VIC：例示入力図を用いた Spatial Parser Generator

藤山 健一郎　　田中 二郎

ISE-TR-01-178

VIC：例示入力図を用いた Spatial Parser Generator

藤山健一郎　　田中二郎

〒 305-8577 茨城県つくば市天王台 1-1-1
筑波大学電子情報工学系田中研究室
TEL：(0298) 53-5165　　FAX：(0298) 53-5206
e-mail：{fuji,jiro}@iplab.is.tsukuba.ac.jp

1　はじめに

ビジュアルシステムの研究 [1][2][3][4] 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。しかし個々のビジュアルシステム毎に Spatial Parser を実装するのは困難で時間のかかる仕事である。そこで図形言語の仕様を定義することにより、この Spatial Parser を生成し、様々な図形言語を解析し、さらに実行することができる Spatial Parser Generator が開発されている。

従来の Spatial Parser Generator[5][6][7][8] 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 [9]。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、といった多くのパラメータで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、定義したい線を 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、必要な数多くのパラメータを容易に表現できる。

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。例示入力図とはその名の通り、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。そのため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。

我々は提案手法を実現したインターフェイスをもつ Spatial Parser Generator VIC [10][11][12][13] を開発した。本論文では、 VIC の使用方法について述べる。

2 Spatial Parser Generator VIC

VIC の実行画面は図 1のようになる。

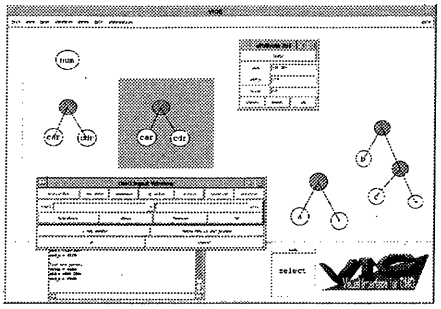

図 1: 実行画面

図形言語の仕様記述には CMG[14] にアクションの概念を付加した拡張 CMG[5] を用いていている。ユーザは拡張 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、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。

2.1 Eviss

Eviss とは我々が開発した Spatial Parser Generator である。 VIC は Eviss を拡張したシステムである。

Eviss のソフトウェア配布キットは WWW 経由で以下の URL から入手可能である。

`http://www.iplab.is.tsukuba.ac.jp/software-j.html`

2.2 VIC と Eviss の違い

VIC と Eviss の違いは次の通りである。

1. Eviss ではテキストを用いて図形言語の定義を記述する。そのため直感的に理解できな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。一方 VIC では図形を用いて直接図形言語を定義するので直感的に理解しやすく、また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。

2. Eviss では、図形言語の定義を行う定義窓と実際に図形言語の解釈を行う実行窓に分かれているが、VIC ではこれらを一つの窓に統一している。このためユーザは定義/実行モードの区別なくなり、スムーズな操作と、連続的かつインタラクティブな図形言語の定義を可能としている。

3 セットアップ

3.1 動作環境

現在以下の環境で動作を確認している。

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

3.2 ダウンロード

VIC のソフトウェア配布キットは WWW 経由で以下の URL から入手可能である。

```
http://www.iplab.is.tsukuba.ac.jp/~fuji/PUB/VIC/index.html
```

3.3 インストール

VIC は以下の順番にインストールを行う。

1. `Makefile` の中のマクロを書き換える。
   - VPG_DIR： VIC がインストールされているディレクトリのパス
   - TCL_DIR： 本配布キットで提供される Tcl のライブラリのパス
   - WISH： wish のコマンド名
   - TCLSH： tclsh のコマンド名
2. コマンドラインから `make` コマンドを実行する。そうすると `eviss` という名前のファイルができる。

3.4 起動と終了

VIC を起動するには、VIC 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する。% はプロンプトである。

```
% eviss
```

VIC を終了するには、「File」メニューから「Exit」を選択する。

4 使い方

4.1 基本的な流れ

1. キャンバスに定義したい図形言語の大まかな概要を描画する。
2. 描画した図形 — 例示入力図に対し様々な操作を行うことで、図形言語の仕様（構成要素、制約、属性、アクション）を定義する。
3. 仕様を定義すると、以降キャンバスに描画された図形が、図形言語として解析、実行される。

以上のサイクルを繰り返して Spatial Parser を作成していく。具体的にはビジュアルシステムの作成例（5節）を参照。

4.2 各手順の詳細

4.2.1 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て、直接描画する。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。

また、構成要素の種類の変更は、例示入力図から変更したい構成要素を直接クリックし、表示されるメニューから変更種類を選択することで行える。

4.2.2 制約の定義

例示入力図から制約を課す属性をもつ構成要素をクリックし、表示される属性リストから目的の属性を選択する。選択した属性を制約定義ウィンドウに入力する。制約を課する2つの属性と制約の種類を選択したら、決定ボタンを押してその制約を定義する。なお制約は以下の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。

定義した制約は制約リストで確認できる。また、制約リストで制約をクリックすることにより、その制約の有効／無効を自由に切替えられる。

4.2.3 属性の定義

属性は属性の型、属性名、属性値を計算するスクリプトを属性定義ウィンドウの各フィールドに入力することで定義する。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
- point ··· 2 次元座標

属性値を計算するスクリプトを記述する際に既にある構成要素の属性値を参照したい場合、その属性をもつ構成要素を例示入力図からクリックし、表示された属性リストから望む属性を選択し、Cut&Paste の要領でスクリプトフィールド入力できる。

また属性名フィールドで同様の操作を行うと、属性名が自動的に入力される。

定義した属性は属性リストで確認できる。また、属性リストで属性をクリックすることにより、その属性の有効／無効を自由に切替えられる。

4.2.4 アクションの定義

tcl のスクリプトを直接記述することで定義する。ただし、図形言語特有の操作である図形操作 — create（図形の生成）、alter（図形の変更）、delete（図形の削除）は例示入力図に対する例示操作で定義できる。

cleate は生成したい図形を実際にキャンバスに描画することによって定義する。alter は変更したい図形を例示入力図からクリックし、表示された属性リストの属性値を直接書き換えることで定義する。delete は削除したい構成要素を例示入力図から削除することによって定義する。

5 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。

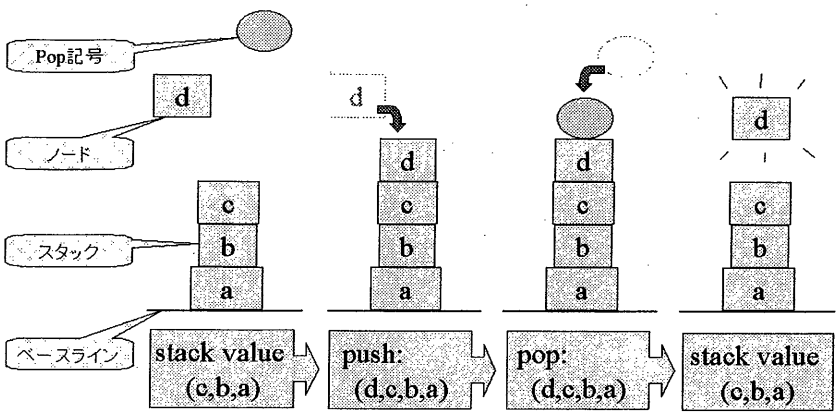

図 2: ビジュアルシステムの例：スタック

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2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
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:     }
```

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 3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。

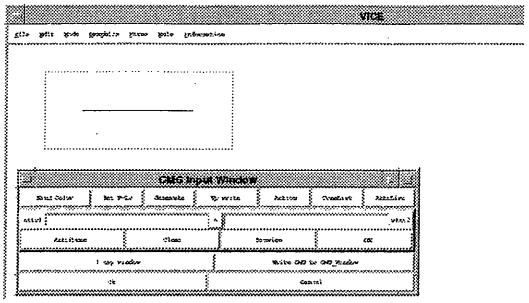

図 3: 構成要素の定義

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 4A）

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 4B 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。

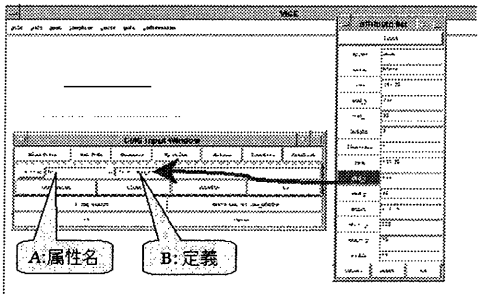

図 4: 属性の定義

2-5 OK を選択し、定義完了。

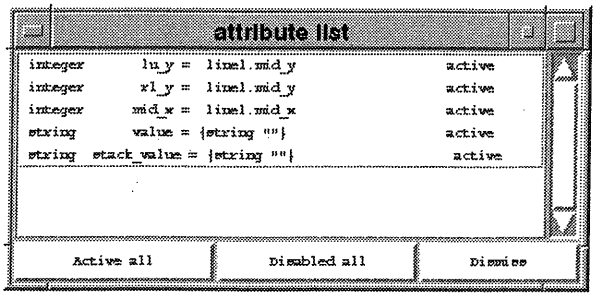

図 5: 定義した属性

2-6 他の属性も同様に定義する。図5は全ての属性を定義したところである。

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6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。

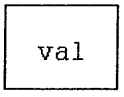

図 6: 構成要素の定義

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 7)

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。

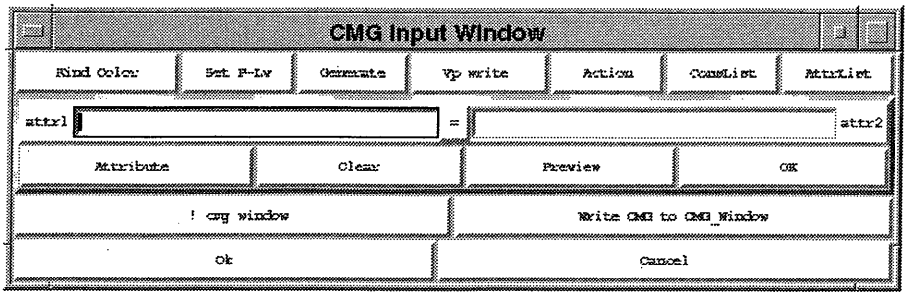

図 7: 制約の定義 1

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 8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。

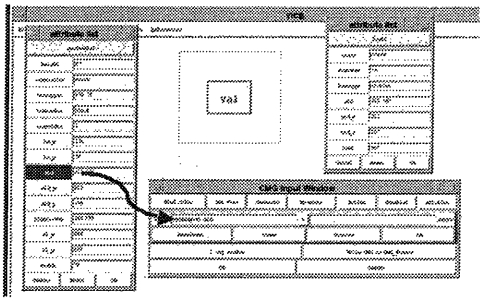

図 8: 制約の定義 2

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。

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 9)

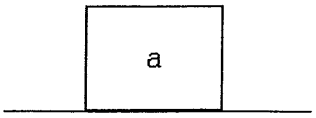

図 9: 構成要素の定義 1

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 10は、 rectangle と text が node と認識されていることを示している。

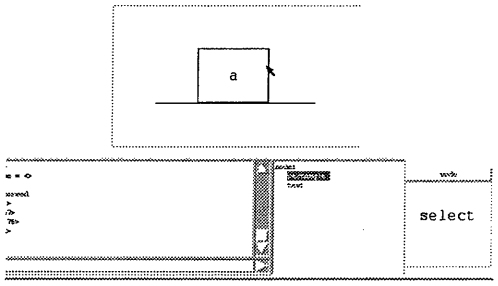

図 10: 構成要素の定義 2

1-3 構成要素 stack を exist 要素と定義する。図 11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。

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 12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。

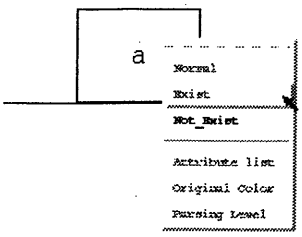

図 11: 構成要素の定義 3

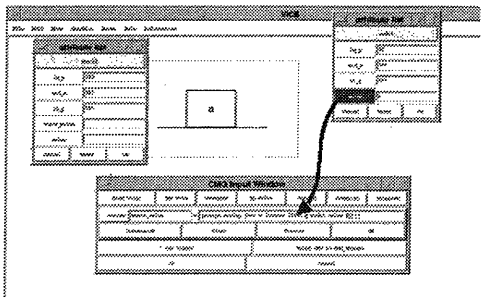

図 12: 属性の定義

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 13)

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。

図 13: 構成要素の定義 1

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 14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。

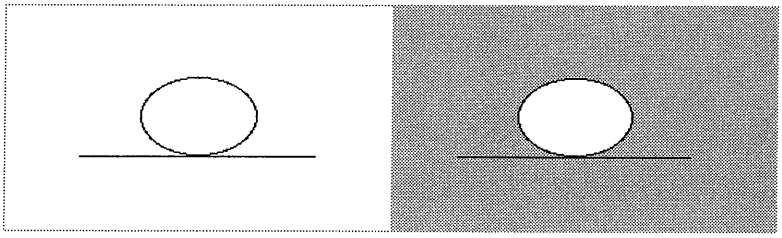

図 14: アクションの定義

4-2 circle：C、stack：S を delete するアクションを定義する。図 15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 16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 17）

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。

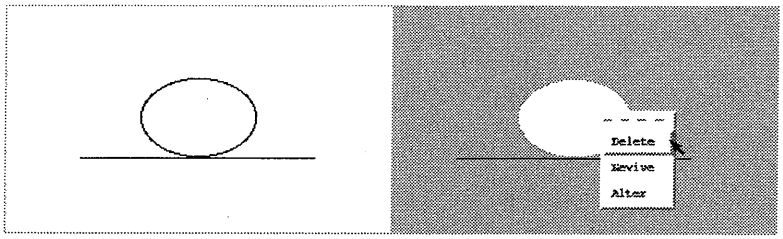

図 15: delete の定義

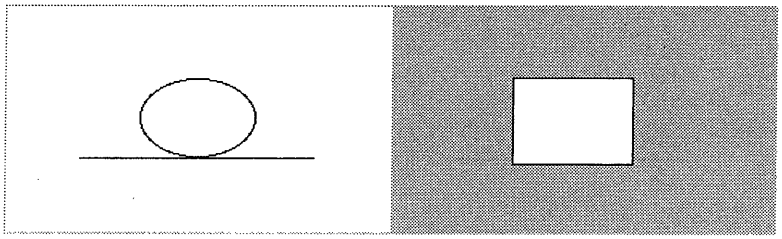

図 16: create の定義 1

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 18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 {
```

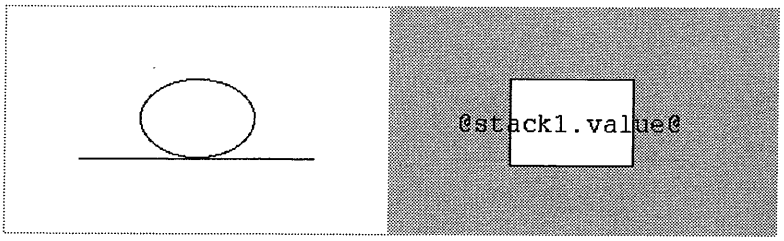

図 17: アクションの定義 2

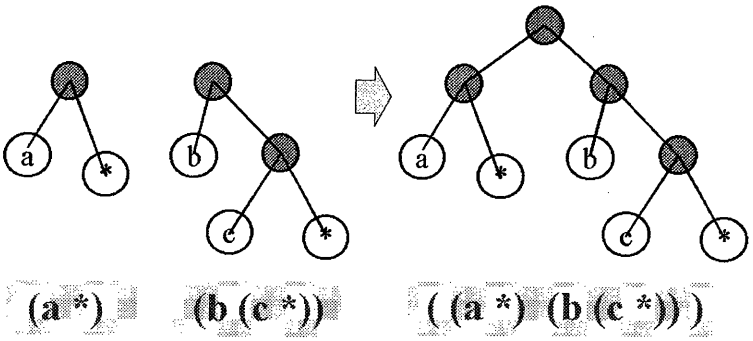

図 18: ビジュアルシステムの例：二分木リスト

```
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:     }
```

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19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。

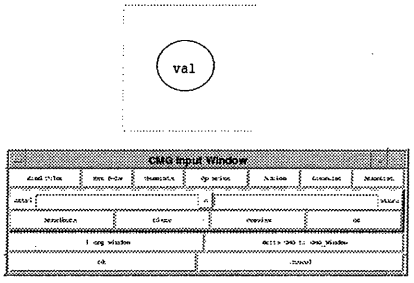

図 19: 構成要素の定義

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24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。

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。

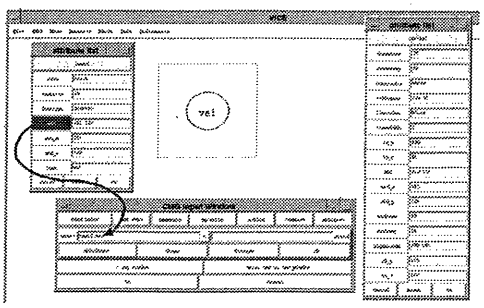

図 20: 制約の定義

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 21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。

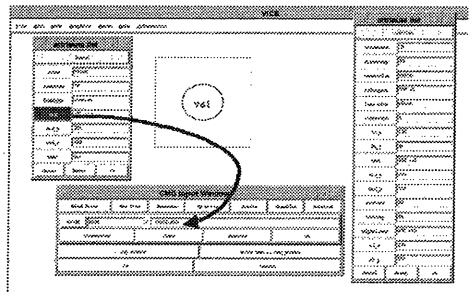

図 21: 属性の定義

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 22は、circle と text が node と認識されていることを示している。

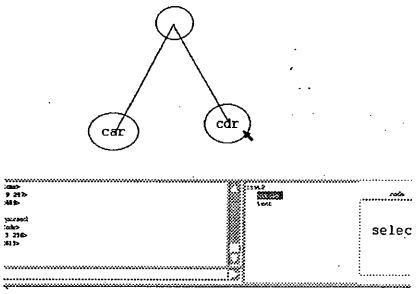

図 22: 構成要素の定義 1

1-3 必要な構成要素を exist 要素と定義する。図 23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。

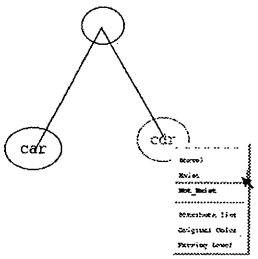

図 23: 構成要素の定義 2

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 24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。

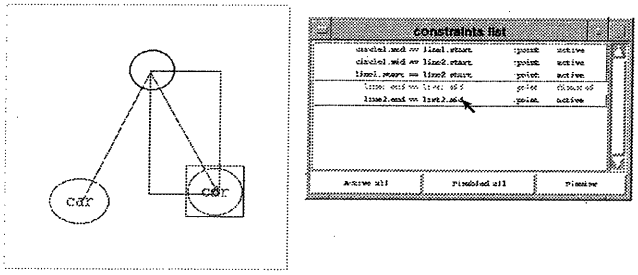

図 24: 制約の定義

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 25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。

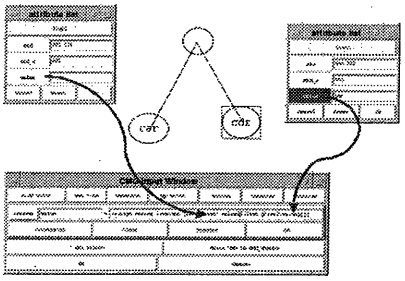

図 25: 属性の定義

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。

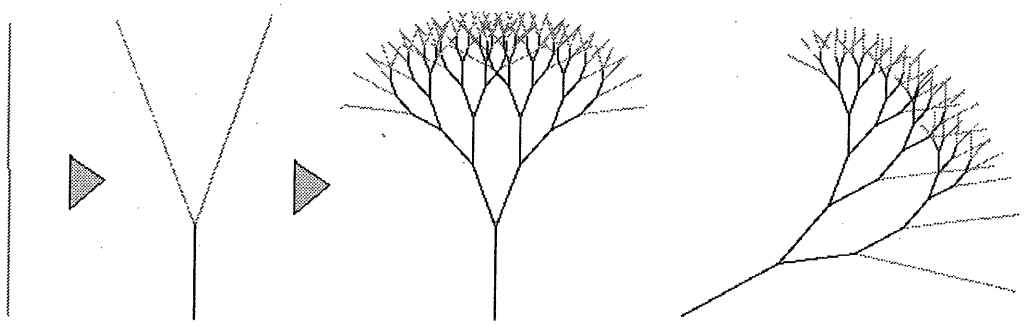

図 26: ビジュアルシステムの例：フラクタルの樹

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 26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:     }
```

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 27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。

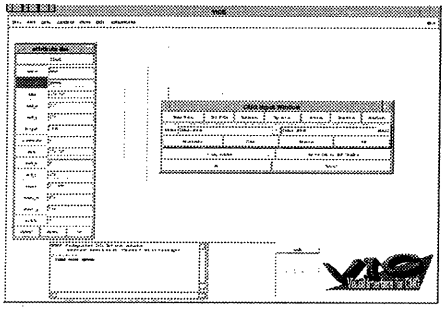

図 27: 構成要素の定義

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 28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。

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 29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。

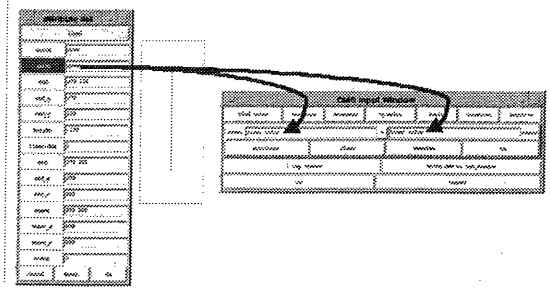

図 28: 制約の定義

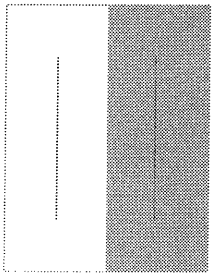

図 29: アクションの定義

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 30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。

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 31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。

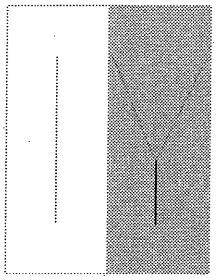

図 30: アクション定義後

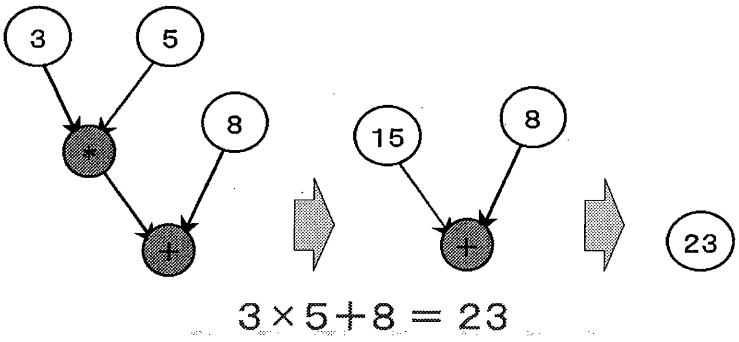

図 31: ビジュアルシステムの例：計算の木

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) ::=
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:     }
```

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 32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。

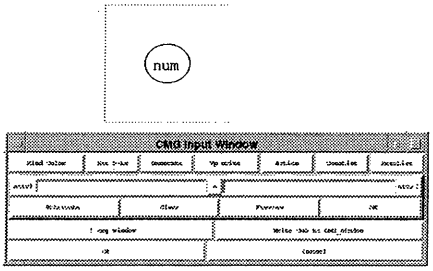

図 32: 構成要素の定義

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。

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 33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。

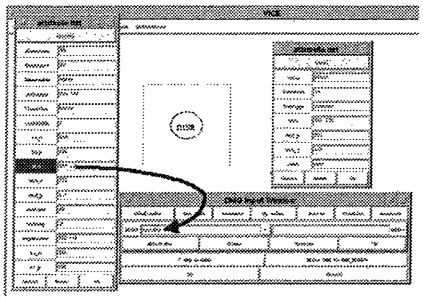

図 33: 制約の定義 1

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 34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。

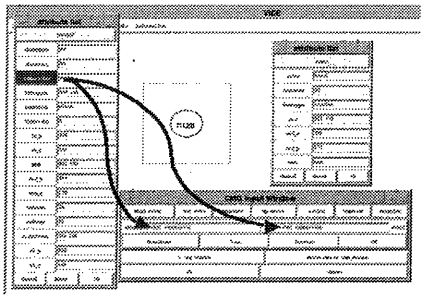

図 34: 制約の定義 2

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。

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 35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。

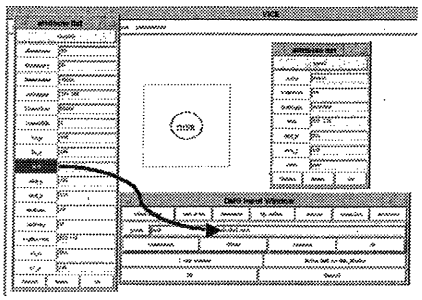

図 35: 属性の定義

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 36は、 circle と text が node と認識されていることを示している。

1-3 必要な構成要素を exist 要素と定義する。図 37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。

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 38のような制約リストで確認できる。

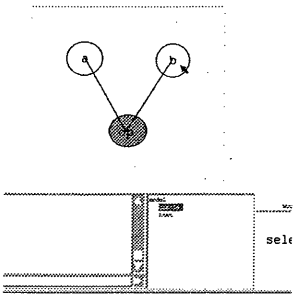

図 36: 構成要素の定義 1

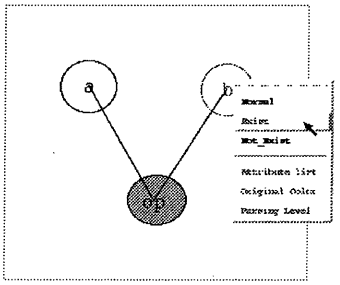

図 37: 構成要素の定義 2

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。

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 39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。

4-2 N1,N2,L1,L2 を delete するアクションを定義する。図 40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 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 41）

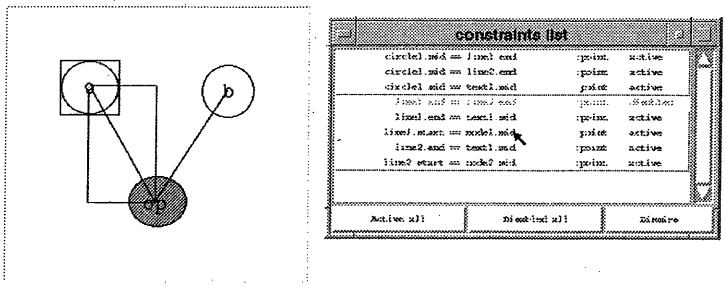

図 38: 制約の定義

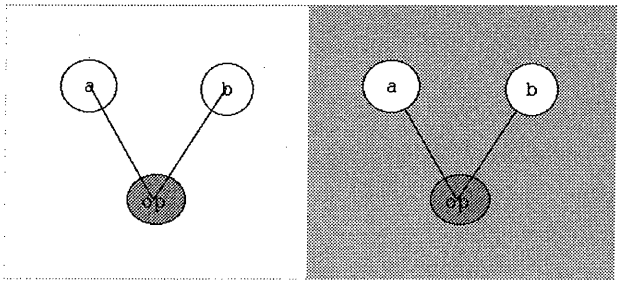

図 39: アクションの定義

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 42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。

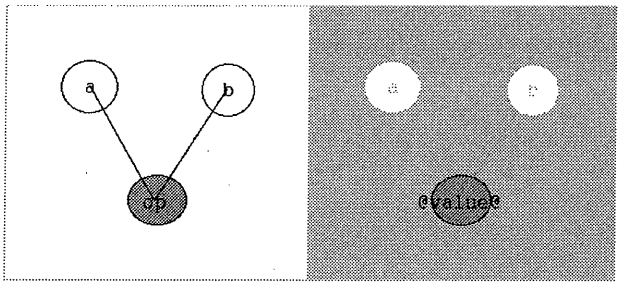

図 42: アクション定義後

以上で生成規則 2 の定義は終了である。

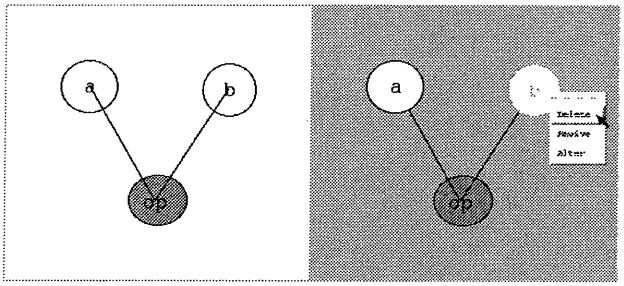

図 40: delete の定義

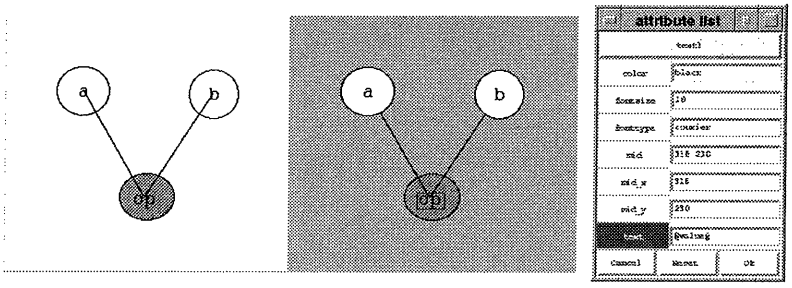

図 41: alter の定義

これで計算の木の定義は完成である。

6 まとめ

本論文では、例示入力図を用いた Spatial Parser Gnerator VIC の使用方法について述べた。

参考文献

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

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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, 9 8 年度筑波大学卒業論文, 1998.

[10] 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.

[11] 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.

[12] 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, 2001 年度.

[13] 藤山健一郎. 例示入力図を用いた Spatial Parser Generator. 筑波大学大学院博士課程工学研科修士論文, 2001.

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