出、汎化することで直接図形言語を定義する手法である。このとき最初に描かれる図形を例示入力図と呼ぶ。この手法ではテキスト記述を行わないので、メタ文法に対する厳密な知識がなくても生成規則の定義が直感的に可能である。さらに例示入力図を参照しながら作業を行うことにより、ユーザは自分の作業が図形言語のどの部分を定義しているのか把握しやすく、かつ操作結果を視覚的にフィードバックすることが可能となる。

さらに提案インターフェイスを持つ Spatial Parser Generator として VIC を作成し、実際に VIC を持ちいてビジュアルシステムを作成する例を示した。最後に評価として VIC の有効性を示すため、既に挙げたビジュアルシステムの作成課程を通して恵比寿と作業時間の比較実験を行なった。その結果、信頼係数 95% で VIC の有効性が示された。

謝辞

本研究を進めるにあたり、終始ご指導下さった指導教官の田中二郎教授に心から感謝いたします。また研究全般においてなにかと助けていただいた志築文太郎助手、飯塚和久氏にも感謝したいと思います。そして田中研究室の皆さんにはシステム作成にあたり多くの貴重な助言をいただきました。ここに感謝の意を表します。

参考文献

[1] Nan C. Shu. Visual Programming. *Van Nostrand Reinhold,* 1988.

[2] 田中二郎. ビジュアルプログラミング. ビジュアルインタフェース, pp65-78, 共立出版, 1996.

[3] Jiro Tanaka. PP:Visual Programming System for Parallel Logic Programming Language GHC. In *Parallel and Distributied Computing and Networks '97*, pp188-193, 1997.

[4] Nobuo Kawaguchi, Toshiki Sakabe, and Yasuyoshi Inagaki. TERSE:Term rewriting support enviroment. In *Workshop on ML and its Appriction,* pp91-100, 1994.

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

[6] 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,* pp34-41, 1986.

[7] 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,* pp254-259, 1991.

[8] 松岡聡, 宮下健. 例示による GUI プログラミング. ビジュアルインタフェース, pp79-97, 共立出版, 1996.

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

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

[11] 馬場昭宏, 田中二郎. Spatial Parser Generator の Tcl/Tk を用いた実装. 情報処理学会シンポジウムインタラクション '97 論文集, pp71-78, 1997.

[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. インタラクティブソフトウェア V, 日本ソフトウェア学会 WISS'98, pp135-140, 近代科学社, 1997.

[13] Akihiro Baba and Jiro Tanaka. Eviss: A Visual System Having a Spatial Parser Generatr. In *Proceedings of Asia Pacific Computer Human Interaction 1998 (APCHI'98),* pp158-164, 1998.

[14] 藤山健一郎. ビジュアルシステム恵比寿における視覚的表現を用いた制約の入力, ９８年度筑波大学卒業論文, 1998.

[15] Kenichirou Fujiyama, Kazuhisa Iizuka and Jiro Tanaka. VIC: CMG Input System Using Example Figures. In *Proceedings of International Symposium on Futuer Software Technology 99,* pp67-72, 1999.

[16] Kenichirou Fujiyama, Kazuhisa Iizuka and Jiro Tanaka. VIC: Spaital Parser Generator for Visual Systems using Example Figures. In *Proceeding of 1999 IEEE Symposium on Visual Languages (VL '99),* pp314, 1999.

[17] 藤山健一郎, 田中二郎. 例示入力図を用いた Spatial Parser Generator. 日本ソフトウェア科学会第 17 回大会論文集, 2000.

[18] Kim Marriot. Constraint Multiset Grammars. In *Proceedings of the 1994 IEEE Symposium on Visual Languages,* pp118-125, 1994.

[19] Sitt Sen Chok and Kim Marriot. Automatic Construction of User Interfaces from Constraint Multiset Grammars. In *Proceedings of the 1995 IEEE Workshop on Visual Languages,* pp242-249, 1995.

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

[21] Micheal Sannella. Constraint Satisfaction and Debugging for Interactive User Interfaces. Technical report, University of Wasington, 1994.

[22] 丁 錫泰, 田中二郎. ビジュアルシステム「恵比寿」におけるレイアウト制約の実現. 情報処理学会論文誌プログラミング Vol.40, No.SIG 10(PRO 5), pp76, 1999.

[23] 丁 錫泰. ビジュアルシステム生成系へのレイアウト制約の導入. 筑波大学大学院博士課程工学研究化博士論文, 2000.

[24] Sitt Sen Chok and Kim Marriot. Automatic Construction of Intelligent Diagrams of OMT. In *Proceeding the ACM Symposium on User Interface Software and Technology*, pp185-194, 1998.

[25] Eric J. Golin and Tom Magliery. A Compiler Generator for Visual Languages. In *Proceeding of the 1993 IEEE Symposium on Visual Languages*, pp314-321, 1993.

[26] 佐々政孝. プログラミング言語処理系. 岩波書店. 1989.

[27] Alan Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. In *ACM Transaction on Programming Languages and Systems,* Vol.3, No.4, pp353-387, 1981

[28] Satoshi Matsuoka, Shin Takahasi, Tomihisa Kamada and Akinori Yonezawa. A General Framework for Bidirectional Translation between Abstract and Pictorial Data. In *ACM Transaction on Information Systems* Vol.10, No.4, pp408-437, 1992.

[29] Henry Lieberman. Mondrian: A Teachable Graphical Editor. In *Watch What I do,* pp340-338, 1993.

[30] David Kurlander. CHIMERA: Example-Based Grapical Editing. In *Watch What I do,* pp270-290, 1993.