第 3 章

システム「Rainbow」

3.1 「Rainbow」の概要

「Rainbow」では、図 3.1のような図形エディタを備えている。図形エディタは、図形言語の文法を定義する場合と実際に図形言語を実行する場合に用いられる。

「Rainbow」で生成規則を定義するときに、ユーザは一つの非終端シンボルとしたい図形を図形エディタに描く。次に、その図形を選んで CMG 入力ウィンドウ（図 3.2）を開く。そうすると「Rainbow」は図形から構成要素とそれらの属性間に成り立っている制約を CMG 入力ウィンドウに書き出す。CMG 入力ウィンドウは上から順番に非終端シンボルの名前、属性、制約、構成要素を書く欄になっている。ここにユーザは制約を修正し、また非終端シンボルの名前、属性を追加することにより、生成規則を定義していく（図 3.3）。

CMG 入力ウィンドウに書かれる制約としては、`eq`（equal）、`neq`（not equal）、`gt`（greater than）、`ge`（greater or equal）、`lt`（less than）、`le`（less or equal）、`vp_close` がある。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。制約解消系としては SkyBlue[23] を用いている。CMG 入力ウィンドウの中で、制約の書き方は「制約名 変数 1 変数 2」である。ここで、変数 1 と変数 2 は定義している非終端シンボルの構成要素になる終端シンボルもしくは非終端シンボルの属性を示している。

`eq` 制約は、二つの変数 1 と変数 2 の値が等しいことを意味する。この制約が一度成り立つと一つの変数の値が変っても制約解消系によって常にこの制約が成り立つように他の変数の値が書き換えられる。`neq` 制約は、二つの変数 1 と変数 2 の値

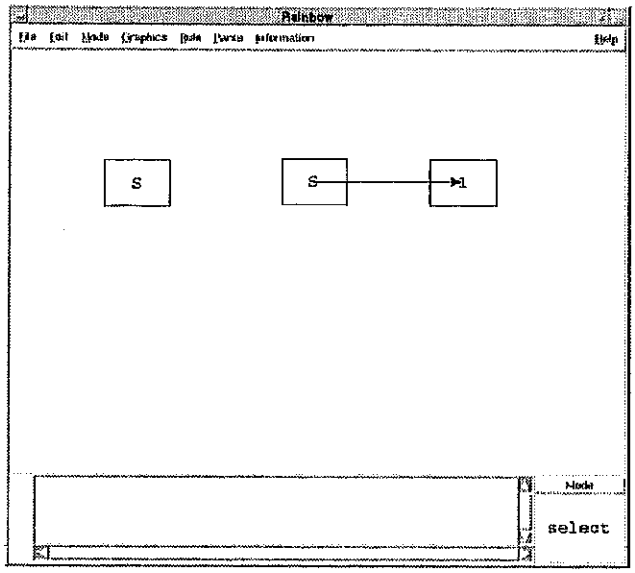

図 3.1: 「Rainbow」の図形エディタ

が等しくないことを表す。この制約は一度成り立っても一つの変数の値が変ると維持されない。gt 制約は変数 1 の値が変数 2 の値より大きいこと、ge 制約は変数 1 の値が変数 2 の値より大きいか等しいことを意味する。lt 制約は変数 1 の値が変数 2 の値より小さいこと、le 制約は変数 1 の値が変数 2 の値より小さいこか等しいことを表す。vp_close 制約は、変数 1 の値と変数 2 の値がある程度近いことを意味する。この制約が一度成り立つと eq 制約が課せられ、一方の値が変化すると他方も同じ値に変化する。

属性の参照は、「構成要素の種類.構成要素の順番.属性名」の形で行う。構成要素の種類は構成要素になる終端シンボルもしくは非終端シンボルが normal (exist) の構成要素だったら 0 (1) になり[1]、構成要素の順番は構成要素の種類の中で何番目の構成要素かを表す (0 から始まる)。例えば、normal の構成要素の 2 番目の構成要素の属性 mid (中心) を表す場合には「0.1.mid」のように記述する。

図 3.1の上左の図形はリスト構造の生成規則 1 を定義するため入力した図形で、上右の図形はリスト構造の生成規則 2 を定義するための図形である。上右の図形を選択して開かれた CMG 入力ウィンドウが図 3.2である。

[1]not_exist は 2、all は 3 になる。

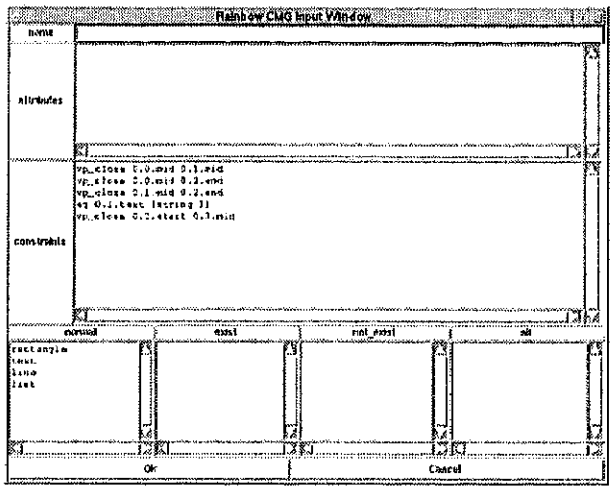

図 3.2: 「Rainbow」の CMG 入力ウィンドウ (1)

図 3.2の CMG 入力ウィンドウの構成要素の欄には、以下のように書かれる。

```
rectangle
text
line
list
```

また、制約の欄には以下のような制約が書かれる。

```
vp_close 0.0.mid 0.1.mid
vp_close 0.0.mid 0.2.end
vp_close 0.1.mid 0.2.end
eq 0.1.text {string 1}
vp_close 0.2.start 0.3.mid
```

ここで、1 行目は rectangle (0.0) の中心 (mid) が text (0.1) の中心とほぼ一致していることを表す。2 行目は rectangle の中心が line (0.2) の終点とほぼ一致していることを表す制約である。3 行目は text の中心と line の終点 (end) がほぼ一致していることを表す。4 行目は text の文字列 (text) が 1 であるという制約である。5 行目は line の始点 (start) か list (0.3) の中心がほぼ一致していることを表す。

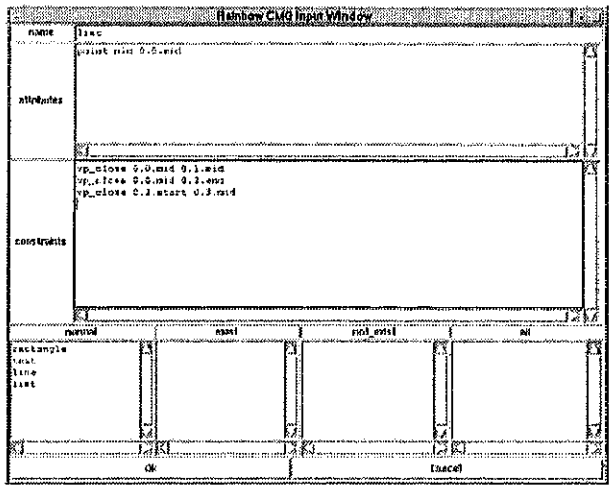

図 3.3: 「Rainbow」の CMG 入力ウィンドウ (2)

ユーザは、この CMG 入力ウィンドウの非終端シンボルの名前の欄に `list` と書き、非終端シンボルの属性の欄には `list` の属性 `mid` を `rectangle` の中心にするように書く。

```
point mid 0.0.mid
```

また、制約の欄には 1 行目、2 行目、5 行目の制約を選ぶ（図 3.3）。これで、リスト構造の生成規則 2 が定義される。

我々は、軟かいレイアウトの生成規則を定義するときにも、通常の生成規則を定義する場合と同じように、図形を用いて行う。まず、ユーザは解釈したい図形を図形エディタに描いてその図形またはその一部を選択し（図 3.4の上の図形）、「軟かいレイアウト CMG 入力ウィンドウ（図 3.5）」を開く。このウィンドウは上から各軟かいレイアウト制約の定数を設定するメニュー（Layout Constant Input）、非終端シンボルの名前、軟らかいレイアウト制約の名前 SC、再帰的に生成される非終端シンボルの名前 GS、`node` の構成要素の名前、`edge` の構成要素の名前を書く欄になっている。「Rainbow」では、図形を選択すると、`node` と `edge` の構成要素の名前の欄には、四角、円、直線などの終端シンボルを除いて非終端シンボルの名前がシステムにより自動的に以下のように書き出される（図 3.5）。

```
lNode
```

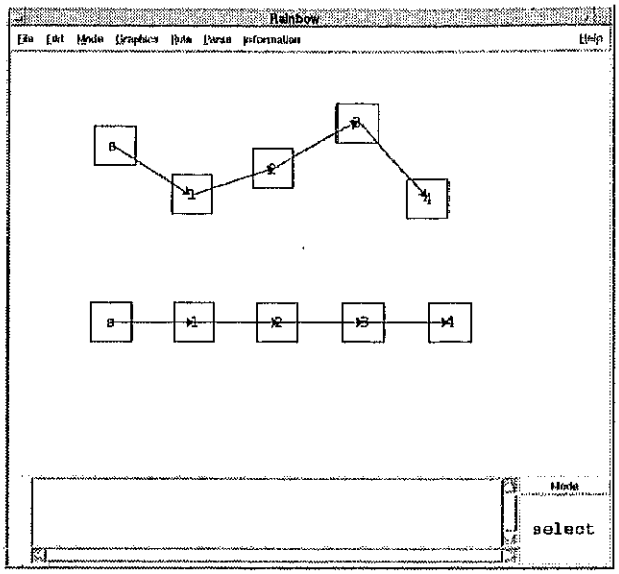

図 3.4: レイアウト前後のリスト構造

lEdge

list

図 3.5は、リスト構造（list）の構成要素ノード（lNode）とエッジ（lEdge）を認識する生成規則があらかじめ存在した場合に、開いた軟かいレイアウト CMG 入力ウィンドウである。

次に、ユーザは node や edge の構成要素にしたい非終端シンボルの名前を選び、非終端シンボルの名前、SC、GS を定義する。非終端シンボルの名前として listModel、SC として listStructure、GS として list を定義する。また、指定するレイアウトの定数を設定する（図 3.6）。レイアウトの定数を設定しなかった場合には、「Rainbow」で用意した定数の値が使われる。これで、軟かいレイアウトの生成規則が定義される。

図形言語の生成規則の定義が終わったら実際に図形を図形エディタに入力し、パーシングすることが可能になる。一般に「Rainbow」では、図形を解釈するモードとして自動モードと要求モードの二種類を用意している。新たな図形の入力があるたびにパーシングを行うのが自動モード、また、ユーザからの要求があったときのみにパーシングを行うのが要求モードである。

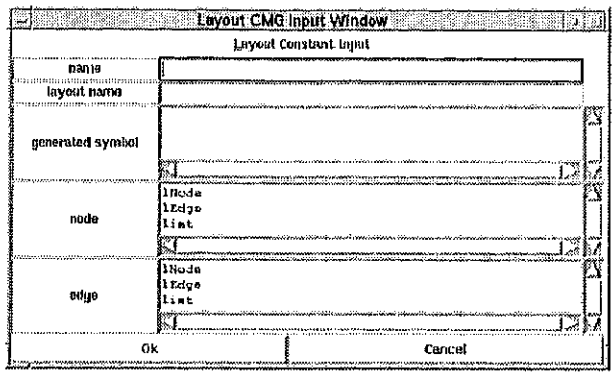

図 3.5: リストの軟かいレイアウト CMG 入力ウィンドウ (1)

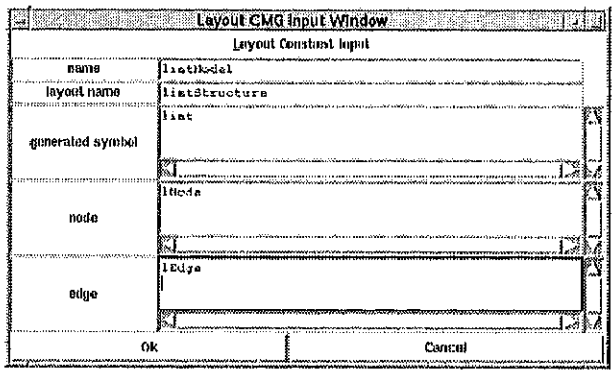

図 3.6: リストの軟かいレイアウト CMG 入力ウィンドウ (2)

定義した生成規則を用いて、図 3.4の上の図形を解釈すると下の図形のようにレイアウトされる。

3.2 「Rainbow」の構造

図 3.7は、「Rainbow」の構成要素を表している。「Rainbow」への入力は、通常の生成規則、レイアウトの生成規則、解釈したい図形で、「Rainbow」からの出力は、入力の図形をレイアウトした図形である。

図形言語の通常の生成規則と軟らかいレイアウトの生成規則は、「Rainbow」により内部表現が作られ、生成規則の集合に保存される。図 3.8は、生成規則の内部表現を表している。ただし、P は連想配列で、*p-id* は一つの生成規則に付けられ

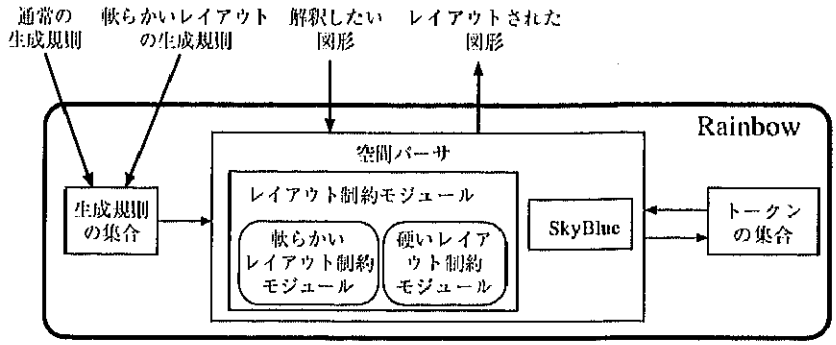

図 3.7: 「Rainbow」の構成図

名前	値
P(*p-id*.name)	非終端シンボルの名前
P(*p-id*.attrs)	非終端シンボルの属性のリスト
P(*p-id*.constraints)	構成要素の属性間の制約のリスト
P(*p-id*.negative_constraints)	ネガティブ制約のリスト
P(*p-id*.normal)	非終端シンボルの normal の構成要素のリスト
P(*p-id*.exist)	非終端シンボルの exist の構成要素のリスト
P(*p-id*.not_exist)	非終端シンボルの not_exist の構成要素のリスト
P(*p-id*.all)	非終端シンボルの all の構成要素のリスト

図 3.8: 生成規則の内部表現

た id である。

また、解釈したい図形を「Rainbow」に与えると、空間パーサにより図形を構成する全ての終端シンボルのトークンはトークンの集合に保存される。さらに、トークンの内部属性間に存在する制約は制約解消系 SkyBlue[23] に追加される。ここで、トークンの内部属性間に存在する制約として、例えば、四角を表すトークンの中心の x 座標は、四角の左上の x 座標と右下の x 座標を足して 2 で割るなどの制約がある。各トークンはトークンの属性を内部データとして持つ。トークンの属性は、属性の型と値を用いて新たな SkyBlue の変数を作成し、その SkyBlue の変数を値として持つ。トークンの属性値を SkyBlue の変数とするので、レイアウト制約を実現するためのレイアウト制約モジュールによりトークンの属性値が変ったときでも制約解消を SkyBlue に行わせることが可能である。

3.3 「Rainbow」の解釈アルゴリズム

CMG で記述された生成規則の集合として文法を記述し、解釈の対象となる図形を与えると通常は以下のように解釈が進む [3]。まず、ユーザが記述した生成規則と軟かいレイアウトの生成規則は、生成規則の集合 SCC に保存される。また、解釈の対象となる図形を構成する全ての終端シンボルのトークンは、トークンの集合 ParseForest に格納される。さらに、それらのトークンの内部属性間に存在する制約は制約解消系 SkyBlue に追加される。

実際の図形の解釈は、以下の [1] を行い、[1] が終了したあとで [2] を行うことにより実行する。

[1] ParseForest が変更されなくなるまで、SCC の各通常の生成規則に対して、構成要素の候補になれるトークンの組合わせリストを ParseForest から求めることを繰り返す。次に、各トークンの組合わせに対して、生成規則に記述された制約を満し、さらに硬いレイアウト制約が存在する場合にその解が存在するかを繰り返し検査を行う。もし、生成規則に記述された制約および硬いレイアウト制約を満す場合には、その生成規則が適用され、硬いレイアウト制約モジュールが実行される。そして、新たな非終端シンボルのトークンを ParseForest に挿入し、生成に用いられたトークンを ParseForest から削除する（トークンに「削除マーク」を付けるだけで実際には ParseForest から削除されない）。また、全ての制約を SkyBlue に追加する。

[2] SCC の各軟かいレイアウトの生成規則に対して、GS（再帰的に生成される非終端シンボルの名前）が ParseForest に存在するかを検査し、GS の構成要素が GS を持たなくなるまで繰り返し検査し、node や edge の構成要素のマルチセットを求める。

求められた node や edge の構成要素のマルチセットに SC が与えられ、レイアウトが行われる。さらに、S（新たに生成された非終端シンボルの名前）のトークンは ParseForest に挿入される。

図 3.9に、[1] と [2] を分かりやすくまとめた。

```
repeat
  foreach 通常の生成規則
    トークンの組み合わせリストを求める
    foreach トークンの組み合わせ
      if 通常の制約と硬いレイアウト制約を満たすか then
        硬いレイアウト制約のモジュールの実行
        新たに生成されたトークンをトークンの集合に追加
        生成に用いられたトークンをトークンの集合から削除
        制約を SkyBlue に追加
      end if
    end foreach
  end foreach
until トークンの集合が変更しない

foreach 軟らかいレイアウトの生成規則
  if GS がトークンの集合に存在するか then
    repeat
      node や edge の構成要素のマルチセットを求める
    until GS の構成要素が GS を持ったなくなる
    軟らかいレイアウト制約を与える
    新たに生成されたトークンをトークンの集合に追加
  end if
end foreach
```

図 3.9: 解析アルゴリズム

3.4 レイアウト制約モジュール

3.4.1 軟らかいレイアウト制約モジュール

軟かいレイアウト制約モジュールの実行により軟かいレイアウト制約の処理が行われる。スプリングモデル制約モジュールでは、GS の構成要素が GS を持ったなくなるまで繰り返し検査し、node や edge の構成要素のマルチセットを求め、図形要素間のグラフ構造を得る。グラフ構造を用いて隣接するノード間に働く力 f_s、隣接しないノード間に働く力 f_r が求められる。

マグネティックスプリングモデル制約モジュールでは、スプリングモデルに必要なグラフ構造に加えて、それぞれのエッジの種類を決めることとそのエッジが磁場に対して指す方向が必要である。これらを用いて f_s、f_r、エッジが受ける回転力 f_m が求められる。

リスト構造制約モジュールでは、GS の構成要素が GS を持ったなくなるまで繰り返し検査し、node や edge の構成要素のマルチセットを求め、図形要素間のグラフ構造を得る。ヘッドのノード（テキストとして s を持つノード）を決められた位置に配置し、あらかじめ与えられた順序に従いヘッドのノードの平行線上にノードを左から右に並べる。ノードの y 座標はヘッドのノードの y 座標と同じであり、ノードの x 座標は連続するノードの間を一定の間隔に開けるようにする。

tree 制約モジュールは、GS の構成要素が GS を持ったなくなるまで繰り返し検査し、node や edge の構成要素のマルチセットを得て、根ノード、ノード間の親子関係や兄弟関係などの情報を求める。次に、これらの情報を用い、Walker の一般木描画アルゴリズム [22] によってノードの最終座標を計算を行う。このアルゴリズムでは、木の根となる親ノードは描画の最上の位置に配置され、親ノードの各子ノードは下方の階層に対応した平行線上に、あらかじめ与えられた順序（後順）に従い左から右に並べられる。

3.4.2 硬いレイアウト制約モジュール

layout_eq 制約モジュールでは、このモジュールの引数になる変数 2 の値として変数 1 の値を代入し、変数 1 と変数 2 に SkyBlue の eq 制約を与える。eq 制約は、一度成り立つと一つの変数が変っても SkyBlue によって常にこの制約が成り立つ

ように他の変数の値を変った値に等しくする。

layout_dist 制約モジュールでは、このモジュールの引数になる変数１の値に変数３の値を足して変数２の値として代入し、SkyBlue の plus 制約を与える。plus 制約は、一度成り立つと一つの変数が変っても常にこの制約が成り立つように他の変数の値を変えて一定の距離を維持させる。

3.4.3 「Rainbow」と恵比寿の比較

恵比寿の図形エディタは、図形言語の文法を定義する定義ウィンドウと実際に図形の解釈を行う実行ウィンドウに分れていたが、「Rainbow」では一つのウィンドウに統合した。なぜなら、ユーザは文法を定義しながら図形を描いて文法の正しさを検査することの繰り返しによって文法を定義していくので、文法の定義モードと図形の実行モードを一緒にする方が便利である。そうすれば、一つの非終端シンボルとしたい図形の中に他の生成規則によって定義された非終端シンボルを認識することができる。また、複雑な図形を表示するのに画面の空間を有効に使える。

「Rainbow」では、恵比寿が扱う制約の他にレイアウト制約を実現した。特に、軟かいレイアウト制約の導入により、図形の全体を解釈しながらインタラクティブにバランスよくレイアウトすることが可能になっている。