インタラクティブシステムにおける空間解析器の研究

工学研究科

筑波大学

2004 年 3 月

飯塚 和久

概 要

我々は、オブジェクト図やフローチャートなど多くの図式を利用している。これらの図式を図形における 1 つの言語ととらえ図形言語と呼ぶ。これら図形言語は、矩形や直線などの基本図形がある規則に従って配置されたものである。この規則をテキストのプログラミング言語の文法と同様に図形文法として定めることができる。この図形文法に従って入力された図式を解析するのが空間解析器である。図形言語を図形文法で定義することによって、容易に空間解析器を作成できる。また、図形文法を修正するだけで簡単に空間解析器の動作を変更することも可能となる。

これまで、いくつかの空間解析器が提案されてきたが、すでに描かれた図式を解析する静的な解析に主眼が置かれていた。一方、図式は計算機上でインタラクティブに作成される。ユーザの入力に応じて動的に解析することができれば、その解析結果を即座にフィードバックすることが可能である。たとえば、レイアウト機能や図形の書き換えなどが実現できる。このような動的な解析を行う空間解析器は、恵比寿や Penguins などで実現されていた。しかし、これらの解析器では、実用上の 2 つの大きな問題があった。1 つは解析速度の問題である。入力された図形の数が多くなった場合に解析速度が低下する問題があり、インタラクティブな解析が困難であった。もう 1 つの問題は、図式に対するインタラクション機能の不足である。特定図形の上にマウスを重ねるとハイライトする機能や、解析結果に基づくモードの切り替え機能などは、従来の恵比寿や Penguins では提供が困難であった。本論文では、これらの問題を解決する手法を提案する。

解析速度の問題に対しては、図形文法の規則における条件（制約条件）を有効に利用し、探索範囲を限定させた解析を実現する。これは、制約条件の依存関係に基づいた “制約条件グラフ” を利用する。この制約条件グラフを利用した解析を行うことで高速な解析を実現する。さらに、前処理の導入や、図形の属性に関するテーブルを利用し高速な解析を実現した。これらの手法により、解析に必要な計算量のオーダを下げることができ、多くの場合、$O(n)$ から $O(n \log n)$ の計算量で解析ができるようになった。

インタラクション機能の不足の問題に対しては、図形エディタと空間解析器を統合して、マウスや GUI コンポーネントの情報を図形文法の枠組みで扱う手法を提案する。また、このときの処理を簡易に記述するために、図形文法の 1 つである CMG に対して拡張を行った。これにより、図式を定義する図形文法と同じ枠組みで簡単に機能の拡張ができるようになる。さらに、イベント処理や制約解消系の利用などの一連の処理の記述が容易に行えるようになった。

これらにより、インタラクティブシステムにおける実用的な空間解析器を実現することができるようになった。

目 次

図目次

表目次

第1章　序論

本章では、まず、図式処理システムで利用される空間解析器について述べる。そして、これまでのシステムにおける問題点を明らかにし、本研究の目的について示す。

1.1　図式処理システムと空間解析器

図式は、物事の関係を分かりやすく示す手段として、我々の身の回りで広く利用されている。例えば、組織図や家系図、数式、フローチャート、E-R 図、状態遷移図、オブジェクト図など、様々な図式が存在し、それぞれの分野で重要な役割を果たしている。

図式を計算機上で扱う “図式処理システム” は、扱う図式によって種々のシステムがある。これらシステムは、様々な種類の図式を扱うことができる一般化されたシステムと、ある特定の種類の図式のみを扱うシステムの 2 つに大別できる。

前者の例としては、Tgif[1)]や xfig[2)]などのエディタが挙げられる。これらは、矩形や線などを組み合わせて任意の図式を描画するために用いられる。どのような図式でも扱える半面、複雑な図式を入力するには手間がかかる。たとえば、矩形と線をつなげて描いた場合を考えてみる。これらの位置を変えるために矩形を移動させた際は、線も同時に動いて矩形と線がつながった状態を維持して欲しい。しかし、ユーザが明示的にグループ化などの操作をしない限りこのようなことは実現されない。これは、システムがどのような図式を描いているのか把握していないため、ユーザの意図が理解できないからである。

後者の例としては、Microsoft 数式エディタが挙げられる。これは、数式といった特定の図式を描画するための専用のシステムとなっている。ユーザが入力するのは数式であると分かっているので、それに対応したインタフェースを提供することができる。入力すべき位置は “スロット” で示すことにより、ユーザは決められたコンポーネントをメニューなどから選んでいくだけで図式の作成が行える。システムは、ユーザが入力したものが何であるか把握しているため、その図式に応じて大きさや位置を調整するレイアウト機能を提供することもできる。しかし、このようなシステムでは編集手法に制限があり、特定の位置にしか図形が入力できなかったり、入力の順序が固定されていたりする。

このような問題を解決するため、一般化された図式処理システムの上に、図式を解釈する機能を加えた “文法指向” の入力操作が可能となる枠組みが提案されている。一般の図形エディタの機能に加えて、さらに図式の解釈を行うことで、その図式の構造を把握し、レイアウトなどの図式の状態に応じた処理を行うことができる。この図式の解釈を行うソフトウェアが空間解析器である。

[1)] `http://bourbon.usc.edu:8001/tgif/`

[2)] `http://www.xfig.org/`

1.2　これまでのシステムの問題点

これまでに、空間解析器を利用した図式処理システムがいくつか提案されているが、これらのシステムでは実用上の 2 つの大きな問題があった。

第 1 の問題は、解析の速度に関する問題である。入力された図式における図形の数が多くなった場合に、解析が非常に遅くなっていたため、リアルタイムに処理を行うことが困難であった。インタラクティブなシステムにおいては、ユーザの入力に応じて即座に解析を行う必要がある。また、解析だけでなく、レイアウトを行うための制約解消系などを同時に動作させる場合があり、解析はできるだけ高速に行われることが期待される。しかし、これまでの解析手法では、実用的な速度を得ることができなかった。

第 2 の問題は、図式に対するインタラクション機能の不足である。これまでの空間解析器では、レイアウト機能を提供したりしているものはあった。このような図式に対する機能はある程度提供されてきたが、図式処理システムとして求められる図式に対する入力や編集のインタフェース、つまり図式に対するインタラクション機能は十分ではなかった。これまでのシステムでは、図形の作成、移動、変形などの標準的で一般的な図形エディタの機能のみ提供されていた。しかし、空間解析器を利用することで図式の構造や意味を捉えることができるので、これを利用した図式に対する編集機能が求められる。これまでのシステムでは、このような機能を実現しようとすると、個別にプログラムを作成する必要があり、図形言語を定める図形文法の定義に加え、別のプログラミング言語を用いて記述しなければならず不便であった。また、空間解析器の解析結果を得るために煩雑な手続きが必要であった。

1.3　本研究の目的

本研究では、これら 2 つの問題を解決し、インタラクティブシステムにおける実用的な空間解析器を実現することを目的とする。

解析速度の問題に対しては、“制約条件グラフ” を利用し、制約条件で組合せを随時検査する探索手法を示す。また、訪問の分岐を回避するための前処理の導入や、属性値テーブルを活用して適合する組合せを効率的に検索する手法を提案する。これら手法により、入力図形数に対してスケーラブルな解析が可能となり、インタラクティブシステムにおける実用的な解析速度を持つ空間解析器を実現する。

インタラクション機能が不足している問題に対しては、図形エディタと空間解析器の統合を提案する。このために、図形文法の枠組みでマウスやボタン等の情報を扱う特別な終端記号を導入する。また、イベント発生時や図形認識の際における処理の記述を可能とする図形文法の拡張を示す。これらにより、図形文法の枠組みを利用してシステムの処理の記述を統一的に実現し、インタラクション機能の拡張が容易に実現できる。

1.4　本論文の構成

本論文の構成は次のようになっている。まず、第 2 章では、準備として空間解析器に関する用語の解説と紹介を行う。第 3 章では、インタラクティブな図式処理システムを記述するための図形文法である CMG について述べる。そして、第 4 章では、解析の速度に関する問題を解決するための空間解析器の高速化手法を提案する。第 5 章では、インタラクション機能が不足している問題について議論し、これを解決するための手法として図形エディタと空間解析器の統合を提案する。最後に、第 6 章でまとめる。

第2章　準備

本章では、準備として空間解析器に関する用語の解説と紹介を行う。図形言語、図形文法、空間解析器生成系と生成される空間解析器について述べる。最後に、インタラクティブシステムにおける空間解析器について議論する。

2.1　図形言語とその解析

フローチャートやオブジェクト図などの図式は、ある決まった書き方に従って描かれ、それらの意味づけがなされている。このような図式を、図形における 1 つの “言語” ととらえ**図形言語**と呼ぶ。これら図形言語は、矩形や線、円などの基本図形が、ある規則に従って配置されたものであると考えることができる。この規則をテキストのプログラミング言語の文法と同様に**図形文法**として定めることができる。つまり、図式の構造や、その図式の表す意味を定義するのである。図形文法を用いて図式、すなわち図形言語を定義することにより、入力された図式が正しい図形言語かどうか判断したり、その図式の構造や、表現している内容などを取得したりすることが可能となる。計算機を用いてこの解析を行うシステムは、図形間の空間的な配置から解析を行うので**空間解析器**と呼ばれる。このような空間解析器を、個別の図形言語ごとにプログラマーが作成するのは、非常に手間がかかる作業であり現実的ではない。そこで、図形言語を図形文法で定義することによって、自動で空間解析器を生成する**空間解析器生成系**と呼ばれるシステムを利用する。空間解析器生成系を利用することにより、手軽に空間解析器を利用できるようになる。また、ルールに基づく宣言的な記述が行える。さらに、図形文法を修正するだけで簡単に空間解析器の動作を変更することも可能となる。

2.2　図形文法

図形言語を定義するための文法が図形文法である。図形文法では、図形間の空間的な位置関係を基に関係を記述することで、対象とする図形言語の構造やその意味を定義する。図形文法としては、Positional Grammars[Cos98]、Relational Grammars[Fer94]、Picture Layout Grammars[Gol91]、Constraint Multiset Grammars (CMG)[Mar94] などが提案されている。テキストにおけるプログラミング言語の文法では、LL 文法や LR 文法などの文法があり、これらによって扱える言語の範囲が異なる。これは図形文法においても同様で、図形文法によってその定義方法が異なるだけでなく、扱える図形言語の範囲も異なってくる。

なお、このような図形文法を用いて図形言語を定義するのは、必ずしも容易ではない。

一般にテキストで表現されるため、入力には手間がかかる上に、理解しにくいといった問題があった。そこで、図形などを用いて定義したい内容を表現することにより、ビジュアルな定義が行えるシステムが提案されている。たとえば、馬場らは、図形言語に対応する図形を最初に入力することでこの手間を軽減し、定義対象の概要を直感的に理解できるような記述手法を提案している [Bab97a]。また、この手法を拡張して、入力された図形から定義を推測する手法が藤山らによって提案されている [Fuj99, Fuj00]。西名らは、図形文法を用いた図形の書き換えを視覚化する手法を示している [Nis99]。さらに、亀山らは、図式表現とテキスト表現を相互に利用する手法 [Kam02] や、ガイドラインなどを用いて図形間の関係を図示することにより、定義内容を容易に理解できる手法 [Kam03a, Kam03b] を提案している。

2.3　空間解析器生成系

テキストのプログラミング言語では、文法を与えることにより自動的に解析器を生成するシステムとして構文解析器生成系（コンパイラ生成系）があり、Yacc[Joh79] や Bison[1)]、Rie[Sas93] などのシステムが存在する。これと同様に、図形文法を与えることにより空間解析器を自動生成するのが空間解析器生成系である。空間解析器生成系としては、VLCC[Cos95, Cos99]、SPARGEN[Gol93]、PROGRES[Rek95]、Penguins[Cho98]、恵比寿 [Bab98b] などがある。

VLCC、SPARGEN、Penguins などのシステムでは、図形文法をシステムに与えると、その図形言語を解析する空間解析器のプログラムソースを出力する。このプログラムソースを C コンパイラなどでコンパイルすることで、求める空間解析器が得られる。一方、恵比寿やその派生システムでは、インタラクティブに図形文法を定義できる。システム中で図形文法を定義するサブシステムがあり、これを用いて図形文法を定義することで、即座に空間解析器を得て実行できるようになっている。また、実行中に図形文法を変更できるため、動作を確認しながら文法定義ができる。恵比寿では、このような動作を実現するため図形文法を内部でデータとして持ち、空間解析器はこのデータに基づいて解析するようになっている。つまり、恵比寿の空間解析器はある特定の図形言語に特化したものではなく、与えられた図形文法データに基づいて解析する一般化した解析器となっている。

以下では、空間解析器生成系のうち、本研究に関連が深い Penguins と恵比寿、Rainbow、Handragen について説明する。

2.3.1　Penguins

Chok と Marriott は、図形文法を与えることで図式を編集するシステムを自動生成するシステムとして、Penguins を提案している。Penguins に与える図形文法は CMG を用いている。図形言語として、状態遷移図や数式、フローチャート、シーケンス図、2 分木、n-分木を利用した例を取り上げている。Penguins では、解析の際に図形の微小な位置関係を

[1)] http://www.gnu.org/software/bison/bison.html

図形文法に基づいて訂正しながら解析を行うことを提案している。また図形間の位置関係を保存するため、制約解消系として QOCA[Bor97] を用いており、これを用いて、図式の整形を行っている [Cho99]。また、文献 [Cho98] と [Cho03] において、CMG に基づく定義における LHS の記号を 2 つ以上に拡張した例をあげており、これで図形の書き換えができると述べている。

2.3.2　恵比寿

馬場らは、図形を用いて図形文法の一部を定義する手法を実現した空間解析器生成系である恵比寿を実装した。恵比寿では、図形文法として、CMG をベースに生成規則に “アクション” の概念を導入した “拡張 CMG” を利用している [Bab98a, Bab98b]。アクションは拡張 CMG の一部として記述され、ルールが適用されて新たな非終端記号が生成された際に実行される、図形の変更や削除、新規図形の追加などのスクリプトの記述が行える。これにより、Marriott らによる図形文法の記述力による分類 [Mar96, Mar98] における type 0 を実現して、広い範囲の図形言語を扱えるようになっている。馬場らは、GUI を記述するためのビジュアル言語 [Bab97b] を恵比寿で実現し、アクションによって自動的に GUI を生成することができることを示している。また、アクションを利用した例として、VISPATCH[Har97] や HI-VISUAL[Yos86, Hir91] などのサブセットの実装も示している [Bab98c, Bab99]。

また、恵比寿では図形の重なり判定において、ある程度のマージンを許して認識を行うようにし、図形を正確に重ねなくても図式が認識されるような機構を導入している。さらに、制約解消系 SkyBlue[San94a, San94b] を利用して図式のレイアウトを実現している。なお、土屋は恵比寿における制約解消系の高速化を図るため、恵比寿で HiRise[Hos99, Hos00] や Cassowary[Bad01] といった制約解消系を利用する手法を提案している [Tsu01]。

2.3.3　Rainbow

丁らは恵比寿をベースに拡張を行い、レイアウト機能を充実させ、“硬いレイアウト制約” と “軟かいレイアウト制約” を提供する空間解析器生成系 Rainbow[Jou00a, Jou00b, Jou01] を提案している。硬いレイアウト制約としては、座標や図形間の距離を固定させるものがあり、隣り合う図形をきれいに並べる際などに利用する。軟かいレイアウト制約としては、スプリングモデル [Ead84] やマグネティックスプリングモデル [Sug94]、Walker のアルゴリズム [Wal90] などを利用したグラフや木構造を持った図形の配置や、リスト構造といった決められた形にレイアウトする機能を提供している。図式の解析に関しては恵比寿と同様の機能を提供するが、これらのレイアウト機能を利用することで、図式を作成する際にインタラクティブに整形を行うことができるようになる。

2.3.4 Handragen

山田らは恵比寿をベースに拡張を行い、手書き入力を利用できる空間解析器生成系 Handragen を提案している [Yam02, Yam03]。Handragen では、キャンバス上に描かれた手書き入力のストローク情報を、ジェスチャ認識ステム SATIN[Hon00] に渡して、そのストロークがどのような形状であるか、そのパターン名と尤度の組のリストを取得する。そして、そのリストと、ストロークの開始点や終了点、バウンディングボックス、ストロークの長さなどの情報を属性として持つ 1 つの記号として空間解析器で扱う。このようにすることで、空間解析器で手書き入力が扱えるようになる。また、図形文法を用いて、その処理を記述できるようになっている。Handragen を用いた例として、手書き入力による図形や数字などの入力のほかに、手書き入力のジェスチャによる図形の削除などを示している。

志築らは空間解析器で手書き入力を扱う利点として、手書きストロークの形状だけでなく、すでに描かれている図式との位置関係や、他の手書きストロークとの位置関係など、コンテキストをふまえて手書き入力が解釈できる点をあげている [Shi03, Shi04]。たとえば、何もないところに円形のストロークを描いた場合は、円として認識するが、矩形の上で円形のストロークを描いた場合は、数字のゼロとして認識するといったことが可能となる。

2.4 空間解析器

図式を、その規則に基づいて解析を行うシステムが空間解析器である。それぞれの図形言語ごとに空間解析器を個別に作成する場合もあるが、恵比寿のような空間解析器生成系を用いて、図形文法から自動的に生成させることができる。

テキストのプログラミング言語の解析器では、字句解析器の出力結果をトークンとして扱っていたが、空間解析器では入力された図式における矩形や線などの基本図形要素を**トークン**として扱う。トークンの属性として、その図形の情報、たとえば、座標、色、線の太さなどを持つ。空間解析器は、この入力された図式が図形文法に当てはまるか、つまり図形文法の定義する図形言語の 1 つであるかを判定できる。さらに、その図式の構造を取得することも可能である。また、図形文法で各非終端記号に対する属性を定義することにより、その図式の意味を得ることもできる。

これまでに提案されてきた空間解析器生成系で生成される空間解析器では、図形エディタで描いた図式をいったん保存した後、それを解析するといった**静的な解析**を行うものが一般的であった。一方で、ユーザの入力にしたがって逐次的に解析を行い、図形の編集などに対応して**動的な解析**を行うことが考えられる。静的な解析はコンパイラ方式にあたり、動的な解析はインタプリタ方式にあたる。

VLCC や SPARGEN、PROGRES は静的な解析を行う空間解析器を生成する。一方、Penguins と恵比寿、Rainbow、Handragen では動的な解析を行う空間解析器を生成する。これにより、図形エディタで編集中の図式に対してインタラクティブに解析が行われる。

2.5 インタラクティブな図式処理システムにおける空間解析器

1.1 節で触れた、文法指向の入力操作が可能な図式処理システムとしては、Relational Grammars を用いた Weitzman らのシステム [Wei93] や、Cocktail Napkin[Gro96b, Gro96a]、Penguins などがある。Chok らは、このようなシステムを “スマートな図式処理システム” と呼んでいる [Cho98]。これらのシステムでは、図式の解釈に空間解析器が利用され、図式に対して動的な解析が行われる。

インタラクティブシステムにおいて空間解析器を利用することで、解析結果が即時に得られるだけでなく、その結果をその場で利用することが可能であり、特に編集機能の強化に有効に活用できる。これは、静的な解析では得られない利点である。

動的な解析の結果を利用することで実現できる機能としては、たとえば、次のようなものが考えられる。

- 解析結果をユーザに示し、正しい図式を入力しているかどうかフィードバックする。
- 認識した図形間の構造を保持するためにレイアウト機能を追加する。
- 操作対象の図形をハイライトさせるなどしてユーザの入力を補助する。
- 図形間の位置や大きさなどを他の図形と揃える [Iga97]。
- 図形の移動中などに他の図形に対するスナッピングを行う [Mas00]。
- ドラッグ中のオブジェクトと他のオブジェクトとの関係を明示する [Oga02]。

空間解析器を利用することで、このような機能を提供するのに必要な図式の解析情報を得ることができる。これらの機能のうち、レイアウト機能は、すでに提案されている空間解析器生成系で提供されている。これ以外の場合は、図形エディタなどが解析結果を利用してこれらの機能を提供することができる。

インタラクティブシステムにおいて空間解析器を利用する場合、ユーザの入力した図式に対して即座に解析が終了する必要がある。図式に変更があった場合に、図式のすべてを解析し直していては速度に悪影響を及ぼしかねない。そこで、現在までの解析結果を利用して、新たな入力との差分のみを解析するインクリメンタルな解析を行う必要がある。また、この際にも、図形の削除や編集などに対して正しく解析を行う必要がある。

静的な解析では、開始記号から生成を繰り返して解析木を作成し、入力された図式をすべて含むような解析木が 1 つでも見つかればよい。つまり、ある部分記号列に対して、複数の解釈が可能である場合でも、どれか 1 つの解釈が最終的に利用されればよい。しかし、動的な解析においてすべての可能な解釈を探索した場合、非常に多くの計算量が必要となる。また、動的なフィードバックを与えるためには、現在の図式を決定的に認識することが求められる。よって、動的な解析に用いるインクリメンタルな解析では、曖昧な解釈が発生しないような、図形文法の定義ができることが求められる。このようなインクリメンタルな解析を実現できる図形文法として CMG がある。

インクリメンタルな解析を行う空間解析器を生成するシステムとしては、図形文法として CMG を利用した Penguins や恵比寿などがある。一方で、テキストのプログラミン

グ言語における構文解析においても、LR 属性文法におけるインクリメンタルな解析の研究 [Nak96, Nak97] などがなされている。実際のシステムでは、eclipse[2)]における Java のプログラム環境において、プログラムの動的な解析を行い、オブジェクトに対するメソッドの候補を提示することなどが可能となっている。また、Harmonia-Mode[3)]でも、インクリメンタルな構文解析を行っている。

CMG に基づく解析では、動的な解析を実現するため、解析木をボトムアップに作成する形で行われる。つまり、終端記号を非終端記号に還元し、さらに終端記号や非終端記号を還元しながら解析を行う。一般に、入力された図形言語が図形文法に従った “正しい” 図形言語であるためには、最終的に開始記号 1 つに還元されることが求められる。しかし、インタラクティブシステムにおける空間解析器ではその図式の作成過程が重要であり、途中の状態で常に正しい図形言語である必要はない。また、最終目標が正しい図形言語の入力である場合はその言語を定義する際に開始記号を与える必要があるが、インタラクティブシステムにおける空間解析器では開始記号が必要でない場合がある。特に、恵比寿のアクションや図形の書き換えのような機能を用いて、インタラクティブに処理を実行し、それを目的とするようなシステムでは各状態での解析と処理が重要であり、開始記号に還元する必要はなく、常に正しい図形言語である必要もない。

[2)] http://www.eclipse.org/

[3)] http://www.cs.berkeley.edu/~harmonia/harmonia/projects/harmonia-mode/doc/index.html

第3章　CMGによる図形言語の定義

本章では、インタラクティブな図式処理システムを記述するための図形文法である CMG について述べる。まず、CMG による図形文法の記述について説明し、その後に図形言語の例をあげて、CMG による図形言語の定義例を示す。さらに、CMG による図形言語の定義についての調査結果を示す。

3.1　CMG

CMG は、Marriott によって 1994 年に文献 [Mar94] にて正式に紹介されているが、文献 [Hel91] 中でも解析器を定義するための記法として非公式に利用されていた。

CMG は、他の図形文法に比べ簡潔に記述することが可能である。また、定義の内容が分かりやすいといった特徴がある。さらに、Marriot と Mayer は文献 [Mar96] において、Positional Grammars と Relational Grammars、Unification Grammars[Wit90] の各図形文法が CMG を用いて書き換えることができることを示している。つまり、これらの文法で扱える図形言語はすべて CMG で記述できるということである。さらに、CMG では図形間の任意の関係を記述できるため、広い範囲の図形言語を記述することが可能となる。

このように、文法の記述力が強く、また、解釈の曖昧性を取り除くことができ、インクリメンタルな解析が行えるといった理由から、インタラクティブな図式処理システムを定義する図形文法として CMG を利用する。

CMG では、**ルール**（生成規則）によって図形言語を定義する。複数のルールを用いて、対象の図形言語の構造や意味を指定する。ルールの記述の基本形は以下のようになる。

T ::= T_1, T_2,$\cdots$, T_k **where (**
　Constraints
) {
　AttributeAssignments
}

これは、“::=”の右辺にある **RHS の記号列** T_1, T_2,$\cdots$, T_k に対応するトークンが *Constraints* で示された**制約条件**をすべて満たしたときに、左辺にある **LHS の記号** T に還元されることを表している。つまり、制約条件を満たした RHS の記号列が新たな LHS の記号に置き換わる。LHS の記号 T の属性値は、*AttributeAssignments* で示された**属性定義**で決定される。入力された矩形や線などの基本図形は終端記号として扱われる。非終端記号は、ルールによって還元された記号である。

CMG では任意で **exist の記号**や **not-exist の記号**を含むルールの記述が可能である。これらの記号は、認識の際の曖昧性を取り除いて決定性のある解釈を行うため、RHS の記号以外の図式の状態を利用する際に用いられる。また、**all の記号**を含んだルールの記述も可能である。この場合のルールの記述は以下のようになる。

T ::= T_1, T_2,$\cdots$, T_k, **exist** E_1, E_2,$\cdots$, E_i, **all** A_1, A_2,$\cdots$, A_j **where** (
　Constraints
　not_exist N_1, N_2,$\cdots$, N_m **where** (
　　NegativeConstraints
　)
) {
　AttributeAssignments
}

ここで、E_1, E_2,$\cdots$, E_i は exist の記号である。exist の記号は、それに対応するトークンが存在したときのみ、そのルールが適用できることを示す。RHS の記号と同様に exist の記号も制約条件でその条件を指定される。制約条件を満たした場合、ルールによって RHS の記号列は LHS の記号に還元されるが、exist の記号は還元されない。しかし、LHS の記号 T は、どのトークンが exist の記号として利用されたかを把握しており、属性定義においても exist の記号が利用できる。A_1, A_2,$\cdots$, A_j は、all の記号であり、同じ種類の記号を集めるのに利用される。all の記号の条件も制約条件で記述される。また、all の記号は、RHS の記号と同様に LHS の記号に還元される。N_1, N_2,$\cdots$, N_m は、not-exist の記号である。not-exist の記号に関する制約条件は、*NegativeConstraints* で指定される。この制約条件を**ネガティブ制約条件**と呼ぶ。ネガティブ制約条件は not-exist の記号間の関係だけでなく、RHS の記号や exist の記号との関係も記述する。あるトークンの組合せによって RHS の記号などが制約条件を満たした場合でも、その組合せに対して、ネガティブ制約条件を満たすような not-exist の記号列が存在した場合、そのルールは適用されないことを示す。exist の記号と not-exist の記号は、すでに他の非終端記号に還元されたトークンでもマッチする。なお、RHS の記号、exist の記号、not-exist の記号、all の記号のそれぞれは異なるトークンである。

3.2　図形言語の例

3.2.1　計算の木

ここで、CMG による図形言語の定義の簡単な例として、文献 [Bab98a] に示されている “計算の木” の定義を一部変更した例を示す。計算の木は、図 3.1 のような図式である。節が演算子を表しており、それぞれの葉の値から計算を行う。図 3.1 の例では、左から順に、“4”、“$3+5$”、“$(6/2)+7$” という計算式を表している。この図形言語は、次に示す 2 つのルールで記述される。なお、終端記号として、円（Circle）とテキスト（Text）、線（Line）が利用されている。

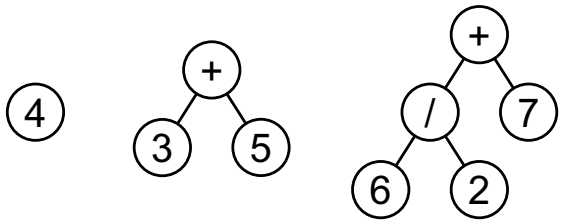

図 3.1: 計算の木

```
# ルール 1
n:Node ::= c:Circle, t:Text where (
  c.mid == t.mid
  isValue(t.text)
) {
  n.mid = c.mid
  n.midx = c.mid_x
  n.val = t.text
}

# ルール 2
n:Node ::= c:Circle, t:Text, l1:Line,
     l2:Line, n1:Node, n2:Node where (
  c.mid == t.mid
  isOperand(t.text)
  c.mid == l1.start
  c.mid == l2.start
  l1.end == n1.mid
  l2.end == n2.mid
  n1.midx < n2.midx
) {
  n.mid = c.mid
  n.midx = c.midx
  n.val = eval(n1.val, t.text, n2.val)
}
```

計算の木は、非終端記号 Node として認識される。再帰的な定義を用いて “木” の構造を定義している。

ルール 1 は、木構造の葉にあたる部分を認識する。RHS の記号として、記号 Circle と記号 Text が指定されており、これが制約条件を満たした場合に非終端記号 Node に還元する。ここで、記号 Circle に対応する変数として c、記号 Text には t が関連付けられている。制約条件では、その中心点が一致していているという条件 “`c.mid == t.mid`” と、テキストが数字であるという条件 “`isValue(t.text)`” が指定されている。ここで、テキストが

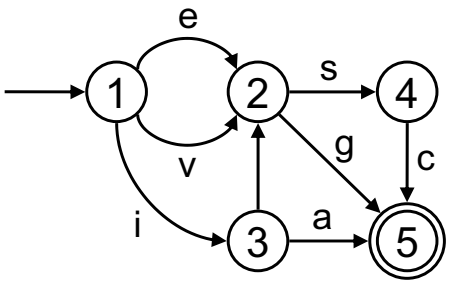

図 3.2: 状態遷移図

数字であるという条件は、同じ形状の節の部分と区別するために用いられている。属性定義では、RHS の記号の属性を用いて、節とエッジで接続する際の位置を示す属性 `mid` と、左右の位置関係を示す属性 `midx` を定義している。また、この非終端記号が表す値として属性 `val` にテキストの文字列を設定している。

ルール 2 では、計算の木を再帰的に定義している。RHS の記号として、節にあたる円と演算子にあたるテキスト、この節につながる 2 つの記号 Node、それらを結ぶエッジである線が指定されている。これらから、LHS の記号 Node を定義している。制約条件では、節とその節に対する葉の接続の条件のほかに、テキストが演算子の記号であるという条件“`isOperand(t.text)`”と、左右の葉の位置関係“`n1.midx < n2.midx`”を指定している。左右の葉の位置関係は、5 – 3 を 3 – 5 と誤認識しないようにするため、どちらが左の記号 Node で、どちらが右の記号 Node か判別するために用いている。属性定義では、ルール 1 と同様に、属性 `mid` と `midx` が定義されている。また、この非終端記号 Node が表す値として、2 つ葉が持つ値と節の持つ演算子から、eval() 関数を利用して計算された値を属性 `val` に設定している。

3.2.2　状態遷移図

exist の記号、not-exist の記号、all の記号を利用する例として、文献 [Cho03] に示されている状態遷移図の例を取り上げる。ここで定義される状態遷移図は図 3.2 のようなものである。終端記号として、円（Circle）とテキスト（Text）、矢印（Arrow）が利用されている。この図形言語は 10 個のルールで定義される。各ルールは、定義の内容で以下の 4 つに分類される。

1. 遷移を表す非終端記号を定義するルール。
2. 状態を表す非終端記号を定義するルール。
3. 状態を表す記号や遷移を表す記号を集める非終端記号を定義するルール。
4. 状態遷移図全体を表す非終端記号を定義するルール。

遷移を表す非終端記号は、2 段階で認識される。まず、“遷移を表す図形” の認識が行われ、その後に、この図形を利用して遷移を表す非終端記号が認識される。

```
# Arc
r:Arc ::= a:Arrow, t:Text where (
  a.mid == t.mid
) {
  r.start = a.start
  r.end = a.end
  r.mid = a.mid
  r.label = t.label
}
```

```
# StartArc
s:StartArc ::= a:Arrow where (
  not exist r:Text where (
    r.mid == a.mid
  )
) {
  s.start = a.start
  s.end = a.end
  s.mid = a.mid
}
```

図 3.3: 状態遷移図の定義（遷移を表す図形）

```
# Transition(A->B)
t:Transition ::= a:Arc,
    exist s1:State, s2:State where (
  onCircle(a.start, s1.mid, s1.radius)
  onCircle(a.end, s2.mid, s2.radius)
) {
  t.start = s1.label
  t.tran = a.label
  t.end = s2.label
}
```

```
# Transition(A->A)
t:Transition ::= a:Arc,
    exist s:State where (
  onCircle(a.start,s.mid,s.radius)
  onCircle(a.end,s.mid,s.radius)
) {
  t.start = s.label
  t.tran = a.label
  t.end = s.label
}
```

図 3.4: 状態遷移図の定義（状態の遷移を表す非終端記号）

遷移を表す図形は 2 種類あり、一般の遷移を表す図形が記号 Arc、開始状態への遷移を表す図形が記号 StartArc として認識される。これらは図 3.3 のように定義される。

記号 Arc と記号 StartArc を区別する基準は、矢印の中心位置にテキストが存在するか否かである。テキストが存在する場合は記号 Arc として認識され、テキストが存在しない場合には記号 StartArc として認識される。この “存在しない” という条件を記述するために、not-exist の記号 Text が利用されている。ネガティブ制約条件の “`r.mid == a.mid`” で、中心が重なるテキストが存在しないという条件を示している。もし、このような条件が指定されていない場合は、テキストが存在しても記号 StartArc として認識できるので解釈が曖昧になる。

状態の遷移を表す非終端記号 Transition は記号 Arc を用いて定義される。2 つの状態間を遷移する場合と、同一の状態に遷移する場合で、2 通りに定義される。これらは図 3.4 のように定義される。なお、記号 State は状態を表す非終端記号である。また、記号 StartArc は開始状態の定義で利用されるため、ここでは利用されていない。

それぞれのルールで exist の記号が利用されており、“記号 State が存在する” という条件として利用されている。これにより、単に遷移を表す図形だけでは遷移であると認識されない。また、遷移として認識された場合は、どの状態からどの状態への遷移か把握することができる。仮に、記号 State を RHS の記号として利用した場合、その状態は還元されてしまうため他の遷移でその状態を利用できなくなる。このような理由から exist の記号が使われている。なお、異なる RHS の記号や exist の記号には同一のトークンが当てはまる

```
# State(final)
s:State ::= c1:Circle, c2:Circle,
    t:Text where (
  c1.mid == c2.mid
  c1.mid == t.mid
  c1.radius <= c2.radius
) {
  s.mid = c1.mid
  s.radius = c2.radius
  s.label = t.label
  s.kind = "final"
}

# State(start)
s:State ::= c:Circle, t:Text,
    a:StartArc where (
  t.mid == c.mid
  onCircle(a.end, c.mid, c.radius)
  not exist m:Circle where (
    m.mid == c.mid
  )
) {
  s.mid = c.mid
  s.radius = c.radius
  s.label = t.label
  s.kind = "start"
}

# State(normal)
s:State ::= c:Circle, t:Text where (
  not exist m:Circle where (
    m.mid == c.mid
  )
  not exist a:StartArc where (
    onCircle(a.end, c.mid, c.radius)
  )
  t.mid == c.mid
) {
  s.mid = c.mid
  s.radius = c.radius
  s.label = t.label
  s.kind = "normal"
}
```

図 3.5: 状態遷移図の定義（状態を表す非終端記号）

ことがない。このため、遷移元と遷移先の状態が異なる場合と、同じ状態への遷移は別のルールとして記述されている。

状態を表す非終端記号は、一般の状態に加え、開始状態と終了状態があるため、それぞれ異なるルールで定義されるが、これらはともに非終端記号 State として指定されている。このことにより、記号 Transition の定義において、どの種類の状態でも利用することが可能となる。これらは図 3.5 のように定義される。

二重円の場合には終了状態として認識され、記号 StartArc がつながっている場合には開始状態、二重円ではなく記号 StartArc もつながっていない場合は一般の状態として認識される。これらの区別は、遷移を表す図形と同様に not-exist の記号を用いて記述されている。これらの状態は同じ記号 State として認識されるが、それぞれの区別は属性 `text` にて保持されている。

なお、この定義では開始状態が終了状態であるような図式は認識できない。これは、単に定義し忘れているだけだと考えられる。開始状態が終了状態であっても認識できるようにすることは可能であり、そのようなルールを 1 つ加え、他のルールを若干変更すれば実現できる。

状態遷移図では、多くの状態を表す記号や遷移を表す記号が存在することになる。これらを集めて 1 つの非終端記号として扱うために、すべての状態を集める記号 States と、すべての一般の遷移を集める記号 Transitions がある。これらは図 3.6 のように定義される。

これらは、all の記号を用いて、存在するすべての状態や遷移を集めている。たとえば、記号 States はすべての記号 State を還元し、それら記号は属性 `set` に設定される。

```
# States
ss:States ::= all s:State where (
) {
  ss.set.add(s)
}
```

```
# Transitions
ts:Transitions ::=
    all t:Transition where (
) {
  ts.set.add(t)
}
```

図 3.6: 状態遷移図の定義（状態の記号や遷移の記号を集める非終端記号）

```
# STD
f:STD ::= ss:States, ts:Transitions,
    exist s:State where (
  s.kind == "start"
) {
  f.ss = ss
  f.ts = ts
}
```

図 3.7: 状態遷移図の定義（状態遷移図全体を表す非終端記号）

状態遷移図全体を表す非終端記号は、記号 Transitions と記号 States から、記号 STD として定義される。これにより、図式全体が 1 つの状態遷移図として認識される。これは図 3.7 のように定義される。

ここで、exist の記号として、記号 State が利用されている。これは、状態遷移図に必ず 1 つは開始状態が存在するという条件を示している。

3.3 図形言語におけるルール定義の調査

CMG で記述された図形言語を調査し、その定義の内容について分析を行った。調査の対象は、論文に定義が掲載されているものや、公開されたシステムに付属した例題である。対象となった図形言語は以下の 9 種類である。なお、恵比寿システムは、筑波大学田中研究室の Web ページ[1)]で公開されている。

- 折れ線（図 3.8）
  複数の線分で構成される折れ線を扱う図形言語。
  1 つにつながっている線分の集まりを折れ線と認識することができる。文献 [Iiz03b] で示されている。定義は、図 4.6 で示す。

- リスト構造（図 3.9）
  データを矢印でつなぐことで線形のリスト構造を表した図形言語。
  1 つにつながった線形リスト構造を認識することができる。文献 [Iiz03b] で示されている。定義は、図 4.7 で示す。

- 計算の木

[1)] http://www.iplab.is.tsukuba.ac.jp/

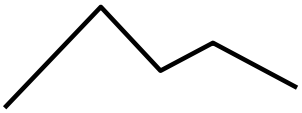

図 3.8: 折れ線

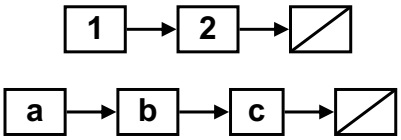

図 3.9: リスト構造

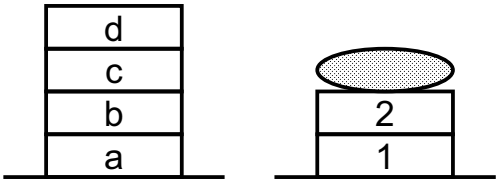

図 3.10: スタック構造

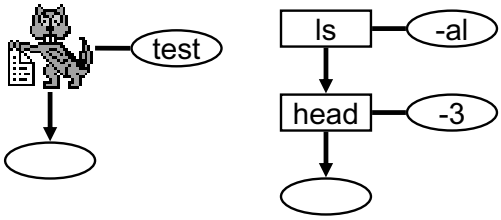

図 3.11: VSH

3.2.1 節で示した計算を扱う図形言語。
木構造で表現された計算式が認識することができる。

- スタック構造（図 3.10）
  スタック構造を扱う図形言語。
  データを積み重ねた 1 つのスタック構造を認識することができる。また、アクションの定義により、スタックへのデータの追加や、最上位のデータを取り除く処理が行える。定義は文献 [Fuj01] に示されている。

- 状態遷移図
  3.2.2 節で示した状態遷移図を扱う図形言語。
  グラフ構造で表現された状態遷移図を認識することができる。

- VSH（図 3.11）
  パイプでつなげたシェルコマンドを視覚化した図形言語。
  たとえば、“`ls -al | head -3`”のようなコマンド列を木構造で表現した図式を認識できる。アクションにより、図式に対応するコマンドを実行することができる。定義は恵比寿システムの例題に含まれている。

- GUI（図 3.12）
  ボタン、テキストエリア、スクロールバーなどの GUI コンポーネントのレイアウトを定義するための図形言語。
  文献 [Bab97b] に示された GUI の記述が認識できる。アクションにより、定義された GUI に対応する Tcl/Tk プログラムを出力し、実行することができる。定義は恵比寿システムの例題に含まれている。

- HI-VISUAL（図 3.13）
  アイコンの重ねあわせによって動作を指示することのできるシステム HI-VISUAL のサブセット。

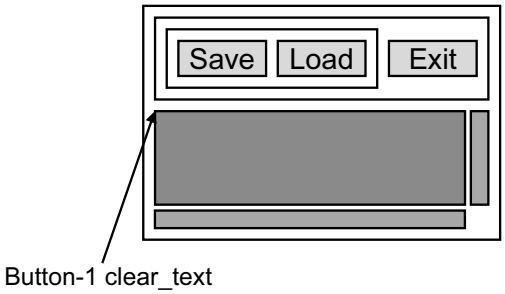

図 3.12: GUI

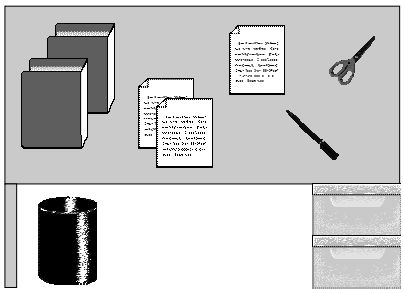

図 3.13: HI-VISUAL

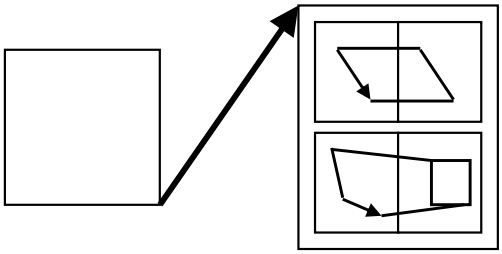

図 3.14: VISPATCH

アクションにより、アイコンが重ねあわされた場合に処理が実行される。たとえば、ペンをファイルに重ねると、対応するファイルがエディタで開かれる。定義は恵比寿システムの例題に含まれている。

- VISPATCH（図 3.14）
  イベントドリブンな図形の書き換え規則を図式で表現し実行できる VISPATCH のサブセット。
  アクションにより、ルール領域に定義された処理が、イベントセンサ上で実行することができる。定義は恵比寿システムの例題に含まれている。

最初に、各図形言語におけるルール数と RHS の記号数の分布に関する調査について述べる。その結果を表 3.1 に示す。なお、本論文では “RHS の記号数” として RHS の記号と exist の記号を含んだ数を指す。これは、4.9 節で示すように、解析において一般の RHS の記号と exist の記号は同一に処理されるからである。また、all の記号を持つルールについても解析時に異なる処理を行うため別にカウントした。

ここで、k 個の RHS の記号を持つルールを R_k と表すことにする。たとえば、R_3 は 3 個の RHS の記号を持つことになり、計算の木のルール 2 は 6 個の RHS の記号を持つので R_6 になる。なお、R_{all} は all の記号を持つルールである。

今回の調査対象の図形言語では、あわせて 83 個のルールが定義されていた。この中で、RHS の記号数で分類した場合に一番多かったのは R_2 のルールであり約 47%を占めている。これらのルールは、図形言語の基本的な構成要素を定義している。たとえば、計算の木のルール 1 は R_2 である。

表 3.1: ルール数と RHS の記号数の分布

分類 / 図形言語	R_1	R_2	R_3	R_4	R_5	R_6	R_{all}	計
折れ線	-	1	-	-	-	-	-	1
リスト構造	-	1	-	1	-	-	-	2
計算の木	-	1	-	-	-	1	-	2
スタック構造	1	3	-	-	-	-	-	4
状態遷移図	1	3	4	-	-	-	2	10
VSH	1	2	3	3	1	1	-	11
GUI	6	3	4	-	-	-	1	14
HI-VISUAL	8	7	-	-	-	-	-	15
VISPATCH	2	18	2	-	-	-	2	24
計	19	39	13	4	1	2	5	83

2 番目に多かったのは R_1 のルールであった。R_1 のルールは、1 つの記号を単に置き換えているに過ぎないが、重要な役割を果たしている。R_1 のルールの中で、11 個のルールはその記号の属性が特定の値を持つという制約条件を持っていた。また 5 つのルールは、not-exist 記号を利用し、周りの環境の条件を指定している。この 2 つのタイプは、1 つの記号に対して、異なる非終端記号を割り当てている。一方、3 つのルールは、制約条件が何もなかった。これらはいくつかの記号を 1 つのグループにまとめるために利用されている。つまり、異なる記号を 1 つの記号に対応させている。このようなルール定義を行うことで、他の RHS の記号で利用する際に 1 回の定義でまとめて記述できる。R_1 の例ではないが、状態遷移図の記号 State で 3 種類の定義があったのと同様の理由である。

RHS の記号数が最大のものは R_6 のルールであった。これは、先に挙げた計算の木のルール 2 と VSH の例で利用されている。RHS の記号の数がそれほど大きくならないのは、そうならないように定義されているとも考えられる。たとえば、状態遷移図の例では、遷移を表す非終端記号 Transition を認識する前に、遷移を表す図形に対応する記号 Arc の認識を行っている。このように、1 つにまとめることができるものでも、分割して定義している場合が多い。これは、そのルール定義を複数の部分に分割することで、分かりやすくしようとしているものと考えられる。このような理由からも、R_2 のルールが多くなっていると思われる。

ルール数が一番少なかった図形言語は折れ線の例であり、ルール 1 つで定義されている。ルール数が一番多かったのは、VISPATCH の例であり、24 個のルールで定義されていた。これからも分かるように、一般に複雑な図式ほど定義するルール数が多くなる。

制約条件をその利用方法に基づき分類を行い、その出現数について調査した。結果を表 3.2 に示す。なお、今回の調査対象の図形言語では、あわせて 204 個の制約条件が利用されていた。

表 3.2: 制約条件の分類と出現頻度

個数	C_k	分類	例
60	C_2	座標の一致	`c.mid == t.mid`
35	C_1	図形属性の値の指定	`c.innercolor == "red"`
31	C_2	x 座標または y 座標の一致	`r.x0 == s.x0`
24	C_2	領域の包含関係	`contains(r.bb, h.bb)`
11	C_1	属性と固定値との大小比較	`d.length <= 10`
10	C_1	一般属性の値の指定	`s.kind == "start"`
8	C_2	属性の大小比較	`c1.radius <= c2.radius` `f.id < p.id`
7	C_2	高さまたは幅の一致	`r.width == s.width`
7	C_2	領域の交差	`overlap(b.bb, f.bb)`
6	C_2	円周上に存在する点	`onCircle(a.end, c.mid, c.radius)`
4	C_2	領域に含まれる点	`contains(t.mid, h.bb)`
3	C_2	左右または上下の位置関係	`n1.right < n2.left`

この結果より、座標の一致に関する制約条件が非常に多く含まれていることが分かる。つまり、図形言語の傾向として、何かがつながっていることにより図形間の関係を記述する例が多いということである。また、x 座標または y 座標の一致に関する条件も 3 番目に多かった。このような制約条件は、1 つのルールで x 座標と y 座標の属性をそれぞれ指定しており、一方だけの制約条件を指定している例は存在しなかった。たとえば、リスト構造の例では、ノードとエッジがつながれているという条件を、矩形の右端とエッジの x 座標、矩形の中心とエッジの y 座標のそれぞれの一致で指定している。この場合は、結局座標の一致を表現している。矩形に右中央に対応する座標情報が属性として存在しなったためこのような定義がなされている。また、2 つの図形が隣接するというような条件でも、この種類の制約が利用されていた。

制約条件を、その中で利用されている記号の数によって分類を行った。ここで、k 個の記号を利用している制約条件を C_k と表すことにする。たとえば、制約条件 “`c.mid == t.mid`” は `c` と `t` の 2 つの記号を利用しているので C_2 になる。なお、“`a.v1 == a.v2 + 10`” という制約条件は、2 箇所で記号が利用されているが、1 種類の記号しか利用されていないので C_1 である。

制約条件のうち、C_1 は 56 個であったが、C_2 は 148 個利用されており、非常に多く用いられていることが分かった。なお 3 個以上の記号を利用している制約条件は 1 つも存在しなかった。C_1 の制約条件は、その記号の属性が特定の値を持つという条件により他の記号と区別している。たとえば、計算の木のルール 1 における制約条件 “`isValue(t.text)`” は `t` の文字列属性 `text` が数値の値を持つという条件を指定している。C_2 の制約条件は 2 つの記号間の関連を指定している。RHS の記号や exist の記号、not-exist の記号間におけ

るすべての関係はすべてこの制約条件で指定されていることになる。

いくつかのルールでは、制約条件中に利用されていない記号が存在した。つまり、そのルールでは、その記号の条件がないということである。これらのルールは、条件がない記号に対応するトークンは 1 つしか存在しないと仮定することができる点が共通していた。つまり、実際の利用に関しては問題がない。たとえば GUI の図形言語では、その記号が存在するかどうかで内部的なモードを切り替えるのに利用されている。

いくつかのルールでは、制約条件をまったく持たないルールがあった。つまり、それらの記号が存在するだけで無条件に還元が行われる。これらは 2 つの種類に分類できる。1 つは上述のような、記号に対して 1 つしか対応するトークンが存在しないと仮定できるケースであり、もう 1 つは R_1 のルールでいくつかの記号を 1 つのグループにまとめるケースである。たとえば、GUI の例では、テキストボックスやボタンなどを、GUI コンポーネントというグループにまとめている。これにより、フレームで構造化させる際に GUI コンポーネントを含むという条件で、テキストボックスとボタンの両方に対応させている。

第4章　空間解析器の高速化

本章では、解析の速度に関する問題を解決するための空間解析器の高速化手法を提案する。最初に、既存の解析手法を示し、問題点を挙げる。その後に、これを高速化するための手法を提案する。未処理トークン集合を利用した解析や制約条件グラフを利用した解析手法などを提案する。また、これらの手法の有効性を実験により示す。

4.1　従来の解析手法

4.1.1　Chok と Marriott の解析手法

CMG に基づく解析は、Chok と Marriott が 1995 年に文献 [Cho95a] と [Cho95b] において示している。

入力された図形はトークンとして空間解析器に渡される。図形に対応する終端記号のトークンや、ルールによって生成された非終端記号のトークンは**トークンデータベース** D に設定される。解析は、このトークンデータベース中のトークンをルールに基づいて還元することにより行われる。つまり、このトークンデータベースには、解析木における部分木の根にあたるトークンが設定される。

解析は大きく 2 つの部分に分けられる。1 つは、ある 1 つのルールに関してトークンデータベースのトークンを還元することを試みる**ルール適用**の部分である。もう 1 つは、このルール適用を行うルールを選び出し、ルール適用を繰り返す**ルール選択**の部分である。

ルール選択の部分については、以下のアルゴリズムを示している。

```
routine Parser()
  for S := 1 to maxstrata do
    repeat
      changed := false
      for each P in strata(S) do
        if EvaluateRule(P) then
          changed := true
        endif
      end
    until changed == false
  end
end
```

ルール選択では、図形言語を定義するすべてのルールに関するルール適用を行う。ルー

ル適用は必要に応じて繰り返し行われ、どのルールでもトークンデータベースが変更されなくなった場合、つまりトークンが還元できなくなった時点で解析が終了する。

ルール選択で重要なのは、ルールが複数存在した場合のルール適用の順序である。ルールの適用順序を最適化することで解析の高速化を実現できる。解析木におけるトークンは、ルールの依存関係に基づいた順序に並んでいるので、ルールの適用順序としてこの依存関係を利用できる。つまり、解析木の下のほうで利用されるルールから先に適用する。まず、空間解析器生成系では、ルールの依存関係を元に *call-graph* を作成する。ルール A の RHS の記号がルール B の LHS の記号であった場合に、ルール A はルール B に依存関係が発生し、*call-dependent* であると言う。*call-graph* において、この関係に基づく強連結成分、つまり互いに依存関係のあるルールの集合は、空間解析器生成系によって計算され、その順序が求められる。アルゴリズムの部分では、strata() が強連結成分で構成される階層を表しており、一番低い階層の 1 から最高の階層の maxstrata まで各階層の適用を行っている。それぞれの階層においては、EvaluateRule() を用いてルール適用を繰り返し、階層内のどのルールでもトークンデータベースが変化しない場合は、その階層を終了し上の階層を適用していく。これにより、ルールに階層がある場合は無駄なルール適用を削減することが可能となる。ただし、図形言語によっては、ルールがフラットな関係になっていて、階層が 1 つしかないものもありえるので、このような場合には効果はない。

たとえば、計算の木の例では、ルール 1 は終端記号しか利用していないため、他のルールへの依存関係はない。ルール 2 は、終端記号のほかに、記号 Node を利用しているため、記号 Node を定義しているルール 1 とルール 2 自身に依存している。これより、最初にルール 1 を適用し、その後にルール 2 を適用すればよいことが分かる。なお、状態遷移図の例におけるルール適用順序は 4.11 節で示す。

ルール適用の部分については、以下のアルゴリズムを示している。

```
routine EvaluateRule(P)
  T := Variables(P)
  C := Constraints(P)
  changed := false
  for each A in AllCombination(T, D) do
    if SatisfiesConstraints(A, C) then
      N := NewNonTerminal(A, P)
      D := (D\A) ∪{N}
      changed := true
    endif
  end
  return changed
end
```

EvaluateRule() は、ルール選択で選ばれたルールを引数として渡され、そのルールに関するルール適用を試みる。このルールにおける RHS の各記号に対応する**トークンの組合せ**を AllCombination() を用いてトークンデータベース D にあるトークンから生成し、それ

らすべてに対して SatisfiesConstraints() で制約条件をチェックする。ルール適用において、制約条件を満たすトークンの組合せが見つかった場合、そのトークン列は LHS の記号に還元される。この LHS の記号に対応するトークン N は、NewNonTerminal() で作成され、トークンデータベース D に追加される。また、RHS の記号に対応するトークン列 A は、トークンデータベース D から取り除かれる。さらに、トークンデータベースに変更があったというフラグを立てている。どのトークンの組合せも制約条件に合致しなかった場合は、ルール適用は失敗となる。

この解析アルゴリズムは、トークンデータベースのトークンを還元することにより処理が進んでいく。トークンが追加される場合も、このトークンがトークンデータベースに追加されて、Parse() が呼ばれて解析される。つまり、ひとたび還元した非終端記号は、そのままトークンデータベースに残ったままトークンの追加が行われる。このように変更前の解析結果をそのまま利用しながら解析が進む。このことにより、図式の追加に対するインクリメンタルな解析を実現している。

このアルゴリズムの正当性については、文献 [Cho95b] に加えて [Cho03] でも触れられている。このアルゴリズムが図形言語を正しく識別し、解析が終了するためには、対象となる図形文法に、以下の 3 つの条件が必要であると述べている。

1. 図形文法が *cycle-free* であること。

   つまり、ある記号列が無限に還元を繰り返すことがないことが求められる。言い換えれば、生成列が有限長であることである。以下の例では、記号 T があった場合に無限列となる。

   A ::= B **where** (true) {}
   B ::= A **where** (true) {}
   B ::= T **where** (true) {}

   この例では、T が 3 番目のルールで B に還元され、それが 2 番目のルールで B に還元される。これがさらに 1 番目のルールによって A に還元されるため、トークンデータベースの変化が {T} → {B} → {A} → {B} → {A} → · · · というループに陥る。

2. 図形文法におけるルールが依存関係による階層化がされていること。

   つまり、ネガティブ制約条件によるループがないことが求められる。たとえば、次のような例である。

   N ::= T **where** (
   not-exist N **where** (true)
) {}

   この場合、記号 N が存在しない状態で記号 T が存在した場合、このルールが適用されるため、非終端記号 N が生成されトークンデータベースに追加される。しかし、このルールの not-exist 記号で N があり、ネガティブ制約条件を満たすため、この新

たに生成された非終端記号 N が在存できなくなる。このため、トークンデータベースから N が取り除かれ、元の RHS の記号 T がトークンデータベースに戻される。このとき、再びこのルールが適用されるため、トークンデータベースの変化が {T} → {N} → {T} → {N}→ · · · というループに陥る。

Chok と Marriott は、このようなことが起こらない図形文法のことを “階層化された図形文法” と呼んでいる。また、このような図形文法は、前述の強連結成分を求める際に発見できることを示唆している。not-exist の記号が下位の階層にあるならば、このようなループが発生しないことになる。

しかし、この条件は非常に強いものであり、この主張は、あくまで十分条件である。たとえば、not-exist の記号が同じ階層の記号であってもネガティブ制約条件が適切に設定されていればループが発生しない場合がある。

3. 図形文法に決定性があること。

つまり、階層化された図形文法によって定義される図形言語の任意の記号列が、ただ 1 つの解析木を持つ場合に、その図形文法は決定性がある。簡単に言えば、ある記号列に対して、還元することができるルールが 1 つしか存在しない場合である。同じ記号列に対して 2 つ以上のルールが適用できる場合は、曖昧な解釈となりうるので決定性はない。

これらは、アルゴリズムの側面から見た正当性と停止条件を示している。逆に、CMG で図形言語を定義する際には、これらの条件を踏まえた上で定義すればよいことになる。

ここで、ルール適用における計算量を検討する。トークンデータベースにあるトークンの数を n とし、ルールにおける RHS の記号数を k とするとした場合、探索されるトークンの組合せは ${}_n\mathrm{C}_k$ 個ありえるので、最大で $O(n^k)$ の計算量が必要であることが分かる。3.3 節で示したように RHS の記号数が 6 になるルールが存在した。このような場合は、$O(n^6)$ の計算量が必要となる。つまり、このアルゴリズムではトークン数が増えた場合に、インタラクティブシステムにおけるリアルタイムな解析は困難であることが分かる。

なお、ルール選択を含めた、解析全体における計算量の同定は難しい。前述のようなループに陥る場合や、入力されたトークンの状態などによって計算量は大きく変化しうる。Marriott は文献 [Mar96] において、CMG を用いた一般の図形言語の認識は NP 困難であることを示している。

4.1.2 解析の例

ここでは、Chok らの解析手法を用いた場合に実際にどのように解析が行われるか、具体的な例を用いて示す。

図 4.1 に示すような計算の木の解析を考える。この図において、矢印で示されているのは、図形に対応する終端記号のトークンの ID である。これらのトークンと、その属性のうち解析に必要な部分を書き出すと次のようになる。

```
[1] Circle: mid=(150,100)
[2] Circle: mid=(100,187)
[3] Circle: mid=(200,187)
[4] Text: mid=(150,100), text="-"
[5] Text: mid=(100,187), text="5"
[6] Text: mid=(200,187), text="3"
[7] Line: start=(150,100), end=(100,187)
[8] Line: start=(150,100), end=(200,187)
```

解析を始める前の初期状態として、トークンデータベース D に以下のようにトークンが設定される。

D = {1, 2, 3, 4, 5, 6, 7, 8}

この後、解析を行うため Parse() が呼ばれる。まず、ルール選択においてルール 1 が選ばれ、ルール 1 に関するルール適用が行われる。ルール 1 の RHS の記号は記号 Circle と記号 Text である。これにより、探索される組合せは、

[1,4], [1,5], [1,6], [2,4], [2,5], [2,6], [3,4], [3,5], [3,6] (4.1)

の 9 個の組合せとなる。最初に [1,4] の組合せを選ぶと、これは制約条件 “`c.mid == t.mid`” は満たすが、もう 1 つの制約条件 “`isValue(t.text)`” を満たさない。よって、この組合せはルール 1 を適用できない。同様に、[1,5], [1,6], [2,4] の組合せも制約条件を満たさない。[2,5] の組合せは、制約条件を満たすためルール 1 が適用され、新たな終端記号 Node が生成される。この属性は属性定義より決定され、次の新しいトークンが生成される。

```
[9] Node: mid=(100,187), midx=100
```

ここで RHS の記号であるトークン [2] と [5] は還元されるためトークンデータベースから取り除かれる。新たなトークンデータベース D は次のようになる。

D = {1, 3, 4, 6, 7, 8, 9}

同様にルール 1 を適用すると、[3,6] の組合せもルール 1 が適用され、次の新たな記号 Node のトークンが生成される。

```
[10] Node: mid=(200,187), midx=200
```

これによりトークンデータベース D は次のようになる。

D = {1, 4, 7, 8, 9, 10}

この時点でルール 1 を適用できなくなるため、ルール選択に戻り、次のルール 2 のルール適用が始まる。

ルール 2 に関する RHS の記号は、記号 Circle、記号 Text、記号 Line が 2 つと、記号 Node が 2 つである。これに当てはまるトークンの組合せは以下の 4 つである。

[1,4,7,8,9,10], [1,4,7,8,10,9], [1,4,8,7,9,10], [1,4,8,7,10,9] (4.2)

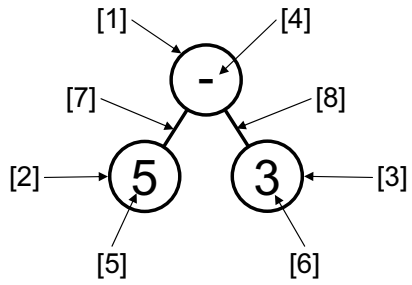

図 4.1: 解析対象の図式

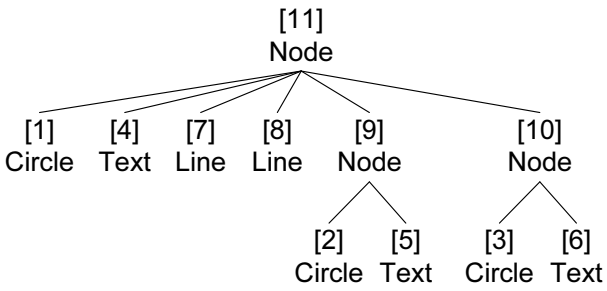

図 4.2: 解析木

なお、たとえば [1,4,7,7,9,9] のようなものは、記号名が一致する組合せであるが、RHS の記号に同じトークンが現れるので除外されている。この中で、2 番目の組合せと 3 番目の組合せは、`l1` と `n1` の中心点が一致しない。4 番目の組合せは中心点の一致はするが、制約条件“`n1.midx < n2.midx`”を満たさない。唯一 1 番目の組合せが、すべての制約条件を満たす。これにより、この組合せに対してルール 2 が適用され、次の新たな非終端記号が生成される。

```
[11] Node: mid=(150,100), midx=150
```

これによりトークンデータベース D は、

D = {11}

となる。この時点でルール 2 を適用できなくなり、解析が終了する。この解析で得られる解析木は図 4.2 のようになる。

4.1.3 Balt の解析手法

CMG に基づく解析に関して、Balt は 1996 年に文献 [Bal96] にて Chok らの手法とは異なる解析手法を提案している。

同じく CMG を利用している Penguins や恵比寿では、インクリメンタルな解析を行い、動的に図式を認識するが、Balt のシステムでは動的な解析を行わない。Chok らの解析手法では利用できる図形文法に、先に挙げたような制限があるが、Balt は、この図形文法の制限をとりはらって、CMG で記述できる言語はすべて扱うことができるようにしたと述べている。そのために、図式をまずボトムアップに解析を行い、さらにトップダウンに解析を行うことで、開始記号からの生成について確認を行っている。なお、解析は、図式が図形文法に従っているかどうかのチェックのみを行い、図式の意味をチェックする機能は実装されていない。

Balt は、ルールと記号の依存関係をグラフにして図示している。例として、次のようなルールを考える。なお、便宜上ルールの先頭に識別子を付与している。

[p_1] AA ::= A **where** $(\cdots)$ {}
[p_2] AA ::= A, BB **where** $(\cdots)$ {}
[p_3] BB ::= B, AA **where** $(\cdots)$ {}

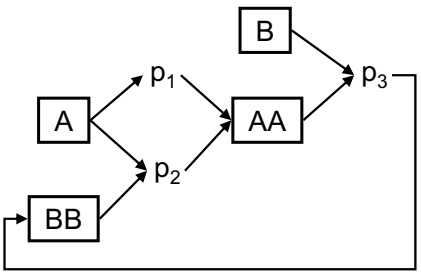

図 4.3: ルールと記号の依存関係

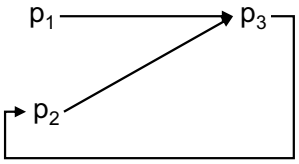

図 4.4: ルールの依存関係

このように定義された場合のルールと記号の依存関係は、図 4.3 のようになる。また、これから記号を取り除くことにより求められるルールの依存関係は図 4.4 のようになる。これより、p_2 と p_3 が強連結成分であることが分かり、ルール選択では、p_1 を適用した後に、p_2 と p_3 の組で適用すればよいことが分かる。

Balt は、解析の高速化を図るため、ルール適用におけるトークンの組合せの選び方に着目した。Chok らはすべての組合せに対して制約条件をチェックしていたが、逆に、制約条件を利用して探索する組合せを限定させ、チェックすべき組合せを減少させる手法を提案している。Balt が利用したのは、C_1 の制約条件である。C_1 の制約条件は、ある 1 つの記号にのみ着目しているため、簡単に探索範囲を限定できる。つまり、C_1 の制約条件を満たさなければ全体の制約条件を満たすこともないので、C_1 の制約条件を満たすトークンのみで構成される組合せを探索する。

たとえば、前述の解析の例で (4.1) の組合せを求める際に、ルール 1 における C_1 の制約条件 “`isValue(t.text)`” を利用すると、トークン 4 が制約条件を満たさないため、これを含む組合せを除外することができる。これにより、探索すべき組合せは、

$$[1,5],\ [1,6],\ [2,5],\ [2,6],\ [3,5],\ [3,6] \tag{4.3}$$

の 6 個に組合せを減少させることができる。このようにすることで一定の高速化が図れる。ただし、各 RHS の記号に対応するトークンの “割合” が減少するだけなので、一般に計算量のオーダに関する変化はない。

4.2 ルール間の依存関係の解析

Chok らは、ルールの依存関係から強連結成分やその適用順序の求める手法として、文献 [Ull88] に示されている演繹データベースの階層チェックの手法を改良したアルゴリズムを示している。また、Balt はルールの解析に独自のアルゴリズムを提案している。これらは多項式時間を必要とするアルゴリズムであった。

ここで、ルールの依存関係は図 4.4 で示されるような有向グラフで表現される。よって、このグラフにおける、強連結成分やその順序を求めるには、このグラフに対してトポロジカルソートを行うことにより、簡単にかつ高速に求めることができる。強連結成分はトポロジカルソートにより発見することができ、適用順序はトポロジカル逆順序に従えばよ

いことは明らかである。なお、強連結成分に関するトポロジカルソートの手法としては、Tarjan の深さ優先探索を行う線形アルゴリズム [Tar72] が知られている。

これによる高速化は、ルールを解析する際にのみに有効である。Penguins のような、特定の図形言語の解析器をあらかじめ出力する方式の空間解析器生成系では、空間解析器の生成時に計算されるため、インタラクティブ性に影響はない。しかし、恵比寿のような、空間解析器の動作時にルールに変更を加えることのできるシステムでは、ルールが変更された際の再計算の時間を減少させることができる。

4.3　制約条件を利用した組合せの削減

Balt は C_1 の制約条件を利用して探索範囲を限定させ、解析を高速化させた。これを拡張し、C_2 の制約条件も利用した解析手法を提案する [Iiz03a]。3.3 節に示したように、図形言語の定義では、C_2 の制約条件を多く利用している。よって、この制約条件を利用することで効率的に計算量を削減することができる。

ルール適用において、まず、各 RHS の記号に該当するトークン列がトークンデータベースから選ばれる。Balt の手法では、この時点でそれぞれの記号に対応するトークンを C_1 の制約条件を利用し、制約条件を満たさないトークンを除外する。この前処理により、トークンが減少するため、探索する組合せも減少することになる。これによって求められたトークンは M という変数に入れられるとする。M は配列になっており、M[k] は k 番目の RHS の記号に対応するトークンの集合が設定されているものとする。

提案する手法では、ここでさらに C_2 の制約条件を利用する。このアルゴリズムは以下のようになる。

```
routine CheckConstraintDouble(C, M)
  [X,Y] := ArgumentPosition(C)
  Xold := M[X]
  Yold := M[Y]
  M[X] := {}
  M[Y] := {}
  for each I in Xold do
    for each J in Yold do
      if StatisfiesConstraint([I,J], C) then
        AddToSet(M[X], I)
        AddToSet(M[Y], J)
        RemoveFromSet(Yold, J)
      endif
    end
  end
  return M
end
```

CheckConstraintDouble() は、第 2 引数で与えられたトークン集合の配列から、第 1 引数

に与えられた制約条件を満たすトークンを選択し、新たなトークン集合の配列を返す関数である。まず、ArgumentPosition() を利用して制約条件で利用している記号の位置を取得し、X と Y に設定する。この後、M[X] と M[Y] で構成される組合せに対して制約条件のチェックを行う。Xold と Yold に元となるトークン集合として、現在の M[X] と M[Y] が設定される。そして、M[X] と M[Y] は空集合に初期化される。この後、Xold に関するループがあり、トークンが変数 I に設定される。さらにその内部で、Yold に関するループがあり、トークンが変数 J に設定される。そこで制約条件がチェックされる。制約条件が成立した場合は、このトークンは利用可能であるので、AddToSet() を用いてトークン I と J が、M[X] と M[Y] に追加される。このとき、次の Yold に関するループでは、トークン J を再度チェックする必要がないので、RemoveFromSet() を用いて Yold からトークン J を取り除いている。以上のループを繰り返すことで、制約条件を満たす可能性のあるトークンの集合を求めることができる。

上記関数を利用したルール適用は、次のようになる。

```
routine EvaluateRule(P)
  T := Variables(P)
  C := Constraints(P)
  M := GetTokenList(T, D)
  for each L in SinglesOf(C) do
    M := CheckConstraintSingle(L, M)
  end
  for each L in DoublesOf(C) do
    M := CheckConstraintDouble(L, M)
  end
  changed := false
  for each A in Combination(M) do
    if SatisfiesConstraints(A, C) then
      N := NewNonTerminal(A, P)
      D := (D\A) ∪{N}
      changed := true
    endif
  end
  return changed
end
```

まず、GetTokenList() を用いて、各記号に当てはまるトークン集合の配列を求め M に設定する。この後、SinglesOf() を用いて C_1 の制約条件を取り出し、これに関して CheckConstraintSingle() を用いて C_1 の制約条件を満たすトークンに M を限定させる。さらに、DoublesOf() を用いて C_2 の制約条件を取り出し、これに関して CheckConstraintDouble() を用いて C_2 の制約条件を満たすトークンに M を限定させる。この 2 段階の制約条件の利用により削減されたトークン集合を利用し、Combination() で組合せを求め、最終的な制約条件のチェックを行っている。

このアルゴリズムでは、C_2 の制約条件を利用したトークンの限定の前処理に $O(n^2)$ の計算量が必要となる。また、トークンの組合せは減少してはいるものの、組合せ数のオーダに変化はないため、全体として $O(n^k)$ の計算量が必要である。前処理の部分があるため、その計算量は増えるが、これによる探索範囲の削減効果によって、全体の計算量は減少するものと期待される。なお、ルールが R_1 のルールの場合には、C_2 の制約条件は存在しないため前処理は $O(n)$ であり、全体としての計算量も $O(n)$ である。

このアルゴリズムは、簡単に 3 つ以上の記号に関する制約条件を利用する場合に拡張できる。しかし、今回の 3.3 節の調査により、このような制約条件は利用されていなかったため、C_2 の制約条件までを利用したアルゴリズムのみを提示した。

ここで具体的な例を用いて、組合せの削減を示す。例として、4.1.2 節の例における初期状態のトークンのうち、以下の 5 つのトークンまでが入力された状態を考える。

D = {1, 2, 3, 4, 5}

このとき、ルール 1 における各記号に対応するトークン集合の配列は次のようになる

M[1]={1, 2, 3}
M[2]={4, 5}

このとき、組合せを求めると次の 6 個となる。

[1,4], [1,5], [2,4], [2,5], [3,4], [3,5]　(4.4)

ここで、C_1 の制約条件 “`isValue(t.text)`” を利用することで、トークン 4 は制約条件を満たさないことが分かるので次のようになる。

M[1]={1, 2, 3}
M[2]={5}

よって、組合せは以下の 3 個に減少される。

[1,5], [2,5], [3,5]　(4.5)

提案手法では、ここでさらに C_2 の制約条件 “`c.mid == t.mid`” を利用する。M[1] と M[2] で構成されるトークンの組合せのうち、この制約条件を満たすのは、[2,5] の組合せのみである。よって、トークン 1 とトークン 3 に接続できる記号 Text に対応したトークンが存在しないことが分かるため、これらが M[1] から除外される。これによりトークン集合の配列は次のようになる。

M[1]={2}
M[2]={5}

よって、組合せは以下の 1 個に減少させることができる。

[2,5]　(4.6)

4.4 未処理トークン集合を利用した解析

インクリメンタルな解析では、トークンデータベースに変更があった場合、Parse() が呼ばれて解析が行われる。Chok らの手法では、トークンデータベースのすべてのトークンの組合せを探索していた。しかし、変更された部分はトークンデータベースの一部分であるため、不必要な解析が生じていた。これを改善するための解析手法を提案する [Iiz03a, Iiz03b]。

Chok らのアルゴリズムの終了条件から、前回の解析が終了した時点で存在するトークンデータベースのトークンの組合せでは、どのルールも適用できないと言える。つまり、ルールが適用できる可能性があるのは、変更されたトークンを含む組合せである。新しいトークンが追加された場合や、あるトークンの属性が変更された場合、そのトークンを含む組合せのみ解析すれば、トークンデータベース中のすべての組合せを探索した場合と同じ結果が得られる。

このような処理を実現するため、トークンデータベースのトークンを次の 2 種類に分ける。**処理済トークン**は、前回の解析でトークンデータベースに残ったトークンであり、それらトークンの組合せではルールが適用できないようなトークンである。**未処理トークン**は、新しく追加されたトークンや変更があったトークンであり、このトークンを含む組合せは解析が行われていない。これらを含むトークンの集合として、**処理済トークン集合** L と**未処理トークン集合** U を考える。この 2 つの集合の和集合がトークンデータベース D になる。

未処理トークン集合を利用したルール選択のアルゴリズムは、以下のようになる。

```
routine Parse()
  while NotEmpty(U)
    E := PopLowestToken(U)
    if ! ParseToken(E) then
       L := L ∪{E}
    endif
  end
end

routine ParseToken(E)
  for S := 1 to maxstrata do
    for each P in strata(S) do
      for each X in PositionOfRule(X, P) do
        if EvaluateRuleToken(E, X, P) then
          return true
        endif
      end
    end
  end
  return false
end
```

変更のあったトークンは、未処理トークン集合 U に設定されて Parse() が呼ばれる。Parse() では、PopLowestToken() を用いて未処理トークン集合 U から 1 つのトークンを取り出し、そのトークンに関して ParseToken() を呼ぶことでルール適用を行う。これを未処理トークン集合 U が空集合になるまで繰り返す。ルール適用が失敗した場合は、そのトークンを処理済トークン集合 L に設定する。なお、PopLowestToken() では、未処理トークン集合のトークンのうち、“記号に関する依存関係” で最下位にあるトークンの 1 つを取り出す。これにより、解析木の下の方で利用されうるトークンを、なるべく先に解析することができる。

ParseToken() は、引数で与えられた未処理トークンに関するルール選択を行う。ルール適用が成功した場合 true を返し、失敗した場合は false を返す。基本構造は、Chok らのアルゴリズムのルール選択と同様であり、ルールの依存関係に基づく強連結成分の階層を下から順にルール適用を試みる。ただし、各ルールに関して 1 度のみルール適用される点が異なる。これは、ルール適用に成功した場合に、着目しているトークンがトークンデータベースから取り除かれるためである。PositionOfRule() は指定されたトークンが当てはまる RHS の記号の位置を返す。これに基づき、未処理トークンが当てはまるようなそれぞれのルールの対応する各 RHS 位置について EvaluateRuleToken() でルール適用を試みる。

ルール適用に関する処理については、4.3 節で示した制約条件を利用した組合せの削減を行う手法を適用すると、以下のようになる。

```
routine EvaluateRuleToken(E, X, P)
  T := Variables(P)
  C := Constraints(P)
  M := GetTokenList(T, L)
  M[X] := {E}
  for each L in SinglesOf(C) do
    M := CheckConstraintSingle(L, M)
  end
  for each L in DoublesOf(C) do
    M := CheckConstraintDouble(L, M)
  end
  for each A in Combination(M) do
    if SatisfiesConstraints(A, C) then
      N := NewNonTerminal(A, P)
      D := (D\A) ∪{N}
      U := U ∪{N}
      L := L\A
      return true
    endif
  end
  return false
end
```

まず、初期状態のトークン集合の配列 M は、処理済トークン集合 L から決定される。そして、X 番目の RHS の記号に対応するトークン集合 M[X] として、トークン E のみを指定している。これはトークン E を含むトークンの組合せのみを探索することを示している。このトークン集合の配列に対して、C_1 の制約条件と C_2 の制約条件を利用して探索範囲を限定させる。その後に、M に基づくトークンの組合せに対して制約条件をチェックしていく。ここで、ルールが適用された場合は、各データ構造の更新が行われる。新たに生成した LHS の記号に対応するトークン N は、トークンデータベース D とともに未処理トークン集合 U に設定される。RHS の記号に対応する A の各トークンは、トークンデータベース D とともに処理済トークン集合 L から取り除かれる。

ここで、この未処理トークン集合を利用したアルゴリズムの計算量を検討する。本アルゴリズムが有効なのは、インクリメンタルな解析を行った場合である。これは、ユーザが新たに記号を追加した場合や、すでに描かれた図形を変形させた場合に該当する。この場合、変更されたトークンは 1 つである。各ルールは 1 度だけチェックされ、そのルールに対応する RHS の記号数だけ EvaluateRuleToken() が呼ばれることになる。EvaluateRuleToken() では、RHS の記号の 1 つは、引数で与えられたトークン 1 つに限定されるので、$O(n^{k-1})$ の計算量で解析が終了する。未処理トークン集合を利用しない場合は、$O(n^k)$ であったため、オーダが 1 つ下がっており、高速化が実現されている。

なお、PopLowestToken() や strata() のための依存関係を解析する作業は、ルールが与えられた時点で行われる。この処理は、Tarjan のアルゴリズムにより、ルール数や記号数とその関係に関する線形時間で終了する。これは、空間解析器の生成時に 1 度だけ行えばよい。また、インクリメンタルな解析においては、この結果のみ利用するので、解析の計算量に影響はない。

4.5 制約条件グラフを利用した解析

CMG に基づく解析が遅かった最大の原因は、ルール適用におけるトークンの組合せの探索部分にある。そこで、この探索を効率的に行うため、制約条件とトークンの関係から構成される “制約条件グラフ” を利用した解析手法を提案する [Iiz03b]。

4.5.1 ルールと制約条件

ここで、ルールにおける制約条件について検討する。3.3 節の調査で示されたとおり、C_1 の制約条件と C_2 の制約条件の 2 種類で記述されている。Balt の提案手法を用いて C_1 の制約条件を利用すれば、C_1 の制約条件をすべて満たすトークンの集合を得られた。つまり、C_1 の制約条件については、これですべて確認されるため、後で再度チェックする必要がなくなる。しかし、4.3 節で示した C_2 の制約条件の利用では、1 つでもその制約条件を満たす組合せが存在した場合、そのトークンを除外することができなかった。つまり、このような手法で C_2 の制約条件で限定した場合は、ある程度の削減効果が得られるが、C_2 の制約条件をより有効にかつ効率的に利用するためには、この手法を根本的に見直す必要

がある。

ここで、C_2 の制約条件について見直してみる。C_2 の制約条件の種類として、3.3 節の調査では、2 つのトークン間の座標の一致、x 座標や y 座標の一致、領域の包含関係などの図形的位置関係が多く利用されていた。逆に、多くのルールでは、ほとんどの場合これらの制約条件で記述されていた。そこで、これらの制約条件を満たすトークンの組合せを高速にチェックできれば、全体として解析時間が高速化されることが期待される。

このような制約条件の特徴として、一方のトークンが決定された場合に、もう一方のトークンがただ 1 つに定めることができるという点が挙げられる。たとえば、計算の木の定義で使用されていた“`c.mid == t.mid`”という制約条件では、円の中心とテキストの中心が一致するという意味である。このような円とテキストの組合せは、1 対 1 の関係になると考えられる。つまり、1 つの円の上に複数のテキストを置くことは計算の木という図形言語ではありえない。同様に、1 つのテキストに重なる円も 1 つのみ対応する。このように、C_2 の制約条件がトークンの対応関係を定めるものとなっている。この特徴をふまえて、トークンの関係に着目して制約条件を利用することを考える。

Chok らのアルゴリズムでは、

1. まず組合せを考える
2. その組合せが制約条件を満たすかチェックする

という流れになっていた。提案する手法では、この逆のアプローチをとり、

1. ある記号に対応する 1 つのトークンに着目する
2. このトークンに関連する C_2 の制約条件を 1 つ選択し、この制約条件を満たす、他の記号に対応するトークンを探す。
3. 求めたトークンとの制約条件を満たす、さらに他の記号に対応するトークンを探すことで組合せを求める。

という流れで探索を行う。この際、先に述べたように、これまで求めたトークンに対応するトークンが、制約条件によって求めることができる。このようにして、制約条件を満たすようなトークンを順次求めて行くことで、最終的にすべての制約条件を満たす組合せを探索することが可能となる。

4.5.2　制約条件グラフ

前述のような探索を実現するため、制約条件と各記号の関係について把握しておく必要がある。それぞれのルールにおいて、RHS の記号をノードとし、C_2 の制約条件をエッジと考えることで、RHS の記号と C_2 の制約条件に関するグラフが構成できる。このグラフを**制約条件グラフ**と呼ぶ。計算の木の例におけるルール 2 の制約条件グラフは、図 4.5 のようになる。なお、各ノードの名前として、RHS の記号に対応する変数名で呼ぶ。たとえば、5 番目の RHS 記号 Node に対応するは変数 `n1` であるのでノード `n1` と呼ぶ。この図で

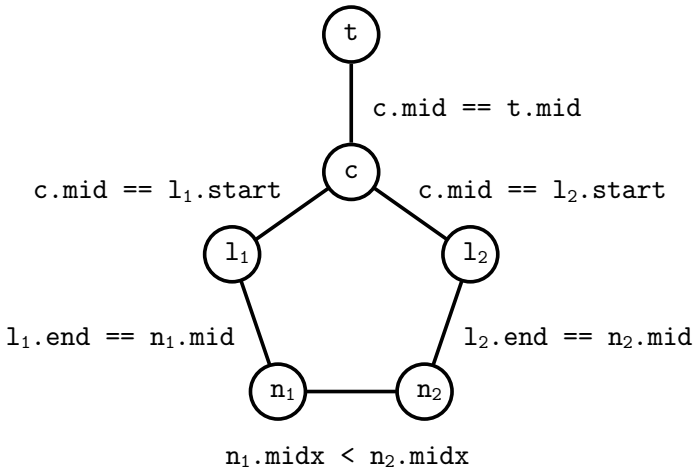

図 4.5: 制約条件グラフ

は、ノード n1 は、ノード l1 とノード n2 の 2 つのノードとつながっていることが分かる。これらは、“l1.end == n1.mid”と“n1.midx < n2.midx”という C_2 の制約条件をエッジとしている。

なお、制約条件によって関連付けられていない記号の場合、対応するノードには 1 つもエッジがつながらない、独立したノードとして存在することになる。つまり、必ずしも連結グラフが得られるわけではない。また、ある記号間に複数の制約条件が設定されていた場合は、その記号間のエッジは、それら制約条件の論理積をとったものが、そのエッジに対応する制約条件となる。

ここで、エッジに対応する制約条件を 2 つに分類する。1 つは、4.5.1 節で述べたような、一方のトークンが決定された場合に、制約条件を満たすもう一方のトークンが 1 つに定まるような制約条件である。このような制約条件を**強い制約条件**と呼ぶ。もう 1 つは、この逆で、一方のトークンが決定された場合に、制約条件を満たすもう一方のトークンが 1 つに定めることができない制約条件である。このような制約条件を**弱い制約条件**と呼ぶ。なお、一方のトークンが決定された場合に、対応するトークン数が 2 つや 3 つといった、図形数に依存しない定数値で抑えられる場合も、その制約条件を強い制約条件と呼ぶ。

4.5.3　探索方法

制約条件グラフにおいて、エッジに従ってすべてのノードを訪問することにより、トークンの組合せを探索することができる。つまり、組合せの探索を、グラフ探索に置き換える。このように、制約条件を利用しながらトークンを 1 つずつ求めていくことにより、トークンの組合せを効率的に求めることができる。

ノードを訪問した場合は、そのノードに対応するトークンを 1 つ選び、そのトークンが制約条件を満たせば次のノードを訪問する。このようにして、すべてのノードを訪問できた場合は、そのときの各ノードに対応するトークンがそのルールを満たすトークンの組合せとなる。

表 4.1: 各ノード訪問でチェックされる制約条件

訪問順序	ノード	チェックされる制約条件
1	c	なし
2	t	isOperand(t.text), c.mid == t.mid
3	l1	c.mid == l1.start
4	l2	c.mid == l2.start
5	n1	l1.end == n1.mid
6	n2	l2.end == n2.mid, n1.midx < n2.midx

各ノードを訪問した際に、そのノードに対応する記号を含む制約条件がチェックされる。この制約条件は大きく 3 種類に分けられる。1 つは、その記号に関する C_1 の制約条件である。2 つめは、その記号に関する C_2 の制約条件である。これは、エッジをたどってこのノードに訪問したわけであるので、そのようなエッジに対応する制約条件が必ず 1 つ存在する。このエッジに関する制約条件は、このノードのトークンと、このエッジにつながる訪問元のノードに対応するトークンとの間で調べることができる。また、訪問に利用したエッジ以外のエッジが存在する場合もある。つまり、訪問したノードとすでに訪問したノードの間に複数のエッジが存在する場合である。このようなエッジに関する制約条件も同時にチェックすることができる。3 つめは、これ以外の制約条件である。これは、訪問したノードとすでに訪問したノードで構成される 3 つ以上の記号を含むような制約条件である。このような制約条件は、3.3 節の調査では存在しなかったが、“`a.val == b.val + c.val`”のような制約条件も考えられ、このような場合もここでチェックされる。

例として、ノード c から訪問を開始した場合に、各ノードを訪問した際にチェックされる制約条件を表 4.1 に示す。

以上のように制約条件をチェックすることで、そのノードに対応したトークンを選ぶことができる。このようにして、すべてのノードを訪問することができれば、すべてのノードに対してそれぞれトークンを求めることができ、さらにそのトークンの組合せは、ルールにおけるすべての制約条件を満たすことになる。

なお、各ノードで制約条件を満たすトークンが存在しなかった場合は、その時点でのトークンの組合せではこのルールを適用できないことが分かるので、これまでの訪問をバックトラックしながら、新たな組合せを探索する。

ここで、ノードの訪問順序は、既に訪問されたノードからエッジをたどっていくことにより決定される。複数のノードが訪問可能である場合には、そのうちの 1 つを選んで訪問する。たとえば、計算の木におけるルール 2 の訪問順序は、表 4.2 のようなものが考えられる。なお、制約条件グラフが連結していない複数のグラフで構成された場合は、まず、開始ノードから連結成分のノードを探索し、次に連結していないグラフのノードのうち 1 つを選んで訪問を続ければよい。

ここで、各ノードを訪問する際に利用したエッジについて検討する。このエッジに対応する制約条件は、C_2 の制約条件であり、前述のとおり、これは図形間の位置関係を指定す

表 4.2: 訪問順序の例

開始ノード	訪問順序
c	c → t → l1 → l2 → n1 → n2
t	t → c → l1 → l2 → n1 → n2
l1	l1 → c → n1 → t → l2 → n2
l2	l2 → c → n2 → t → l1 → n1
n1	n1 → l1 → n2 → c → l2 → t
n2	n2 → l2 → n1 → c → l1 → t

るような強い制約条件である場合が多い。このような場合では、訪問先のノードに当てはまるトークンが 1 つに定まるので、訪問における分岐は発生しない。これにより、効率的にグラフが探索でき、制約条件を満たすトークンの組合せを高速に求めることができる。

制約条件グラフのエッジを利用した訪問は、制約条件を利用することで訪問先のノードにおけるトークンを限定させ、探索する組合せ数を削減している。また、トークンの組合せを求める際に、なるべく早い段階で制約条件を利用してトークンをチェックすることが可能となる。なお、エッジが存在しない場合でも、3 つ以上の記号に関する制約条件によって、訪問先のノードにおけるトークンが限定できる場合は、これを利用して訪問することができる。

制約条件グラフを用いて組合せを探索する場合、すべてのノードから探索を開始するのは無駄である。最低限の探索で制約条件を満たす組合せを求めるために、適切な**探索開始ノード**を決定しなければならない。

4.4 節で述べたように、トークンデータベースが変更された場合に探索すべき範囲は、未処理トークン集合のトークンを含む組合せである。これにより、未処理トークンを 1 つ抜き出し、そのトークンに関して探索すればよい。つまり、探索開始ノードとして未処理トークンに対応するノードを選び、そのノードの**探索開始トークン**として設定して探索を開始すれば、トークンデータベースにおける、この未処理トークンに関するすべての可能な組合せをチェックすることになる。

4.5.4 アルゴリズム

以上の議論をまとめると、解析アルゴリズムは次のようになる。なお、ルール選択に関するアルゴリズムは 4.4 節で示したものと同一であるので、ここではルール適用の部分に関してのみ説明を行う。

```
routine EvaluateRuleToken(E, X, P)
  W := InitialTokenList(X, E)
  return Visit(P, W, X)
end
```

```
routine Visit(P, W, X)
  C := Constraints(P, W, X)
  if ! SatisfiesConstraint(W, C) then
    return false
  endif
  if Finished(W) then
    N := NewNonTerminal(W, P)
    D := (D\W) ∪{N}
    U := U ∪{N}
    L := L\W
    return true
  endif
  Y := NextNode(P, W)
  for each S in GetTokens(Y, L) do
    W[Y] := S
    if Visit(P, W, Y) then
      return true
    endif
  end
  W[Y] := nil
  return false
end
```

EvaluateRuleToken() では、第 1 引数に探索を開始するトークン、第 2 引数に探索を開始する RHS の記号の位置、第 3 引数に対象としているルールが設定されて呼び出される。ここで利用されている W は、各ノードに対応するトークンの列を保持する配列である。たとえば、W[1] は 1 番目の RHS の記号に対応するノードのトークンである。訪問済みのノードに対応する位置には決定されたトークンが設定され、未訪問のノードに対応する位置には *nil* が設定される。InitialTokenList() では、この W を初期設定しており、探索開始ノードの対応した位置に探索開始トークンを設定し、その他の部分には *nil* を設定する。たとえば、計算の木のルール 2 に関する 2 番目の RHS の記号 Text を探索開始ノードとした場合、W は長さ 6 の配列となり、W[2] に探索開始トークンが設定され、そのほかの配列要素には *nil* が設定される。

Visit() は、制約条件グラフを探索する部分である。第 1 引数に対象としているルール、第 2 引数に各ノードに対応するトークンの配列、第 3 引数に現在のノードの位置が設定されて呼び出される。この関数は、探索が成功しルールが適用できた場合 true を返し、探索が失敗した場合は false を返す。Visit() では、最初に、現在のノードに関する制約条件のうち、訪問済みのノードで構成される制約条件を Constraints() で取得し、制約条件が成り立つかチェックする。ここで制約条件が成り立たなければ、その組合せではルールを適用できないので、Visit() は false を返して訪問元のノードに戻る。

制約が成り立った場合は、まず、終了条件をチェックする。すべてのノードが訪問された場合、つまり、W においてすべてのトークンが設定されている場合は訪問が終了する。その場合の W は RHS の記号に対応し、この W はすべての制約条件を満たしていることになる。よって、ルールを適用できるので、新しい非終端記号を生成する。この際は、4.4 節の未処理トークン集合の扱いと同様に処理される。新たに生成した LHS の記号に対応するトークン N は、トークンデータベース D とともに未処理トークン集合 U に設定される。RHS の記号に対応する W の各トークンは、トークンデータベース D とともに処理済トークン集合 L から取り除かれる。

探索が終了していない場合は、次のノードを NextNode() で決定し訪問する。次のノードはすでに訪問されたノードとエッジでつながっているノードである。なお、3 つ以上の記号に関する制約条件においても、1 つを除いたすべての記号に対応するトークンが決定されている場合は、そのノードを訪問することができる。なぜならば、その制約条件を利用することで訪問先ノードに対応するトークンを決定できるからである。つまり、C_2 までの制約条件の場合は制約条件グラフに基づくが、より一般的には、“制約条件を利用してトークンを限定可能なノード” を選択すればよい。これらすべてに対して訪問できるノードがない場合は、未訪問のノードのうち任意の 1 つのノードが選択される。この場合は、制約条件を利用したトークンの限定ができないことになる。その後、次の訪問ノードに対応するトークンを GetTokens() で処理済トークン集合のトークンから取得する。それぞれのトークンに対して W に設定した後、Visit() で次のノードを訪問し探索を行う。すべてのトークンで探索が失敗した場合は、W に設定されたトークンを *nil* で取り消した後に訪問元ノードに戻る。

4.5.5　探索の例

ここで具体的な例を用いて、制約条件グラフを利用した解析の例を示す。例として、4.1.2 節の例と同じデータによる処理の流れを示す。このとき、トークンデータベース D と未処理トークン集合 U、処理済トークン集合 L は以下のようになっている。

D = {1, 2, 3, 4, 5, 6, 7, 8}
U = {1, 2, 3, 4, 5, 6, 7, 8}
L = {}

まず、[1] のトークンが Parse() によって選択される。これにより、[1] が未処理トークン集合 U から取り除かれ、これを引数として ParseToken() が呼ばれる。ここで、最初にルール 1 が選ばれる。[1] のトークンは、記号 Circle であるので、1 番目の RHS の記号に該当する。よって、RHS の記号の位置を 1 として、EvaluateRuleToken() を呼び出す。EvaluateRuleToken() では、各ノードに対応するトークンの配列 W として以下のように初期設定がされる。

W = [1, nil]

これを引数として Visit() が呼ばれる。まず、制約条件がチェックされるが、記号 Circle にのみ関する制約条件はないので適合する。次に終了条件がチェックされるが、W の 2 番目

の要素、つまり記号 Text に対応するトークンが決定していないので次に進む。ここでは、制約条件 “`c.mid == t.mid`” を利用してノード `t` に到達できるため、ノード `t` が選択される。次に、ノード `t` に対応する処理済トークンのループに移るが、これに対応するトークンは、何も存在しないので、Visit() は false を返して戻る。ParseToken() に戻ると、今度は、ルール 2 に対するルール適用が行われる。しかし、これも同様に失敗して、Parse() に戻る。この状態での各トークン集合は以下のようになっている。

D = {1, 2, 3, 4, 5, 6, 7, 8}
U = {2, 3, 4, 5, 6, 7, 8}
L = {1}

次に同様にして、[2] のトークンが選択されるが、[1] のトークンと同じ流れをたどり、ルール適用が失敗する。[3] のトークンも同様である。これにより、各トークン集合は以下のようになる。

D = {1, 2, 3, 4, 5, 6, 7, 8}
U = {4, 5, 6, 7, 8}
L = {1, 2, 3}

次に [4] のトークンが未処理トークン集合 U から抜き出される。ParseToken() でルール 1 が選択され、EvaluateRuleToken() が呼び出される。ノードに対応するトークンの配列 W として以下のように初期設定される。

W = [nil, 4]

Visit() が呼ばれると、まず制約条件がチェックされる。ここでは、記号 Text に対する制約条件 “`isVlaue(t.text)`” を満たさないので、false を返す。ParseToken() まで戻り、次にルール 2 が選択される。同様にして、EvaluateRuleToken() が呼び出される。このときのトークンの配列 W として以下のように初期設定される。

W = [nil, 4, nil, nil, nil, nil]

Visit() が呼び出されると、今度は、制約条件 “`isOperator(t.text)`” を満たすので、次へ進む。この状態では終了していないので、次のノードへ探索を始める。ここで、表 4.2 に従い、ノード `c` に訪問する。対応するトークンは、処理済トークン集合 L より、[1], [2], [3] が選ぶことができる。ここで、[1] を選んで Visit() で訪問すると、これは制約条件を満たす。よって、さらに次のノードへ訪問を試みる。ここでは、ノード `11` が選ばれるが、これに対応するトークンは存在しないので、バックトラックする。次に、記号 Circle に [2] と [3] を当てはめるが、これは制約条件を満たすことができない。これらにより、すべての訪問が失敗することになり、Parse() まで戻る。この状態における、各トークン集合は以下のようになる。

D = {1, 2, 3, 4, 5, 6, 7, 8}
U = {5, 6, 7, 8}
L = {1, 2, 3, 4}

次に、[5] のトークンが選択され、ParseToken() が呼ばれる。ここでは、ルール 1 が選択され、EvaluateRule() が呼ばれる。ノードに対応するトークンの配列 W として以下のように初期設定される。

W = [nil, 5]

この場合、テキストが数字であるので制約条件を満たし、次のノード c を訪問する。対応するトークンは、処理済トークン集合 L より、[1], [2], [3] が選ぶことができる。ここで、[1] が選ばれ、W に設定されて訪問される。この場合、中心が一致しないので制約条件を満たさない。次に、[2] が選ばれて、W に次のように設定される。

W = [2, 5]

この状態の場合、制約条件を満たす。また、W の要素がすべて設定されているため、非終端記号が生成される。これに対応するトークンは、[9] である。これが、未処理トークン集合 U とトークンデータベース D に追加される。また、RHS の記号となる W の各要素は処理済トークン集合 L とトークンデータベース D から削除される。これにより、各トークン集合は以下のようになる。

D = {1, 3, 4, 6, 7, 8, 9}
U = {6, 7, 8, 9}
L = {1, 3, 4}

Visit() は true を返し、Parse() まで戻り、次のトークンを選択する。ここでは、[6] が選ばれて同様の処理を行う。この場合もルール 1 が適用され、各トークン集合は以下のようになる。

D = {1, 4, 7, 8, 9, 10}
U = {7, 8, 9, 10}
L = {1, 4}

次に、[7] のトークンが選択される。ParseToken() では、ルール 1 では記号 Line に対応する RHS の記号がないので、次にルール 2 が選ばれる。記号 Line の位置として 3 番目の位置が最初に選ばれて EvaluateRuleToken() が呼ばれる。このときのトークンの配列 W として以下のように初期設定される。

W = [nil, nil, 7, nil, nil, nil]

Visit() では、次にノード c に対応するトークンを訪問する。ここでは、処理済トークン集合 L に [1] しかない。これを W に設定し、Visit() で訪問する。これは制約条件を満たすため、次のノード n1 を訪問する。しかし、処理済トークン集合 L に記号 Node のトークンは存在しないので失敗する。これにより、ParseToken() まで戻る。次に記号 Line の位置として 4 番目の位置が選ばれて探索されるが、これも失敗する。次に [8] のトークンが選択されるが、これについても同様に訪問が失敗する。これにより、各トークン集合は以下のようになる。

D = {1, 4 7, 8, 9, 10}
U = {9, 10}
L = {1, 4, 7, 8}

次に、[9] のトークンが選択される。ルール選択で、ルール 1 は対応する記号がないため、ルール 2 が選択され、5 番目の位置で EvaluateRuleToken() が呼ばれる。このときのトークンの配列 W として以下のように初期設定される。

W = [nil, nil, nil, nil, 9, nil]

Visit() では、次のノード l1 が選ばれる。これに対応する処理済トークンは、[7] と [8] が存在する。まず [7] が選択されて、次に訪問する。このときのトークンの配列 W として以下のように設定される。

W = [nil, nil, 7, nil, 9, nil]

この訪問は成功し、次のノード n2 を訪問する。しかし、これに対応するトークンが処理済トークン集合 L に存在しないため、訪問は失敗しバックトラックする。次に、ノード l1 にトークン [8] を設定してノード n2 を再び訪問する。しかしこれも失敗し、Parse() まで戻る。これにより、各トークン集合は以下のようになる。

D = {1, 4 7, 8, 9, 10}
U = {10}
L = {1, 4, 7, 8, 9}

次に、[10] のトークンを選び、探索を行う。まずは、5 番目の位置で EvaluateRuleToken() が呼ばれる。ノード l1 では、[7] が当てはまらないが、[8] は制約条件を満たす。よって次のノード n2 を訪問する。これに対応するトークンは [9] のみである。このときのトークンの配列 W として以下のように設定される。

W = [nil, nil, 8, nil, 10, 9]

これは、制約条件“n1.midx < n2.midx”を満たさないので失敗し、バックトラックする。これにより、ParseToken() まで戻る。次に、RHS の記号の 6 番目の位置で EvaluateRuleToken() が呼ばれる。この訪問では、まずノード l2 の訪問で、[7] が選択されるがこれは失敗する。[8] が選択されると、制約条件を満たすので、ノード n1 を訪問する。ここでは、[9] しか存在しないが、これは、制約条件を満たす。このときのトークンの配列 W として以下のように設定される。

W = [nil, nil, nil, 8, 9, 10]

この後、ノード c を訪問するが、対応する処理済トークンは [1] のみである。これは制約条件を満たすので、次にノード l1 を訪問する。ノード c からノード l1 を訪問する際に利用するエッジの制約条件“c.mid == l1.start”がチェックされるとともにノード l1 とノード n1 の制約条件“l1.end == n1.mid”も同時にチェックされる。ここでは、[7] のみが対応し制約条件を満たす。最後に、ノード t が訪問される。ここで W に [2] が設定され次のようになる。

W = [1, 2, 7, 8, 9, 10]

この組合せは、制約条件を満たし、要素のすべてが設定されている。これにより、ルールが適用され、非終端記号が生成される。ここで、各トークン集合が更新され以下のようになる。

D = {11}
U = {11}
L = {}

ここで、[11] が選択されて訪問が開始する。この訪問は対応するトークンが存在しないためすべて失敗する。これにより、各トークン集合は以下のようになる。

D = {11}
U = {}
L = {11}

ここで、未処理トークン集合 U が空集合となったので、解析が終了する。

4.5.6　計算量

ここでは、トークンデータベース D にあるトークン数 n に対する計算量について検討する。1 つのルールに対するルール適用では、探索開始ノードに対して、各ノードが訪問される。訪問するノードは、RHS の記号数 k から探索開始ノードを除いた $k-1$ 個のノードである。各訪問では、対応するトークンにより最大 n 個の分岐がある。これにより、計算量は最悪の場合 $O(n^{k-1})$ になる。これは、未処理トークン集合を利用した場合の計算量と同じであり、Chok らのアルゴリズムの $O(n^k)$ よりは改善がなされている。なお、記憶領域については、探索のために特別な領域を利用していないので $O(1)$ である。

ただし、一般的な場合の計算量は最悪の場合に比べ非常に良い結果になる。これは、前述のとおりノードを訪問した際に、制約条件によって訪問できるトークンが 1 つに決定される場合が多くあり、このケースでは訪問に分岐が発生しないためである。訪問に利用したすべてのエッジに関する制約条件がこのような場合であれば、計算量は $O(n)$ になる。

訪問に分岐が発生するのは、訪問に利用するエッジのうち、対応する制約条件が弱い制約条件である場合である。つまりトークンが 1 対 1 に決定できず、対応するトークンが複数存在するような場合である。残念ながら、このようなエッジは存在する。

計算の木のルール 1 では、ノード `c` とノード `t` を結ぶエッジは座標の一致を制約条件としており、トークンに決定性がある。また、ルール 2 では、ノード `n1` とノード `n2` の間のエッジ以外は座標の一致条件が利用されている。これらの制約条件はトークンの決定性がある強い制約条件であり、訪問に分岐が発生しない。ノード `c` と、ノード `l1` とノード `l2` を結ぶエッジに関しては、左右の葉と接続するために、円に接続する線は 2 つ存在する。しかし、これはたかだか 2 つであるので、計算量のオーダには変化はない。一方、ノード `n1` とノード `n2` の間のエッジについてはトークンに決定性のない弱い制約条件である。こ

れは、このエッジの制約条件“n1.midx < n2.midx”が単に左右の位置関係のみ着目しているので、ある記号 Node のトークンの右に存在するような記号 Node のトークンは最大 $O(n)$ 個存在するためである。よって、表 4.2 で示した訪問順序の場合は、このエッジを利用した訪問がなされているため計算量は $O(n^2)$ となる。しかし、この訪問順序を適切に選び、ノード n1 とノード n2 の間のエッジを使わない訪問をした場合の計算量は $O(n)$ とすることができる。

4.6　前処理を加えた探索

上述のように、計算量は制約条件と訪問順序に依存する。制約条件は変更できないが、訪問順序を適切に選ぶことによりさらなる高速化が期待できる。しかし、ルールに対して静的な解析を行うだけでは、制約条件を満たすトークンの組合せ数を予想できないため、最適な訪問順序を決定できない。そこで、制約条件グラフにおける各エッジに対して、トークンの対応関係を事前にチェックすることで、組合せが少ないノードを訪問するようにする。

各エッジは、トークンレベルで考えると、双方のノードに対応したトークンのうち、制約条件を満たすトークンの組合せと考えることができる。グラフを探索する前にこの組合せを求めておくことにより、以後の探索で対応するトークンを簡単に求めることができる。また、組合せが少ないエッジを選ぶことにより探索範囲を減少させることが期待できる。

この前処理は、各ルールにおいて組合せを探索する前に行う。各エッジにおけるトークンの組合せは、ノードに対応するトークンが増減した場合にのみ変化がある。よって、探索ごとに組合せを求めるのではなく、それぞれの制約条件グラフで組合せを保持しておき、変更のあった部分のみ再計算することで、インクリメンタルにトークンの組合せを管理することができる。

トークンが追加された場合は、そのトークンを探索開始ノードとする解析が行われるので、ここで探索を行う前に各エッジにおけるトークンの組合せを再計算すればよい。変化のあるエッジは、一方に追加されたトークンに対応するノードを含むエッジである。追加されたトークンを含まない組合せはすでに計算されているので、追加されたトークンに対応する組合せのみをチェックすればよい。つまり、$O(n)$ の計算量で処理を行うことができる。なお、組合せを保持するために $O(n^2)$ の記憶領域を必要とする。

トークンが削除された場合、または、ルールが適用され RHS のトークンとして利用されて D から削除された場合には、各エッジのトークンの組合せからこのトークンを含む組合せを削除する。

このようにして得られた組合せは、ノードを訪問する際の訪問先のトークンを決定する際に利用できる。さらに、訪問順序を決定する際にも利用する。次の訪問先を決定する際に、これまでに決定したトークンからたどれるノードのうち、組合せ数が最少になるノードを選べばよい。このことにより、全体として探索する組合せを減らすことができ、探索を効率的に行うことができる。

4.7 属性値テーブルの利用

前節で述べた前処理には、対応するすべてのトークンに対して組合せを求めていたので、$O(n)$ の計算量が必要であった。制約条件によっては、各エッジの対応するトークンの組合せを高速に求めることが可能である。ここでは、2 つのトークンの属性値が等しいといった制約条件について検討する。このような制約条件は、3.3 節の調査結果で示されたように非常に多く存在する。

属性値が等しいという条件であるので、ノードに関するトークンのすべてに対して、属性値をキーとしてハッシュ表にマッピングを登録することにより、もう一方のノードのトークンと属性値が等しくなるようなトークンを $O(1)$ で探索することが可能となる。

前処理では追加されたトークンをハッシュ表に登録する。トークンが削除された場合は、ハッシュ表から削除する。これらは、$O(1)$ の計算量である。次のノードを訪問する際には、このハッシュ表を利用し、現在のトークンの属性に対応するトークンを選べばよい。これも $O(1)$ の計算量で行うことができる。なお、属性値を保持するために $O(n)$ の記憶領域が必要である。

以上のような処理を行うことで、このような制約条件を利用したエッジでは高速な探索が可能になる。このようなエッジを用いてすべての訪問が行うことができ、トークンが 1 対 1 の対応関係にあった場合、1 つのルールに対する解析は $O(1)$ で処理が行える。

なお、このハッシュ表を用いる手法は、1 つの属性だけでなく、複数の属性が互いに一致するという条件でも利用可能である。また、属性値が等しいという制約条件のほかにも、同様の処理を行うことで対応するトークンを高速に求められる。たとえば、一方の点がもう一方のある直交領域に含まれるといった条件の場合、領域に対応する点は、領域に対する直交領域探索を行えばよい。これは、領域木にフラクショナルカスケーディングを用いることで、$O(\log n)$ の計算量で求めることが可能である [Lue78]。ほかにも、2 点間の距離がある値以下であるという条件などにも応用することが可能である。

これら属性値に関するテーブルを利用できる場合は、できるだけこれを利用した探索を行うことで高速な解析が実現できる。しかし、制約条件が複雑な計算を必要とする場合などは、このような手法を用いることができない。その場合は、テーブルを利用できない制約条件に関するエッジを利用しない訪問順序を検討する。もし、そのような訪問順序が存在すれば、複雑な制約条件のチェックは、他のエッジを利用した訪問の後で確認できる。一方、そのような訪問順序が存在しない場合は、そのエッジに関してのみ前節で述べた前処理を利用して探索することで、一定の高速化が期待できる。

4.8 実験

提案したアルゴリズムを用いた図形言語の解析について実験を行った。実験に利用したのは 3.3 節で示した図形言語の例のうち、以下の 3 つである。これらを選んだ理由は、トークン数と計算時間の関係を調査するため、簡単にトークン数を増やして実験できるからである。なお、折れ線の定義を図 4.6 に、リスト構造の定義を図 4.7 に示す。

```
l:Line ::= l1:Line, l2:Line where (
  l1.end == l2.start
) {
  l.start = l1.start
  l.end   = l2.end
}
```

図 4.6: 折れ線の定義

```
n:List ::= r:Rect, l:Line where (
  r.top_right == l.start
  r.bottom_left == l.end
) {
  n.x = r.left
  n.y = r.mid_y
}
```

```
n:List ::= r:Rect, t:Text,
    l:Line, s:List where (
  r.mid == t.mid
  r.right == l.start_x
  r.mid_y == l.start_y
  l.end_x == s.x
  l.end_y == s.y
) {
  n.x = r.left
  n.y = r.mid_y
}
```

図 4.7: リスト構造の定義

- 計算の木
  入力図形として、平衡木になるような計算の木を利用した。
- 折れ線
  入力図形として、1 つにつながった折れ線を利用した。
- リスト構造
  入力図形として、一列につながった線形リストを利用した。

入力順序により解析時間が変化するため、入力図形に対応するトークン（線分や文字列など）列をでたらめな順序で解析器に与え、20 回の平均値を取得した。それぞれのトークンを 1 つずつ解析器に渡すことにより、インクリメンタルな解析の速度を比較する。すべてを解析するのに要した計算時間と入力トークン数の関係について比較を行った。以下の 5 種類の解析手法を比較する。

- 4.1.1 節で示した Chok らの 1995 年のアルゴリズム
- 4.3 節で示した制約条件を利用した組合せの削減手法を用いた解析
- 4.5 節で示した制約条件グラフを利用した解析
- 4.6 節で示した制約条件グラフを利用した解析に前処理を加えた解析

- 4.7 節で示した制約条件グラフを利用した解析にハッシュ表を利用した解析

実験に利用した環境は以下のとおりである。

- Linux, kernel 2.4
- Athlon XP 3000+, 896MB Memory
- Java 2 SDK 1.4.2

実験では、Java インタプリタの実行時オプションに“`-Xint`”を指定し、interpreted mode で実行した。これは、Java の HotSpot 機能を無効にするオプションである。HotSpot が有効の場合、実験結果にバラつきが出るためこのオプションを指定した。

計算の木に対する総解析時間に関する実験では、図 4.8 に示すような結果になった。なお、グラフは両対数グラフとなっており、横軸が入力されたトークン数、縦軸が総解析時間（ミリ秒）を示している。Chok らのアルゴリズムでは総解析時間が $O(n^7)$ になり、インタラクティブなシステムでは利用できないことが分かる。また、制約条件を利用した組合せ数の削減手法を用いた場合、Chok らのアルゴリズムに比べ高速化されている。しかし、計算量のオーダは変わっていない。一方、制約条件グラフを利用した組合せの探索を行った場合、$O(n^3)$ で処理が終了している。訪問を効率化するために前処理を行った場合は $O(n^2)$ に高速化されている。また、ハッシュ表を利用した場合の解析では、$O(n)$ で処理が終わっている。これは、1 つのトークンの追加が $O(1)$ の処理時間で行われたことを示している。

図 4.8 の実験における各入力順序による総解析時間の分布は、図 4.9 のような結果になった。各手法の 20 回分の解析時間がそれぞれ点で示されており、平均値が線で表現されている。これより、Chok らの手法と制約条件を利用して組合せを削減した手法では、分布が非常に大きいことが分かる。つまり、入力順序に非常に敏感なアルゴリズムである。一方制約条件グラフを利用した 3 つのアルゴリズムでは、分布が小さく収まっている。なお、10 ミリ秒や 20 ミリ秒の値にデータが多く取得されているのは、本実験において取得された最小時間単位が 10 ミリ秒であったためである。

図 4.8 の実験における記憶領域については、図 4.10 のような結果になった。なお、グラフは両対数グラフとなっており、横軸が入力されたトークン数、縦軸が記憶領域（KB）を示している。実験は、Java のオプションに“`-Xms256m -Xmx256m`”を与えて実行した。正しい使用量を取得するため System.gc() を実行した後、Runtime.getRuntime().totalMemory() の値と Runtime.getRuntime().freeMemory() の値との差を記憶領域として取得した。実験中に約 20 回記憶領域を取得し、その最高値を示している。前処理を行った場合の記憶領域は、$O(n^2)$ 必要であることが分かる。そのほかのアルゴリズムでは $O(n)$ の記憶領域で動作していることが分かる。

折れ線に対する実験は図 4.11、リスト構造に対する実験は図 4.12 に示すような結果になった。この 2 つに関しては、前処理を行った場合とそうでない場合の総解析時間がほぼ同じ結果になった。これは、制約条件のすべてが強い制約条件であるので、解析時間に訪問順序が影響しなかったためである。しかし、ハッシュ表を用いることで $O(n)$ の解析が実

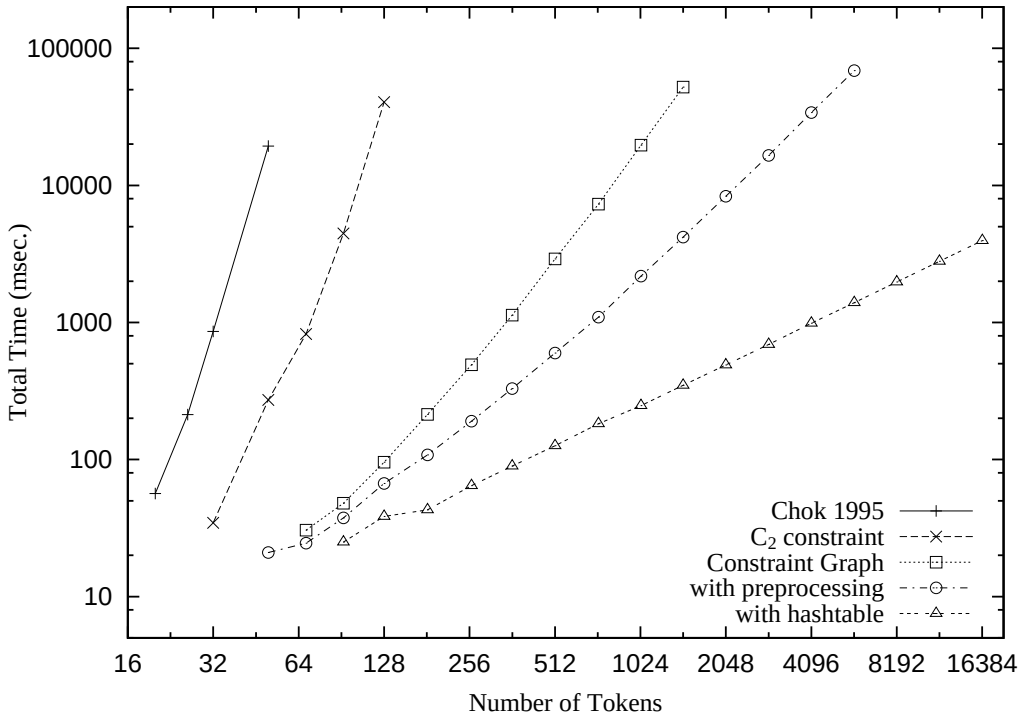

図 4.8: 計算の木の解析における総解析時間

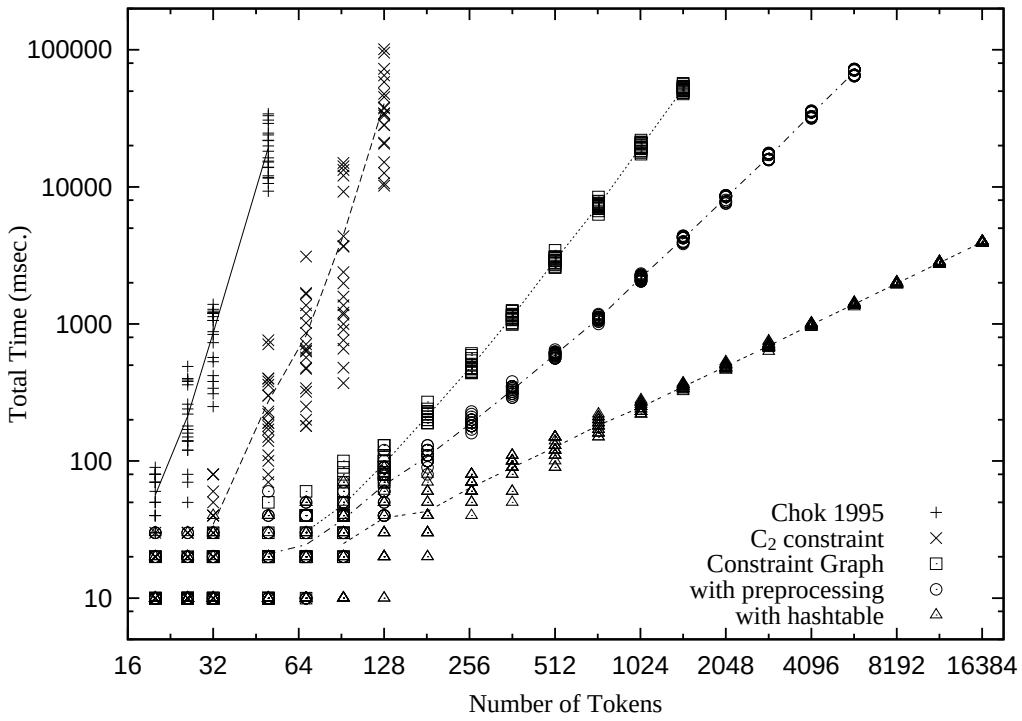

図 4.9: 計算の木の解析における総解析時間の分布

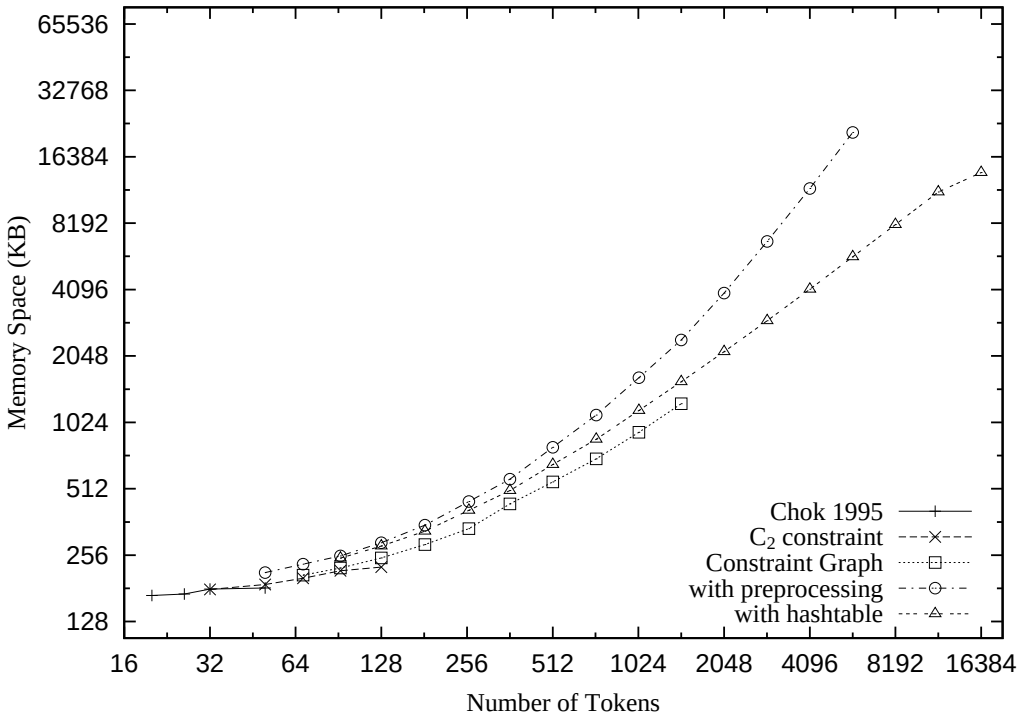

図 4.10: 計算の木の解析における記憶領域

現されている。折れ線に対する実験では、制約条件が 1 つしかなく、それを満たせばルールが適用されるため、制約条件を利用した組合せの削減手法と Chok らのアルゴリズムに差が現れなかった。

なお、参考のため Java のオプションに“`-Xint`”を指定せずに、“`-server`”オプションを指定し、Java HotSpot Server VM を利用した場合の実験結果を示す。計算の木に対する実験は図 4.13、折れ線に対する実験は図 4.14、リスト構造に対する実験は図 4.15 に示すような結果になった。一般に HotSpot を無効にした場合に比べ高速に解析が行えている。グラフの歪みが一部で見られるが、これは HotSpot の影響であると思われる。

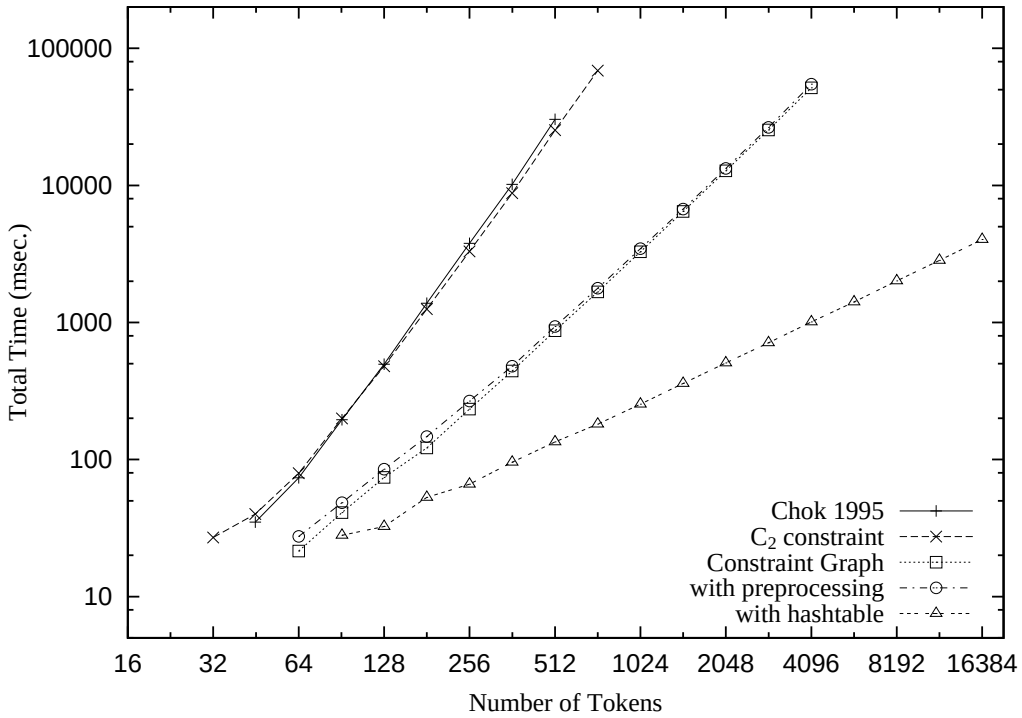

図 4.11: 折れ線の解析における総解析時間

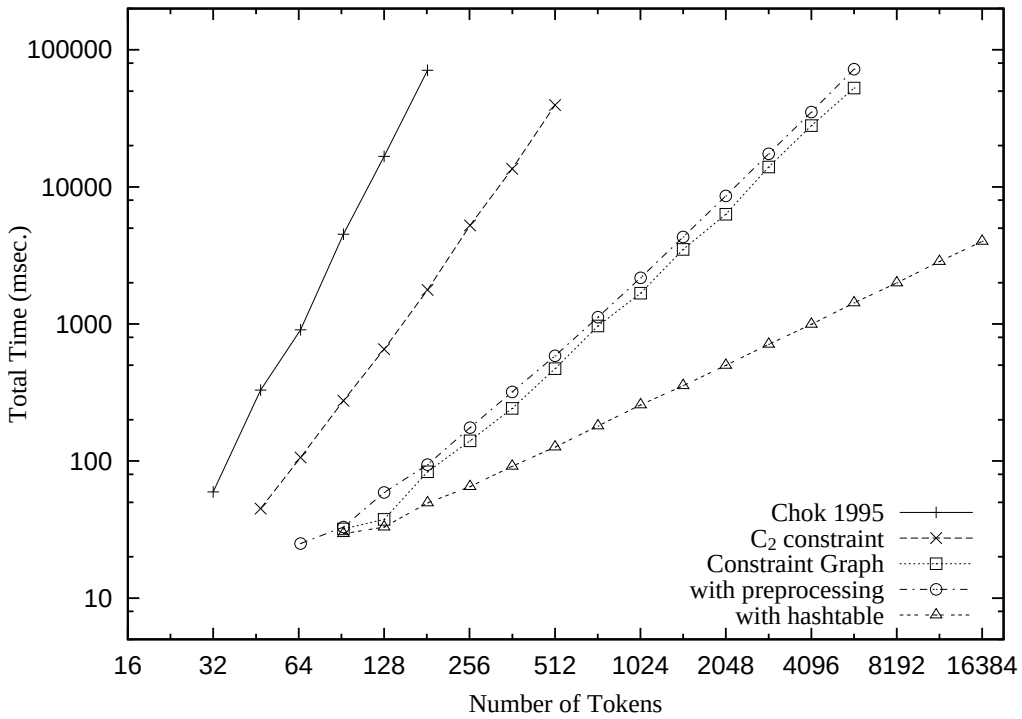

図 4.12: リスト構造の解析における総解析時間

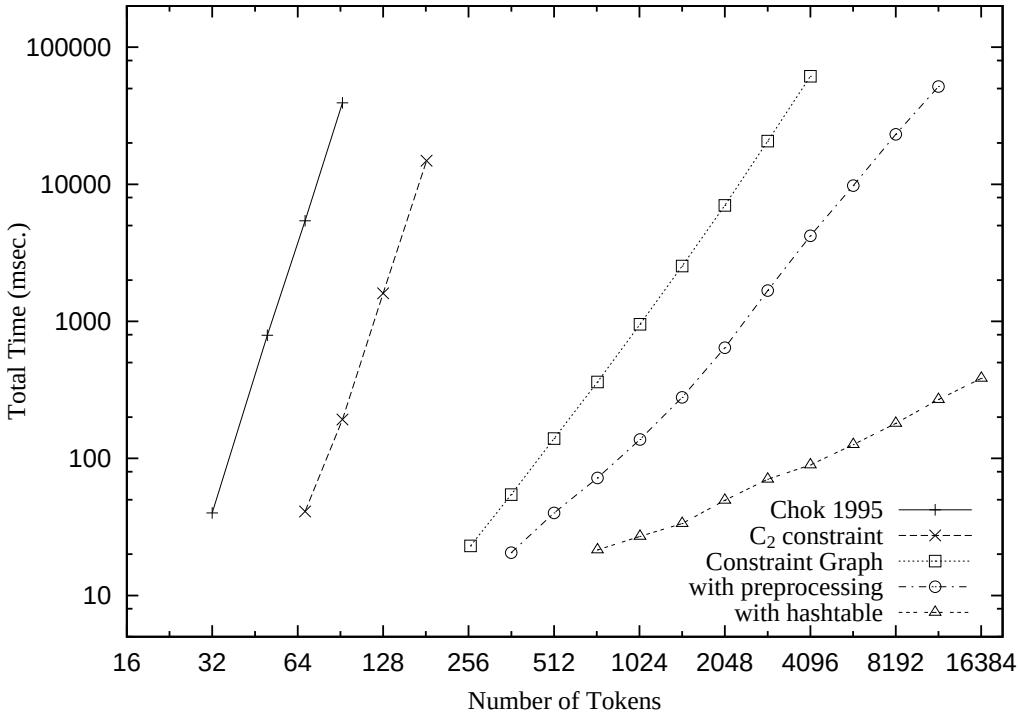

図 4.13: 計算の木の解析における総解析時間（HotSpot 有効）

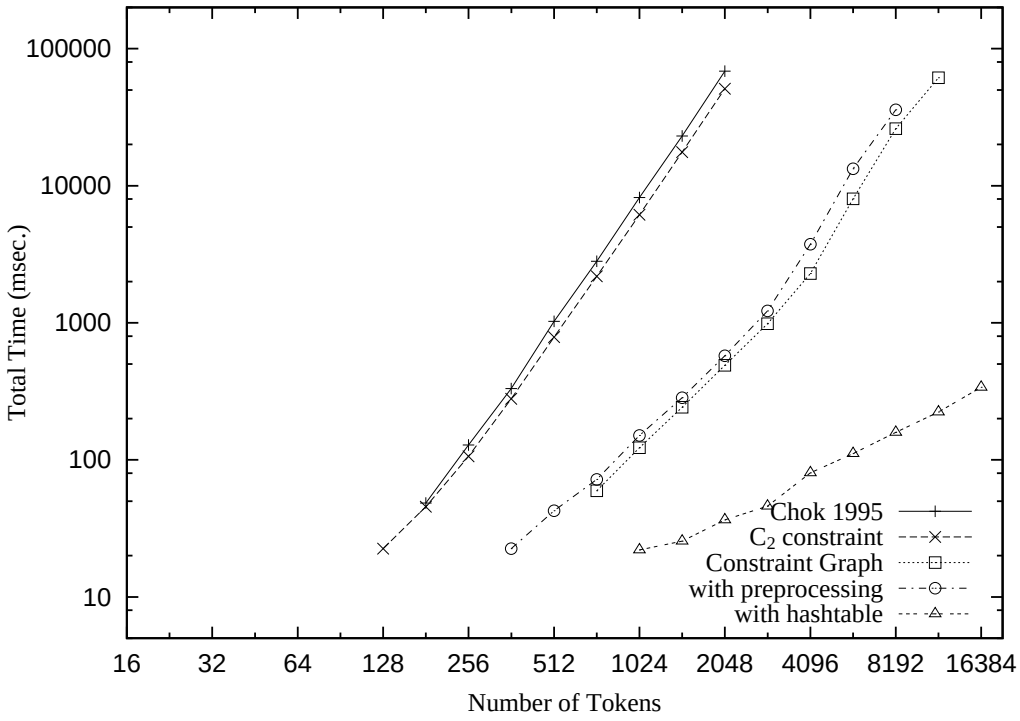

図 4.14: 折れ線の解析における総解析時間（HotSpot 有効）

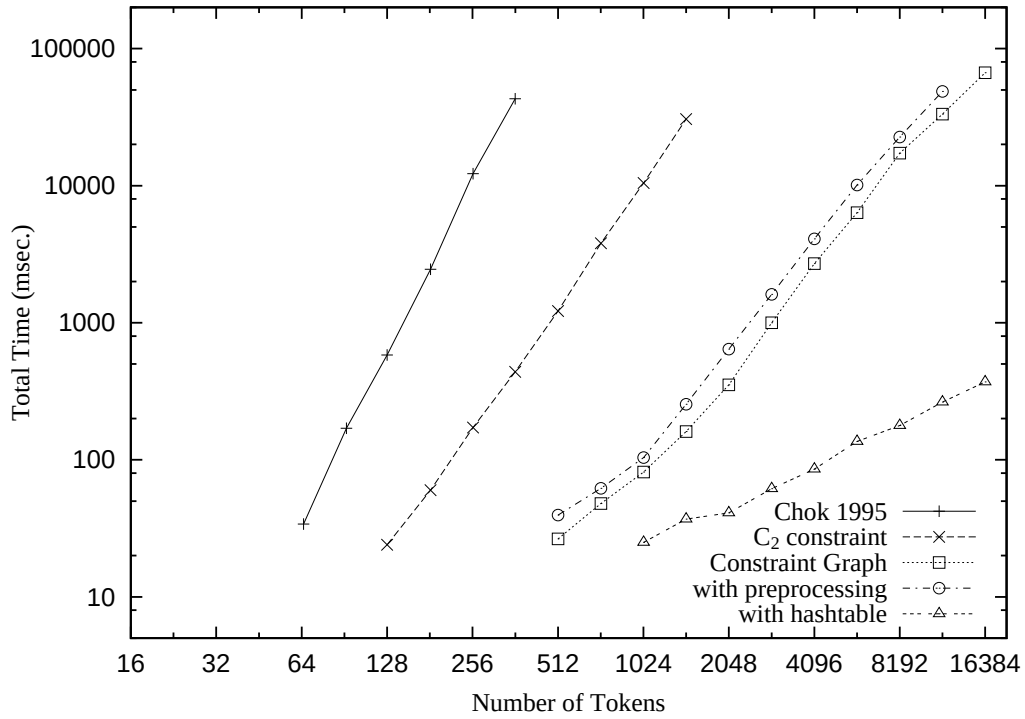

図 4.15: リスト構造の解析における総解析時間（HotSpot 有効）

4.9　exist の記号と not-exist の記号の扱い

これまでに説明した高速化手法は、Chok らのアルゴリズムを基にしたものである。これでは、exist の記号と not-exist の記号については扱っていなかった。よって、ここで、exist の記号と not-exist の記号を含めた解析手法について述べる。

まず、exist の記号と not-exist の記号を扱うための準備として、空間解析器が扱っているすべてのトークンを含む**全トークン集合** E を導入する。これには、トークンデータベースにある場合だけでなく、そのトークンがすでにルールによって還元され、他の非終端記号の RHS の記号として利用されているトークンについても含まれる。このような全トークン集合 E に含まれるトークンは、exist の記号と not-exist の記号が対象とするトークンである。

exist の記号を含むルールに関する解析は、これまでの制約条件グラフを利用した解析から容易に拡張できる。その記号が存在するという条件に関しては、RHS の記号と同じであるため、exist の記号についても制約条件グラフのノードに対応させて解析を行えばよい。ただし、exist の記号に対応するノードの場合、全トークン集合 E に含まれるトークンを探索の対象とする。RHS の記号の場合は、トークンデータベース D を対象としていたので、唯一この部分だけが探索部分で異なる。このようにして制約条件グラフを探索し、制約条件を満たすトークンの組合せが見つかった場合は、RHS の記号のみをトークンデータベースから削除すればよい。また、開始記号として exist の記号から始まった場合は、その記号が還元されずにトークンデータベースに残っていることになる。よって、このような場

合は、解析を継続する必要がある。なお、LHS の記号に対応する非終端記号には、ルール適用時に利用した exist の記号のトークンの情報を保持する。以上の処理によって、exist の記号を含むルールを扱える。

not-exist の記号を含むルールに関する解析も、制約条件グラフを利用した解析を拡張することで対応できる。not-exist の記号についても制約条件グラフにおけるノードに対応させる。ただし、ほかのノードと区別するため、**ネガティブノード**と呼ぶまた、not-exist の記号に関する制約条件、つまりネガティブ制約条件に関しても同様にエッジに対応させる。このネガティブ制約条件に対応したエッジを**ネガティブエッジ**と呼ぶ。なお、ネガティブノードに接続するすべてのエッジはネガティブエッジとなる。

このような拡張された制約条件グラフに対しての探索を行う。ただし、該当するトークンが “存在しないこと” を確かめるため、ネガティブノードに対する訪問は特別な扱いをする。ネガティブノードに対する訪問は、そのノードに接続するエッジ、つまりネガティブエッジにつながるすべてのノードが訪問された後に訪れる。これは、not-exist の記号がすべてのネガティブ制約条件を満たすという条件に対応している。not-exist の記号に対応するあるトークンが、いくつかのネガティブ制約条件が満たしたとしても、それは、現在求められている RHS の記号を否定する十分な条件にはならないからである。よって、すべてのネガティブ制約条件がチェックできるようになるまで、ネガティブノードは訪問されない。このようにして、ネガティブノードに訪問した場合、ネガティブノードに対応するトークンとして、全トークン集合 E の中からネガティブ制約条件をすべて満たすトークンを探す。ここで、1 つでもそのようなトークンが見つかった場合、探索は失敗となる。見つからなかった場合は、not-exist の記号に対応するトークンが存在しないことになるので、このネガティブノードに関する訪問は成功となる。

4.10　トークンの追加と削除

ここでは、インタラクティブシステムにおいてユーザが図形を編集した際に発生するトークンの追加と削除に関して検討する。

まず、トークンの追加であるが、これは、ユーザがキャンバスから図形を描画した場合に生じる。この場合、図形に対応する新たな終端記号を追加する。このときの処理は以下のようになる。

```
routine InsertTokens(T)
  D := D ∪ T
  E := E ∪ T
  U := U ∪ T
  CheckNegativeConstraint()
  Parse()
end
```

InsertTokens() は、追加したいトークン列を引数にして呼び出される。この際、トークンデータベース D と未処理トークン集合 U に加え、全トークン集合 E に新しいトークン

列を追加する。この後、CheckNegativeConstraint() が呼び出されている。これは、新たなトークンが追加されたことによる、ネガティブ制約条件のチェックである。この追加されたトークンが not-exist の記号に該当する場合で、ネガティブ制約条件を満たすような非終端記号があれは、この非終端記号はルールが適用できなくなり削除される。このチェックが終わった後、インクリメンタルな解析を実行している。

インタラクティブシステムでは、図形が追加されるだけでなく削除される場合もある。この場合は、削除する図形に対応する終端記号のトークンを削除する。このときの処理は以下のようになる。

```
routine DeleteToken(T)
  E := E\{T}
  Y := {T}
  if T ∈ D then
    D := D\{T}
    L := L\{T}
    U := U\{T}
    ParseByDelete(Y)
  else
    X := {}
    P := Parent(T)
    do
      X := X ∪ (Children(P)\{T})
      T := P
      E := E\{T}
      Y := Y ∪{T}
      P := Parent(T)
    while P != nil
    D := D\{T}
    L := L\{T}
    U := U\{T}
    ParseByDelete(Y)
    InsertTokens(X)
  end
end
```

DeleteToken() は、削除したいトークンを引数にして呼び出される。まず、このトークンは全トークン集合 E から削除される。削除するトークンがトークンデータベース D に存在する場合は、トークンデータベース D から削除する。同様に処理済トークン集合 L、または、未処理トークン集合 U に削除するトークンが含まれていれば削除される。なお、トークンが削除された場合でも、トークンデータベースに変化があったため解析を行う必要がある。ただし、この際に解析すべき必要があるのは、削除されたトークンが not-exist の記号に該当する場合のみである。よって、この解析は普通のインクリメンタルの解析とは異

なる解析を行う必要がある。これは ParseByDelete() で行っている。

一方、削除するトークンが、トークンデータベース D に含まれていない場合は、そのトークンは何らかの非終端記号の RHS の記号として利用されていることになる。この非終端記号は RHS の記号のトークンを失うため存在できなくなるので、この非終端記号も削除する必要がある。このとき他の RHS の記号に対応するトークンは、トークンデータベースに戻される必要がある。これらは後でまとめて処理するため、変数 X に設定される。この非終端記号がさらに他の非終端記号に利用されている場合は、この処理が再帰的に行われる。最終的には、DeleteToken() の引数で渡されたトークンから解析木を上に向かって探索し、最上位のトークン 1 つのみをトークンデータベースから削除する。なお、全トークン集合 E から削除されたトークンに対しては、ParseByDelete() で必要な解析を行われる。このようにして、削除するトークンに関連するトークンを処理した後、変数 X に保存していたトークン列をトークンデータベースに追加するため InsertTokens() を呼び出している。

トークンが削除され、そのトークンが存在しなくなったことにより適用できる可能性のあるルールは、そのトークンが not-exist の記号として該当し、ネガティブ制約条件に当てはまる場合である。つまり、制約条件を満たす RHS の各記号に対応するトークンが存在し、さらに、この削除するトークンがネガティブ制約条件をすべて満たす場合である。このときの解析は、制約条件グラフを用いた解析をさらに拡張することで実現できる。このトークンが対応する not-exist の記号に関するノードを新たに追加し、さらに、この記号に関するネガティブ制約条件を一般の制約条件と同様にエッジに対応させる。このようにして得られた制約条件グラフに対して、追加されたノードに開始トークンとして削除されたトークンを与えることで、目的の組合せを求めることができる。このとき探索が成功した場合は、ルールを適用することができる。なお、この削除するトークンによって、ルールが適用できるトークンの組合せはほかにも存在する場合があるため、探索が失敗するまでこの処理を繰り返せばよい。ParseByDelete() では、このような処理を行うことで実現される。

4.11 Chok と Marriott の新しい解析手法

Chok と Marriott は、2003 年に文献 [Cho03] にて 4.1.1 節で説明した手法を改良し、高速な解析手法を提案している。

高速化手法のポイントは、以下の 2 点である。

- トークンを *new* と *old* の 2 つに分類して解析を行う。
- 制約条件を利用し、誤ったトークンをなるべく速い段階で発見する。

第 1 の手法は、トークンを 2 つに分類することで、インクリメンタルな解析の際のトークンの組合せの範囲を限定させている。new に含まれるトークンは、新たに追加されるトークンであり、これまで解析されたトークンは old に含まれる。インクリメンタルに解析する場合、前回の解析の結果 old に含まれているトークンのみではルールが適用できないので、解析範囲を new の中のトークンを含む組合せに限定できる。Chok らは、この手法は、

演繹データベースにおける semi-naive な不動点アルゴリズムの変形であると述べている。この手法に基づくルール選択アルゴリズムとして、以下を示している。

```
procedure IncParse(T)
  for each t in T do
    AddNode(t,[ ],[ ],NULL)
  end for
  for each SCC in order do
    repeat
      for each production R in the SCC do
        call EvalProduction_R
      end for
    until no production can be applied
  end for
  for each symbol type s do
    OldRooted[s] := NewRooted[s]::OldRooted[s]
    OldReduced[s] := NewReduced[s]::OldReduced[s]
    NewRooted[s] := nil
    NewReduced[s] := nil
  end for
end
```

ここで、OldRooted と NewRooted はトークンデータベースに含まれるトークンの集合であり、OldReduced と NewReduced はすでに還元されたトークンの集合である。また、OldRooted と OldReduced はこれまでに解析が行なわれたトークンの集合であり、NewRooted と NewReduced は新たに追加されるトークンの集合である。

第 2 の手法は、対応するトークンを求める際に、順次制約条件を利用することで、探索範囲を減少させている。この際の、RHS の記号を利用する順序と、各段階でチェックすべき制約条件を求めるアルゴリズムとして、以下を示している。

```
let v_0 be the initial variable
let V be the set of remaining variables
let C be the set of constraints
seq := [assign(v_0)]
known := {v_0}
while C ≠ ∅ do
  choose c ∈ C s.t. c minimizes |vars(c) \ known|
  let {v_1, ..., v_m} = vars(c) \ known
  seq := seq::[assign(v_1), ..., assign(v_m), test(c)]
  known := known ∪ {v_1, ..., v_m}
  V := V \ {v_1, ..., v_m}
  C := C \ {c}
endwhile
let {v_1, ..., v_n} = V
seq := seq::[assign(v_1), ..., assign(v_n)]
```

ここで、$assign(v)$ は、変数 v のトークンを 1 つ選ぶことを意味する。また、$test(c)$ は、制約条件 c をチェックすることを意味する。このような、$assign()$ と $test()$ の順序を求めることにより、効率のよい探索が行える。

たとえば、状態遷移図の例の終了状態を定義するルールでは、1 番目の RHS の記号 Circle から始める場合は次のようになる。なお、ここで、c1, c2, t は各 RHS の記号に対応する変数である。

assign(c1), *assign*(c2), *test*(c1.mid == c2.mid),
test(c1.radius <= c2.radius), *assign*(t), *test*(c1.mid == t.mid).

このようにして得られた順序に基づき、システムは以下のようなルール適用のプログラムを出力する。

for each c_1 in NewRooted[*circle*] **do**
  c_1.lock := *true*
  $\theta := \{\mathtt{c1} \mapsto c_1\}$
  reduced := *false*
  for each c_2 in NewRooted[*circle*] or OldRooted[*circle*] s.t. c_2.lock = *false* **do**
    c_2.lock := *true*
    $\theta := \theta \cup \{\mathtt{c2} \mapsto c_2\}$
    if $\text{test}_{\text{c1.mid}==\text{c2.mid}}(\theta)$ and $\text{test}_{\text{c1.radius}<=\text{c2.radius}}(\theta)$ **then**
      for each *t* in NewRooted[*text*] or OldRooted[*text*] s.t. *t*.lock = *false* **do**
        t.lock := *true*
        $\theta := \theta \cup \{\mathtt{t} \mapsto t\}$
        if $\text{test}_{\text{c1.mid}==\text{t.mid}}(\theta)$ **then**
          s := create_{state}(c_1.mid, c_2.radius, *t*.label, ”final”)
          AddNode(*s*, [c_1, c_2, *t*], [ ], FS)
          move c_1, c_2, *t* from NewRooted to NewReduced and
          from OldRooted to OldReduced as appropriate
          reduced := *true*;
        endif
        t.lock := *false*
        if reduced = *true* **then break endif** % exit for loop
      endfor
    endif
    c_2.lock := *false*
    if reduced = *true* **then break endif** % exit for loop
  endfor
  c_1.lock := *false*
endfor

まず c1 と c2 にトークンが設定される。その後、c1 と c2 に関する制約条件“c1.mid == c2.mid”と“c1.radius <= c2.radius”がチェックされ、c1 と c2 の組合せが正しい組合せであることが決定される。もし、制約条件を満たさなかった場合は、他のトークンを設定し、制約条件をチェックし直す。これにより、t の変数を求める前に、不要な組合せを除去することが可能となっている。

なお、第 1 の手法は、4.4 節で示した未処理トークン集合を利用した解析とほぼ同一のものである。ただし、Chok らの手法で用いられているルール選択手法では、同じ未処理トークンに対して同じルールを重複してチェックする場合があるが、本研究で提案した手法では、このようなケースは発生しないアルゴリズムを示している。また、第 2 の手法は、

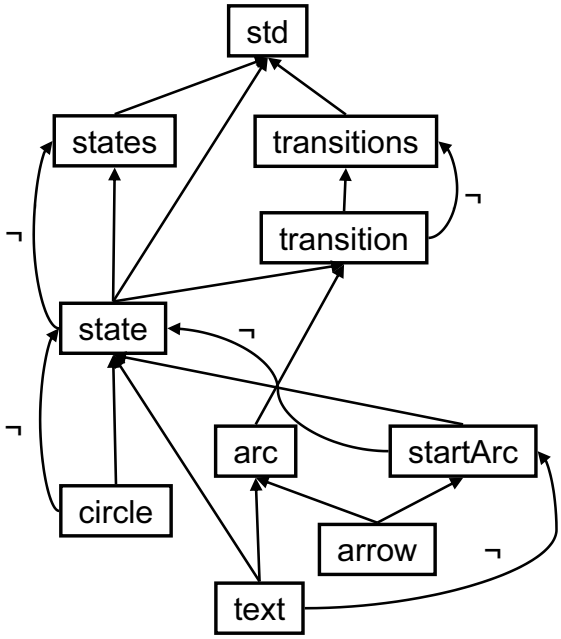

図 4.16: 状態遷移図に関する記号間の依存関係

4.5 節で示した制約条件グラフを利用した探索と同一のものである。これらの高速化手法は、それぞれ、同時期に独立して開発された高速化手法ではあるが、結果として同じアルゴリズムが提案されている。

本研究で提案した、制約条件グラフを利用した探索手法では、グラフを用いて制約条件と記号の関係を示すことにより、記号と制約条件の関係を明示し、探索方法を理解しやすい。また、特に C_2 の制約条件がトークンとトークンの関係を示すことを示唆している。これにより、前処理を加えた探索や属性値テーブルの利用といったさらなる高速化手法へ発展させることができた。

Chok と Marriott は同じ文献において、状態遷移図に関する記号間の依存関係（図 4.16）を示している。この図で否定の記号 “¬” がついたエッジは not-exist の記号または、all の記号として利用されていることを示している。Chok と Marriott は、このグラフから次の強連結成分（SCC）の列を導き出している。

SCC1: arc
SCC2: startArc
SCC3: [$\text{state}_{(normal)}$, $\text{state}_{(final)}$, $\text{state}_{(start)}$]
SCC4: [$\text{transition}_{(a \to b)}$, $\text{transition}_{(a \to a)}$]
SCC5: states
SCC6: transitions
SCC7: std

このグラフでは記号間の依存関係を示しており、ルールの適用の順序を計算するには適切ではない。正しくは、記号間の関係ではなくルール間の関係を利用すべきである。先の

階層では、3 種類の state が同じ階層に含まれていた。しかし、これらは互いに依存していないので、不必要なルール適用が発生しうる。正しい依存関係を利用すると、以下のようにさらに階層を分割できるため、適用順序は効率化される。

SCC1: arc
SCC2: startArc
SCC3: state$_{(normal)}$
SCC4: state$_{(final)}$
SCC5: state$_{(start)}$
SCC6: transition$_{(a \to b)}$
SCC7: transition$_{(a \to a)}$
SCC8: states
SCC9: transitions
SCC10: std

4.12 高速化のまとめ

本研究では、以下のような解析の高速化の手法を提案した。

1. ルール間の依存関係の解析（4.2 節）
2. 制約条件を利用した組合せの削減（4.3 節）
3. 未処理トークン集合を利用した解析（4.4 節）
4. 制約条件グラフを利用した解析（4.5 節）
5. 前処理を加えた探索（4.6 節）
6. 属性値テーブルの利用（4.7 節）

ルール間の依存関係の解析については、トポロジカルソートを行うことにより、線形アルゴリズムによる解析が行えることを示した。ただし、ルール間の解析は図形文法が指定された際に行われるため、インタラクティブに図式を解析する際の高速化ではない。制約条件を利用した組合せの削減では、C_2 の制約条件を利用することでインクリメンタルな解析の高速化を図った。しかし、これは一定の効果があったが、計算量のオーダを減少させることはできなかった。未処理トークン集合を利用した解析では、トークンが追加された際の不要な組合せを探索しない。これは、オーダを 1 つ下げる効果があったが、高速化は不十分であった。

そこで、制約条件グラフを利用した解析により、C_2 の制約条件を有効利用し、効率的に探索を行う手法を提案した。これにより、強い制約条件のみを利用した訪問を行った場合、1 トークンあたりの解析時間として $O(n)$ の解析を実現できた。しかし、弱い制約条件が現れる場合は、オーダが増えてしまう問題があった。そこで、前処理を加えた探索を

行うことで、弱い制約条件を極力利用しないグラフの訪問を実現できることを示した。また、属性値テーブルを利用することで、訪問先の対応するトークンを即座に決定することができるようになることを示した。これにより、計算の木などの例では $O(1)$ の解析を実現できた。つまり、インタラクティブなシステムにおけるインクリメンタルな解析が図形数によらず一定時間で行える。また、全体の解析に関しても線形の処理時間で解析が終了する。

これらの制約条件グラフを利用した提案手法では、解析を高速化できないようなルールも存在する。これは、あるノードに到達するためのすべてのエッジが弱い制約条件であった場合や、ある記号がどのような制約条件にも含まれていない場合である。具体的には、次のような例が考えられる。

```
q:Quadruplet ::= a:Node, b:Node, c:Node, d:Node where(
  a.val <= b.val
  b.val <= c.val
  c.val <= d.val
) {}
```

このように、すべてが弱い制約条件の場合は、最悪の計算量が必要となる。この場合、1 つのトークンの追加に $O(n^3)$ の処理が必要になる。

ただし、このような定義がなされることは一般に考えられない。なぜなら、記号 Node が 5 つ以上あった場合に、どの 4 つ組を選ぶか何も条件を指定していない。つまり解釈が曖昧な定義である。そのような定義をあえてする必要があるか疑問である。また、4.1.1 節で述べたように、このような定義は Chok らの手法をベースにしたインクリメンタルな解析では正しく扱うことができない。したがって、何らかの条件、たとえば、隣接する記号間の 4 つ組であるといった制約条件が加えられることになる。こうすることにより、強い制約条件が与えられ、高速に解析が行えるようになる。

弱い制約条件による曖昧性を not-exist の記号を利用して取り除いているようなケースにおいては、実用的で曖昧性のない定義であっても高速化が実現できない場合がある。たとえば、次のような例である。

```
m:PairNode ::= f:Node, s:Node where (
  f.mid_y == s.mid_y
  f.mid_x < s.mid_x
  not exist n:Node where (
    f.mid_y == n.mid_y
    f.mid_x < n.mid_x
    n.mid_x < s.mid_x
  )
) {}
```

この定義は、水平方向に隣り合う記号 Node を 1 つの非終端記号としている。このとき、横にある記号 Node がどれくらい離れているのか分からない場合は、単に左右に並んでい

るとしか記述できない。この場合では、“`f.mid_x < s.mid_x`”という条件で指定されている。しかしこれだけでは間に他の記号 Node が存在しても認識してしまうため、これを避ける必要がある。このために not-exist の記号を利用して、間に入る記号 Node が存在しないという条件を記述している。これにより、曖昧な解釈が発生しない。しかし、先の弱い制約条件“`f.mid_x < s.mid_x`”があるため、このルールに関する解析では、制約条件グラフを利用することによる高速化が実現できない。

Chok らのアルゴリズムや Balt の手法では、RHS の記号数に依存した計算量が必要であった。一方、提案手法の制約条件グラフを利用した解析手法では、RHS の記号数には直接依存しない。これまで述べたように、エッジで利用されている制約条件に依存し、それらがどのように記号と結びついているかによって計算量が変化する。訪問に利用するエッジが、座標の一致のような制約条件の場合は、ハッシュ表を用いて $O(1)$ で対応するトークンが発見できた。また、それ以外の場合でも多くの場合 $O(\log n)$ で対応するトークンを見つけることができる。これにより、すべてが強い制約条件のエッジを利用して訪問した場合、1 トークンに対する解析は、最速で $O(1)$ の計算量で、遅くとも $O(\log n)$ の計算量で解析が可能である。つまり、図式全体の解析には、$O(n)$ から $O(n \log n)$ の計算量で解析できるということである。3.3 節で調査した図形言語は、すべてこの条件に当てはまる。

提案したアルゴリズムの正当性、つまり、このアルゴリズムが図形言語を正しく識別し、解析が終了するための条件は、4.1.1 節で示した Chok らのアルゴリズムにおける条件と同じである。つまり、扱える図形言語の範囲は変わっていない。なぜなら、提案手法では、Chok らのアルゴリズムのインクリメンタルな解析部分に関する高速化を行っただけだからである。

提案手法は、このような条件を満たす図形言語に対して適用できる高速化手法である。特定の図形言語に関する解析器を高速化したのではない。もちろん、図形言語が特定されれば、それに特化した解析アルゴリズムを構築することで高速化が実現できるが、本研究では、より一般的な解析アルゴリズムを提案した。これにより、図形文法を与えるだけで、高速な解析器を利用することが可能となる。

本研究では、図形文法として CMG を対象に解析アルゴリズムの高速化を行った。もちろん、他の図形文法に基づく解析も知られている。たとえば、Wittenburg は Relational Grammars に対して Earley-style[Ear70] の解析を用いることで高速化を行っている [Wit92]。しかし、これは、Relational Grammars が CMG に比べて記述力の弱い文法であるため高速な解析が実現できている。本研究で提案した手法は、より広い範囲の図形言語を扱える CMG において、高速な解析を実現する。

第5章　図形エディタと空間解析器の統合

本章では、空間解析器を利用した図式処理システムにおいて、図式に対するインタラクション機能が不足している問題について議論する。最初に、これまでのシステムにおける解析結果の利用状況についてまとめる。その後、どのような機能が不足しているのかについて述べ、これを解決するための手法として図形エディタと空間解析器の統合を提案する。また、統合を実現するための特別な記号の導入と図形文法の拡張について述べる。最後に、具体的な例でその利用法を示す。

5.1　解析結果の利用

2.5 節で述べたように、インタラクティブシステムにおいて、空間解析器を用いることで、静的な解析では得られなかった、インタラクティブな処理が行える。

Penguins では、図式のレイアウトや整形に解析結果を利用している。Chok らは、文献 [Cho99] において CMG を拡張し、ルール定義に “**layout** $\{\cdots\}$” という部分を追加することで、非終端記号が認識された際のレイアウトに関する情報を記述できるようにした。具体例として、状態遷移図の遷移の定義を拡張した、次のような例を挙げている。

```
t:Transition() ::= a:Arc,
    exist s1:State, s2:State where (
  onCircle(a.start, s1.mid, s1.radius)
  onCircle(a.end, s2.mid, s2.radius)
) {
  t.start = s1.label
  t.tran = a.label
  t.end = s2.label
} layout {
  if (fabs(a.start.y - a.end.y) < 30)
    a.start.y == a.end.y
  if (fabs(a.start.x - a.end.x) < 30)
    a.start.x == a.end.x
  s1.radius == s2.radius
}
```

この定義は、図 3.4 で示した定義に、レイアウト機能の記述が加えられている。これでは、ほぼ水平に描かれた遷移の矢印は、水平な矢印に整形される。垂直方向も同様に処理

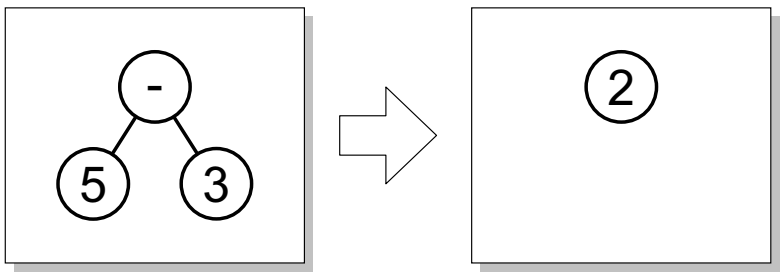

図 5.1: アクションによる図式の書き換え

する。また、遷移の対象の 2 つの状態について、それぞれの円の半径を等しくするようにしている。

これらの処理により、他の記号の属性を変更する場合がある。このため、認識された図式の意味が崩れないように、制約解消系により、そのほかの図式の位置なども変更される。これらのレイアウト機能は、図式を認識する際には関係ないが、認識された後の図式をユーザに見やすく提示することができる。

Rainbow では、レイアウト機能として、硬いレイアウト制約と軟かいレイアウト制約を扱える。特に、軟かいレイアウト制約では、スプリングモデルなどによるレイアウト機能を提供し、制約解消系の機能を補間している。

恵比寿では、制約解消系による図式のレイアウト機能に加えて、アクションの記述ができる。これは、CMG を拡張し、属性定義のあとにアクションの記述を行う。馬場らは、文献 [Bab98a] において計算の木のルール 2 の定義を変更した、次のような例を示している。なお、これは、図 5.1 に示すような図形の書き換えを行う。

```
# ルール 2（アクションによる書き換え）
n:Node ::= c:Circle, t:Text, l1:Line,
     l2:Line, n1:Node, n2:Node where (
  c.mid == t.mid
  isOperand(t.text)
  c.mid == l1.start
  c.mid == l2.start
  l1.end == n1.mid
  l2.end == n2.mid
  n1.midx < n2.midx
) {} {
  t.text = eval(n1.val, t.text, n2.val)
  deleteObject(l1, l2, n1, n2)
}
```

制約条件部分は同じであるが、異なるのはそれ以降の記述である。属性定義は、何も記述されていない。また、その後の部分にアクションが記述されており、この例では 2 つの処理が定められている。最初の行では、2 つの葉が持つ値と節の持つ演算子から計算された

値をテキストの文字列に設定している。これにより、演算子を表していた部分が計算結果に書き換わる。図 5.1 のような“5 – 3”という図式であった場合、テキストの“ – ”が“2”に書き換えられる。次の行では、引数で渡された記号を削除する機能を持つ deleteObject() 関数が用いられている。これにより、2 つの葉の部分にあたる記号と、それを結んでいる 2 つの線を削除している。これにより、木構造をしていた図式が、演算子の部分のテキストと円のみに書き換えられることになる。

この書き換えでは、RHS の記号が削除される。これにより、この LHS の記号は条件を満たさなくなるので削除され、残っている円とテキストをトークンデータベースに戻す処理が行われる。このとき、この円とテキストは中心が一致しており、また数字を持ったテキストであるので、ルール 1 が適用されて記号 Node として認識される。これにより、多段の木になっていた場合でも計算が進んでいくことになる。

5.2　図形エディタと空間解析器の統合

上述のように、既存のシステムでもインタラクティブシステムにおける、空間解析器の解析結果を利用した処理が実現されている。しかし、図式処理システムとして求められる図式に対するインタラクション機能は十分ではなかった。たとえば、特定図形の上にマウスを重ねるとハイライトしてユーザの入力を補助したり、解析結果に基づいて何らかのモードを切り替えて利用できるようにしたり、図形言語特有のメニューやボタンなどの利用したりすることなどが考えられる。これまでのシステムでは、これらの機能を実現するために、その図形言語専用の図形エディタを作成し、図形エディタ側で処理を記述する必要があった。しかし、これは解析結果を空間解析器から取得しながら作業を行う必要があり、煩雑な手続きが発生する。また、図形言語に関する定義と、その処理を別々に記述しなければならず不便であった。

そこで、このような問題を解決するため、図形エディタと空間解析器の統合を提案する [Iiz01a, Iiz01b]。これにより、図式に対するインタラクション機能を提供するのに必要な、マウスなどのイベント情報や GUI コンポーネントなどの情報は図式の解析結果とあわせて空間解析器で統一的に処理が行われる。また、このための処理の記述は、図形言語に対する定義と同一の図形文法の枠組みで実現する。

5.2.1　システムの構成

提案する手法に基づくシステムは、ユーザとのインタラクションを行う入出力部と、図形言語を文法に基づいて処理する空間解析部から成り立つ。入出力部は、図形を描画・編集するためのキャンバスに加え、図形言語ごとにメニューやボタンなどの GUI コンポーネントを付加することができる。空間解析部は、あらかじめ与えられた図形文法をもとに解析を行う。また、システムに制約解消系を組み込むことで、解析結果に応じて図形のレイアウトなどを任意に行うことができる。システムの起動時には、図形文法を読み込むとともにキャンバスを初期化し、必要に応じてメニューやボタンを作成して利用する。

5.2.2 特別な終端記号の導入

マウスやメニューをなどの情報を空間解析器で処理できるようにするため、空間解析器側に特別な終端記号を導入する。これまで図形言語で用いられていた終端記号は、矩形や円、文字といった基本図形要素を表現しているが、本手法では、マウスやメニューについても終端記号として扱えるようする。これは、空間解析器の扱えるデータが記号のインスタンスであるトークンのみであるからである。これらのトークンの属性値がユーザの入力によって変化するようにする。また、逆に、これらの属性値を変更することで、入出力部の GUI コンポーネントを操作できる。つまり、キャンバス上の図形に対応する終端記号と同様に扱える。また、これらの終端記号に対応するトークンは、システムの起動時に生成され、空間解析器に設定される。これにより、マウスの動きや、メニューアクションなどに対する処理を空間解析器が扱えるようになる。また、図式に関する記号と組み合わせて利用することで、解析結果を利用した処理が実現できる。

追加する終端記号は、以下のようなものである。

- Mouse(pos, button1, button2, button3, action, handled)
- Button_*name*(action, handled)
- RadioButton_*name*(current, handled)

マウスを扱う終端記号 Mouse は、マウスカーソルの位置を示す座標の属性 `pos` とマウスボタンが押されているかどうかを示すフラグを属性 `button1`〜`3` として持つ。また、クリックやドラッグなどのアクションイベントを保持するための属性 `action` と、また、その処理を行ったかどうかを示すフラグとして属性 `handled` を持つ。属性 `handled` は、イベントが発生したときに false に設定される。これは、イベントに対する処理が何度も実行されるのを防ぐために用いられる。

ボタンを扱う終端記号 Button_*name* は、ボタンの状態を示す属性 `action` と、ボタンが押された際の処理を行ったかどうかを示すフラグとして属性 `handled` を持つ。この終端記号は、図形言語ごとに必要に応じて利用する。また、各ボタンに対してそれぞれ終端記号が存在し、これを区別するために、終端記号名に名前 *name* が付加される。なお、属性 `action` には、ボタンが押されたときには “pressed” が設定される。

ラジオボタンを扱う終端記号 RadioButton_*name* は、現在選択されているボタンを示す属性 `current` と、状態が変化した際の処理を行ったかどうかを示すフラグとして属性 `handled` を持つ。この終端記号は、図形言語ごとに必要に応じて利用する。また、各ラジオボタンのグループに対してそれぞれ終端記号が存在し、これを区別するために、終端記号名に名前 *name* が付加される。

なお、同種の GUI コンポーネントを上述のように終端記号名によって区別した場合、各終端記号に対応するトークンは 1 つのみ存在することになる。一方 GUI コンポーネントを 1 つの終端記号名でまとめ、属性で区別することもできる。たとえば、複数のボタンを次のように扱うことも可能である。

Button(name, action, handled)

このようにすることの利点は、複数のボタンに対して同一のバインディングを行いたい場合に、その処理をまとめて簡潔に記述することが可能となる点である。一方で個別にバインディングを記述したい場合には、属性の値をそれぞれ指定しなければならず煩雑になる。これは、場合によって使い分けることも可能であるが、本論文では、終端記号名で区別する手法で説明を行う。

また、上記のほかにも、メニューやチェックボタンなどの GUI コンポーネントが考えられるが、同様に終端記号を対応させることによって、空間解析器で扱えるようになる。

これらの終端記号に加え、場合に応じて、システムの内部状態を保持するような終端記号を導入する。たとえば、図形のコピーとペーストを行いたいような場合に、コピーされた内容をこの終端記号に設定する。そして、ペースト処理をする場合には、この情報を利用して新たな図形を生成することができる。また、内部の処理モードなどを保持する際にも利用することができる。このような終端記号は、システム生成時に 1 つだけ生成して利用することが考えられる。また、動的にこのような終端記号を生成することも考えられ、この記号が存在する場合に処理を変更するといった記述もできる。

5.3　図形文法の拡張

空間解析器と図形エディタを統合することにより、マウスなどの情報を空間解析器で扱えるようになった。この際の、マウスなどに対する処理の記述は、図形文法で行う。つまり、矩形や線などの図形とともに、マウスやボタンなどを “図形” として扱い、これらを含めた 1 つの図形言語を図形文法で記述する。

インタラクティブシステムの処理の記述には、従来の CMG では対応できない点がある。従来の CMG では、属性定義しか行えない。これでは、複雑な計算など行ったり、ある処理を繰り返して実行したり、システム外部に対して何らかの処理を行ったりすることは困難である。そこで、何らかの “スクリプト” を記述できるようにすればよい。

このようなものとして、恵比寿ではアクションを実現している。アクションは、ルールが適用され RHS の記号が LHS の記号に還元された場合に実行されるスクリプトとして定義される。これにより、図形の生成や削除、書き換えなどを実現するほかに、解析結果を利用して、データをファイルに書き出したり、外部のプログラムを起動したりすることができる。

しかし、インタラクティブシステムにおいては、図形に対応する終端記号が削除されるため、この削除された終端記号を RHS の記号として利用していた非終端記号は存在できなくなる。また、終端記号が削除される場合以外に図形の属性が変更された場合でも、制約条件を満たさなくなった場合は同様に非終端記号が存在できなくなる。このような場合、アクションが実行された際の変更を元に戻す処理が求められるケースが考えられる。しかし、アクションではこのような処理が記述できない。

この問題を解決するため、以下に示すような CMG を拡張したルール記述を用いる。なお、この拡張した CMG のことを **i-CMG** と呼ぶことにする。

T ::= T_1, T_2,$\cdots$, T_k **where** (
　Constraints
) {
　AttributeAssignments
} {
　ConstructScript
} {
　FinalizeScript
}

このルールは、RHS の記号 T_1, T_2,$\cdots$, T_n が制約条件 *Constraints* を満たしたときに LHS の T に還元され、その属性値は *AttributeAssignments* を基に計算されることを示している。i-CMG が CMG と異なるのは、*ConstructScript* と *FinalizeScript* の部分である。*ConstructScript* は、このルールが適用されたときに実行される処理を記述する。また、*FinalizeScript* は、このルールが適用されて生成された LHS の非終端記号が、何らかの理由で削除された場合に実行される処理を記述する。なお、これらスクリプト処理を必要しない場合は、“{ *ConstructScript* } { *FinalizeScript* }” の双方を省略するほかに、“{ *FinalizeScript* }” のみも省略できるものとする。これにより、i-CMG と従来の CMG の記述を混在させた定義が可能である。

AttributeAssignments で定義された属性は、RHS の記号の属性が変化した際に、自動的に LHS の記号の属性も変更される。この逆で、非終端記号の属性に値を設定した場合は、対応する RHS の記号が変更される。このような計算は空間解析器が行う。一方、*ConstructScript* や *FinalizeScript* で属性を設定した場合、それは、非終端記号の生成時と削除時にそれぞれ 1 度だけ実行される。これにより、一時的に属性を変更したい場合とそうでない場合を区別して記述できる。また、RHS の記号の属性間に関する関係を記述したい場合は、制約解消系を用いて、属性間に制約を設定することで実現できる。

なお、非終端記号の削除は、RHS の記号が削除されるか、属性が変更されて *Constraints* を満たさなくなった場合に行われる。exist の記号が利用されていた場合も同様である。また、not-exist の記号が利用されていた場合は、ネガティブ制約条件を満たす not-exist の記号が存在した場合にもこのルールが適用できなくなり、非終端記号が削除される。

たとえば、円とテキストからなる非終端記号 Node を表すルールは次のようになる。なお、ルールが適用されると、円の内部の色が “gray” に変更されるものとする。

```
1: n:Node ::= c:Circle, t:Text where (
2:   c.mid == t.mid
3: ) {
4:   n.text = t.text
5:   n.mid = t.mid
6: } {
7:   n.prev_color = c.inner_color
8:   c.inner_color = "gray"
```

```
 9: } {
10:   c.inner_color = n.prev_color
11: }
```

1 行目は、RHS の記号 Circle と Text が LHS の記号 Node に還元されることを示している。2 行目でこのルールが適用される制約条件を示しており、ここでは円とテキストの中心が一致していることを表している。4–5 行目では、非終端記号 Node の属性を決定している。7–8 行目は、*ConstructScript* を示しており、円の内部の色を変更している。10 行目は、*FinalizeScript* を示している。たとえば、円の中心とテキストの中心が一致しなくなり、非終端記号 Node が削除された場合に、円の内部の色を元に戻すようになっている。このような記述を行うことにより、単に非終端記号として認識するだけでなく、認識したということをユーザにフィードバックができる。また、記号 Node として認識されなくなった場合にも、円の色を元に戻すことによりフィードバックを実現している。

このように、“認識されなくなった” 場合の処理が記述できるのが特徴であり、これまでのアクションではこのような記述は実現できなかった。マウスなど、属性がたびたび変化するような記号を利用した場合に特に有効であり、ある一定の状況下で特定の状態を維持するといった記述が容易に実現できる。

5.4　応用例：状態遷移図における遷移のシミュレーション

提案した手法を用いて、3.2.2 節で示した状態遷移図の例をもとに、状態の遷移をマウスクリックでシミュレートする機能を実現する。

まず準備として、状態を表す非終端記号 State の定義を図 5.2 のように変更する。

各非終端記号 State の定義では、`color` という属性を持つように変更されている。この属性は、状態を表す円の内部の色を表しており、この色によって状態の種類を区別する。現在の状態は赤色で示し、そのほかの状態は白色であるとする。なお、記号 State が終了状態の場合は、二重円の内側の円とする。この属性により、記号 State の属性 `color` で現在の色を参照できるほか、この属性に値を設定することで、円の色を間接的に変更することが可能となる。

各定義における *ConstructScript* として、“`s.color = "white"`” が定義されている。これにより、ルールが適用され状態として認識された場合、円の内部の色として白色を設定している。なお、*FinalizeScript* については省略されており、これらの非終端記号が削除された場合に特別な処理は行われない。

状態の遷移を表す非終端記号 Transition の定義もあわせて変更する。これは、図 5.3 のようになる。

各定義においては、属性定義のみ変更されている。属性 `start_color` は遷移元の状態の色を保持しており、属性 `end_color` は遷移後の状態の色を保持している。また、属性 `mid` は、遷移を表す図形の記号 Arc の属性からテキストの位置を継承している。なお、これらの定義において、スクリプト処理が必要ないので、*ConstructScript* と *FinalizeScript* は省略されている。

```
# State(final)
s:State ::= c1:Circle, c2:Circle,
    t:Text where (
  c1.mid == c2.mid
  c1.mid == t.mid
  c1.radius <= c2.radius
) {
  s.mid = c1.mid
  s.radius = c2.radius
  s.label = t.label
  s.kind = "final"
  s.color = c1.innner_color
} {
  s.color = "white"
}

# State(start)
s:State ::= c:Circle, t:Text,
    a:StartArc where (
  t.mid == c.mid
  onCircle(a.end, c.mid, c.radius)
  not exist m:Circle where (
    m.mid == c.mid
  )
) {
  s.mid = c.mid
  s.radius = c.radius
  s.label = t.label
  s.kind = "start"
  s.color = c.innner_color
} {
  s.color = "white"
}

# State(normal)
s:State ::= c:Circle, t:Text where (
  not exist m:Circle where (
    m.mid == c.mid
  )
  not exist a:StartArc where (
    onCircle(a.end, c.mid, c.radius)
  )
  t.mid == c.mid
) {
  s.mid = c.mid
  s.radius = c.radius
  s.label = t.label
  s.kind = "normal"
  s.color = c.innner_color
} {
  s.color = "white"
}
```

図 5.2: 遷移のシミュレーションの定義（状態を表す非終端記号）

```
# Transition(A->B)
t:Transition ::= a:Arc,
    exist s1:State, s2:State where (
  onCircle(a.start, s1.mid, s1.radius)
  onCircle(a.end, s2.mid, s2.radius)
) {
  t.start = s1.label
  t.start_color = s1.color
  t.tran = a.label
  t.end = s2.label
  t.end_color = s2.color
  t.mid = r.mid
}
```

```
# Transition(A->A)
t:Transition ::= a:Arc,
    exist s:State where (
  onCircle(a.start,s.mid,s.radius)
  onCircle(a.end,s.mid,s.radius)
) {
  t.start = s.label
  t.start_color = s.color
  t.tran = a.label
  t.end = s.label
  t.end_color = s.color
  t.mid = r.mid
}
```

図 5.3: 遷移のシミュレーションの定義（状態の遷移を表す非終端記号）

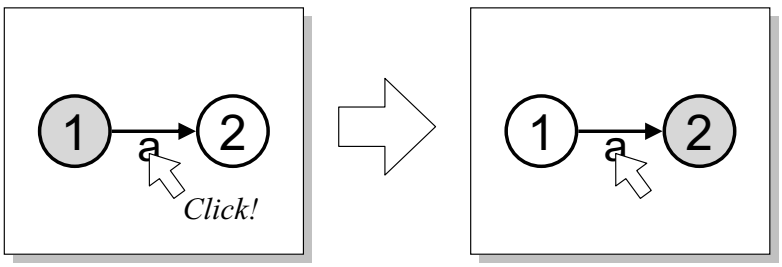

図 5.4: クリックによる状態の遷移のシミュレート

以上のような準備をすることで、マウスクリックで状態の遷移をシミュレートするための処理が記述できる。この定義は、以下のようになる。なお、この定義による動作は、図 5.4 のようなものである。

```
n:StateTransition ::= m:Mouse, exist t:Transition where (
  m.action == "button1-press"
  m.handled == false
  t.start_color == "red"
  m.pos == t.mid
) {} {
  m.handled = true
  t.start_color == "white"
  t.end_color == "red"
}
```

この定義では、RHS の記号として記号 Mouse が、exist の記号として記号 Transition が用いられている。記号 Transition を exist の記号としているのは、記号 Transition が非終端記号 Transitions の all の記号で集められ、トークンデータベースにないためである。制約条件としては、4 つが定義されている。まず、記号 Mouse に対する制約条件として、“`m.action == "button1-press"`”と“`m.handled == false`”が指定されている。これは、マウスのボタン 1 がクリックされ、そのイベントがまだバインドされていないという条件を指示している。また、記号 Transition に対する制約条件として、“`t.start_color == "red"`”が指定されている。これは、その状態遷移の遷移元の状態が、現在の状態を表しているということを指定している。さらに、この 2 つの記号間の制約条件として、“`m.pos == t.mid`”が指定されており、これにより、マウスがクリックされた位置にその状態の遷移が存在するという条件を指定している。

このような条件の下で、*ConstructScript* に記述された処理が実行される。ここでは、まず、“`m.handled = true`”という指定により、このクリックイベントがバインディングされたことを記号 Mouse に設定する。もし、このような処理をしなかった場合、記号 Transition が同じ状態への遷移であったケースでは、このルールが何度も適用可能になりうる。属性 `handled` を利用することで、1 回だけこの処理が実行されることを保証できる。“`t.start_color == "white"`”と“`t.end_color == "red"`”では、この遷移により、遷移元の状態の色を

白色にし、遷移後の状態を赤色に変化させている。このような定義によって、状態の遷移をシミュレーションすることができる。なお、*ConstructScript* において記号 Mouse の属性 handled が変更されるため、RHS の記号 Mouse がこのルールの制約条件を満たさなくなり、このルールによって生成された LHS の記号は削除されることになる。

この例では、マウスによる図式に対する処理を記述した。Penguins や恵比寿などの既存のシステムでは、このような処理を記述するためには、図形エディタに個別に変更を加える必要があった。しかし、図形エディタと空間解析器を統合したことにより、マウスの情報を空間解析器で扱えるようになったため、上記のように簡単に処理を記述できるようになった。また、終端記号としてマウスの情報を扱えるため、図形文法の枠組みを用いて、図式に関する他の記号と組み合わせて利用することができる。これにより、どの状態や遷移が対応するのかといった解析結果を簡単に利用することができた。

5.5　応用例：マウスオーバーによる強調表示

インタラクティブシステムでは、あるオブジェクトにマウスカーソルを重ねると、そのオブジェクトが強調表示され、操作対象を明示することがよく行われる。このような機能を提案手法で実現する。

ここでは、オブジェクトとして矩形とテキストから構成されるもの利用する。この定義は、以下のようになる。

```
o: HighlightableObject ::= r:Rectangle, t:Text where (
  r.mid == t.mid
) {
  o.color = r.inner_color
  o.bb = r.bb
} {
  o.color = "gray"
}
```

この定義では、矩形とテキストの中心が一致していた場合、記号 HighlightableObject として認識される。属性定義では、このオブジェクトの色として属性 color を定義しており、記号 Rectangle の属性 inner_color を継承している。これにより、記号 HighlightableObject の属性 color の値を変更することで、矩形の色を変化させ、このオブジェクトの色を変えることができる。また、属性 bb ではこのオブジェクトのバウンディングボックスとして、矩形のバウンディングボックスを継承している。*ConstructScript* では、このオブジェクトの初期状態の色として“gray”を指定し灰色にしている。

このオブジェクトに対して、マウスカーソルが重なった場合の処理は、次のように記述される。なお、この定義による動作は、図 5.5 のようなものである。

```
h:ObjectHighlight ::= m:Mouse, o:HighlightableObject where (
  contains(m.pos, o.bb)
```

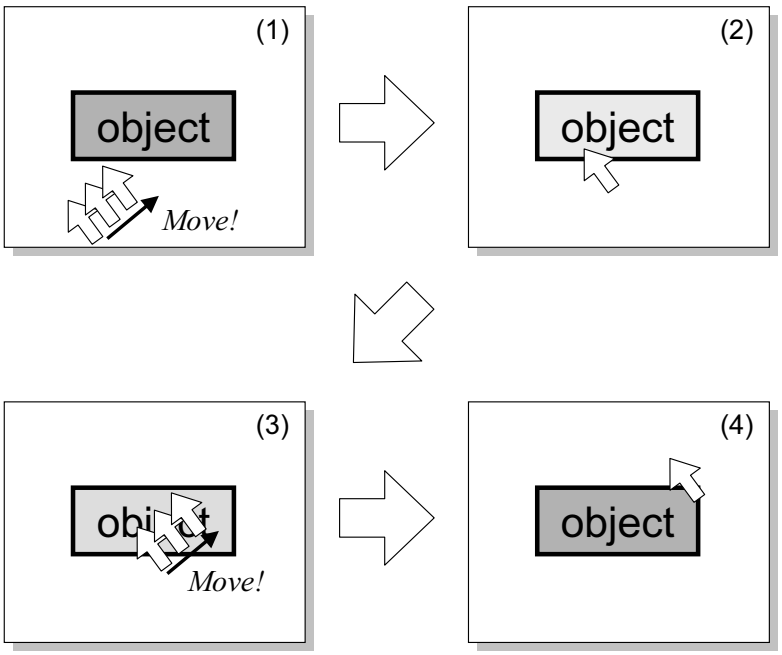

図 5.5: マウスオーバーによる強調表示

```
) {} {
  o.color = "lightgray"
} {
  o.color = "gray"
}
```

このルールは、オブジェクトの上にマウスカーソルがある場合のみ存在する非終端記号 ObjectHighlight を定義している。オブジェクトの上にマウスカーソルがあるという条件は、“`contains(m.pos, o.bb)`”という制約条件で指示している。マウスカーソルがオブジェクトに重なった時点でこのルールが適用され、*ConstructScript* に記述されている“`o.color = "lightgray"`”によってオブジェクトの色が明るい灰色に設定されハイライトされる。このとき生成される非終端記号は、制約条件によりマウスカーソルがオブジェクトの上に置かれている間のみ存在することができる。よって、マウスがオブジェクトから離れた場合は、この非終端記号は削除される。その際に、*FinalizeScript* に記述されている“`o.color = "gray"`”によって、オブジェクトのハイライトが解除される。

このように、ある条件を満たしている間だけに何かを適用した場合や、内部状態を変更させたい場合などに、*ConstructScript* と *FinalizeScript* を利用して、簡潔に処理を記述できるようになる。特に、*FinalizeScript* を用いることで、恵比寿のアクションでは実現できなかった後処理などが記述可能となった。

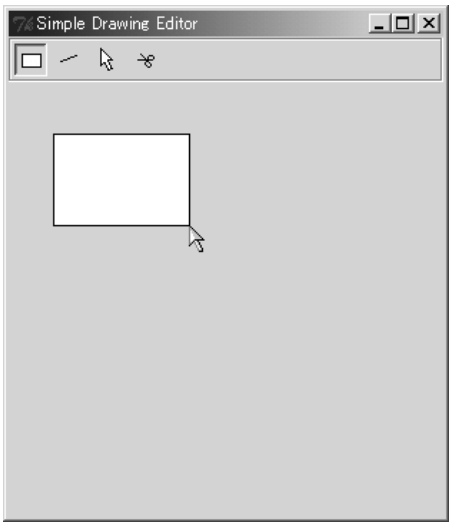

図 5.6: 図形編集機能（図形の描画）

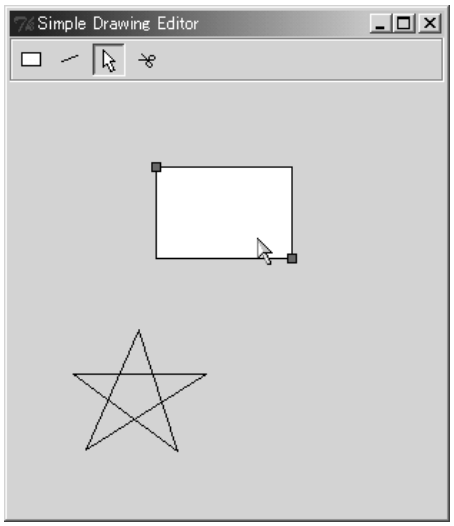

図 5.7: 図形編集機能（図形の選択）

5.6　応用例：図形編集機能の記述

提案手法を用いて、基本的な図形編集機能を記述する例を示す。ここでは、図形の描画、選択、削除について述べる。このような編集機能を空間解析器で処理できるため、図形言語によって、その編集方法を変更したり、図式の状態などに応じて、その機能を変更したりすることが可能となる。

これによって定義されるシステムのイメージは図 5.6 や図 5.7 のようなものである。ラジオボタンで矩形や線などの描画モードを選択できるようになっている。選択モードでは、キャンバス上の矩形や線を選択することができる。また、選択された図形は削除ボタンで削除される。なお、画面上に存在するラジオボタンやボタン、キャンバスは、システムの起動時に作成され、これらのコンポーネントとマウスに対応するトークンは空間解析器に設定される。

5.6.1　図形の描画

図形描画のモードで、キャンバス上をドラッグすると、ラジオボタンで指定された図形を描画するといった処理の記述を示す。

```
d:ObjectCreate ::= m:Mouse, c:RadioButton_Mode where (
  c.current != "select"
  m.button1 == "pressed"
) {} {
  o = createObject(c.current)
  o.start = m.pos
  d.ct = addConstraint("m.pos == o.end")
```

```
} {
  removeConstraint(d.ct)
}
```

ここで、記号 RadioButton_Mode は描画モードを表す終端記号である。ルール中の createObject() は、引数で指定された図形をキャンバス上に追加するとともに、新たに終端記号 Object を生成する関数であり、新しい終端記号を示す識別子を返す。生成された終端記号 Object はキャンバスの図形と対応しており、その属性が変更されると、キャンバス上の図形も変更されるようにバインディングされている。addConstraint() は属性間に制約を課す関数であり、引数で指定した制約を設定する。また、この関数は、その制約を示す識別子を返す。課せられた制約は removeConstraint() で外される。指定された制約は、制約解消系に設定され、ある属性値が変更された場合に、指定された制約を満たすように属性値が変更される。

このルールは、マウスのボタンが押されたときに、図形を作成するためのものである。なお、矩形や線のどちらの描画モードでもこの 1 つのルールで対応する。現在のモードの判定は、制約条件“`c.current != "select"`”で選択モードでないという指定により行っている。記号 Mouse に関しては、属性 `button1` を用いてボタン 1 が押されたことを判定している。これにより、このルールで生成された非終端記号は、ボタン 1 が押されている間存在し、ドラッグ中の処理が記述される。*ConstructScript* 中では、“`o = createObject(c.current)`”によりラジオボタンで選択されていた種類の図形が生成される。その図形の属性 `start`、つまり矩形の一方の頂点や線の始点は、この時点でのマウスの位置に設定される。そして、“`d.ct = addConstraint("m.pos == o.end")`”により、この生成された図形の属性 end、つまり矩形のもう一方の頂点や線の終点とマウスの位置を等しくする制約を与えられている。これにより、マウスの位置が動いた場合に図形が変形することになり、マウスドラッグによる図形の指定が可能となっている。マウスのボタンが離された場合は、記号 Mouse の属性 `button1` が変化し制約条件を満たさなくなるため LHS の記号 ObjectCreate は削除される。この際、*FinalizeScript* に記述されている制約の解除が実行される。

このように、図形の生成を 1 つのルールで簡潔に記述することができた。これは、マウスの情報を空間解析器で扱えるようになり、マウスのボタンが押されてから離されるまでの一連の処理を、*ConstructScript* と *FinalizeScript* を用いて統一的に扱うことができたことが大きく貢献している。また、制約解消系を用いて、マウスと図形の変形を関連付けたことにより、マウスのドラッグ中の処理を簡単に対応付けることができた。なお、ドラッグではなくクリックされた場合、つまり、短い時間でボタンが離された場合は、*FinalizeScript* において、条件を判別することにより処理を変更することも可能となる。

5.6.2　図形の選択

図形選択モードになっている場合に、図形上でクリックすることで図形の選択を行う処理の記述を示す。

```
s:ObjectSelect ::= m:Mouse, c:RadioButton_Mode, o:Object where (
  c.current == "select"
  m.action == "button1-press"
  m.handled == false
  contains(m.pos, o.bb)
) {} {
  m.handled = true
  o.selected = !o.selected
}
```

図形の上でクリックすると、このルールが適用され、記号 Object の属性 selected を反転する。このルールは、*ConstructScript* 中で記号 Mouse の属性 handled を変更しているため、非終端記号 ObjectSelect は直ちに削除される。なお、記号 Object は属性 selected が true の場合、図形にハンドルを付加して選択状態を示すようにバインディングがなされている。

5.6.3 図形の削除

削除ボタンが押された場合に、選択状態にある図形を削除する処理の記述を示す。

```
e:ObjectDelete ::= b:Button_Delete o:Object where (
  b.action == "pressed"
  b.handled == false
  o.selected == true
) {} {
  deleteObject(o)
}

e:ObjectDeleteFinish ::= b:Button_Delete where (
  b.action == "pressed"
  b.handled == false
  not exist o:Object where (
     o.selected == true
  )
) {} {
  b.handled = true
}
```

非終端記号 ObjectDelete を定義するルールは、選択状態にある図形を deleteObject() を利用して削除している。RHS の記号 Object が削除されるため、LHS の記号 ObjectDelete は削除される。これにより、選択状態のオブジェクトがあれば、それぞれに対してこのルールが適用され削除されていく。

非終端記号 ObjectDeleteFinish を定義するルールは、削除する図形がなくなった場合の処理を記述している。イベントバインドが終了した事を示すため記号 Button_Delete の属性 `handled` を変更している。なお、選択された図形がなくなったという制約条件は not-exist の記号を用いて記述している。

5.7 議論

ルールが適用され非終端記号が生成された場合の処理と、削除された場合の処理は、*ConstructScript* と *FinalizeScript* で記述される。この非終端記号が存在している間のユーザの入力に対する一連の処理は、制約解消系を用いて属性間の関係を指定すること実現できる。この制約解消系に対する設定は、*ConstructScript* と *FinalizeScript* を用いて簡潔に記述できる。恵比寿では、制約条件に記述された属性間の関係のみ制約解消系に設定されていたが、本手法を用いることにより、任意の関係を制約解消系に設定することが可能となる。

また、解析結果を簡単に利用することができるため、認識された図式の意味に応じた、イベントのバインディングを変更することも容易になる。さらに、エディタと解析部が統合されており、インタラクティブに処理が行われるため、図式の状態に応じて、システムに変更が可能である。つまり、解析結果に応じて処理方法に変更を加えるといった、リフレクティブなシステムが実現することができる。恵比寿では、*ConstructScript* と同様の機能を提供するアクションを導入しているが、今回提案した手法では、LHS の記号が削除された場合の処理が記述できるため、後処理が簡単に記述できる。

図形文法による記述を行うことで、宣言的なシステムの実装ができるようになる。また、その定義を変更することで、簡単にシステムに変更を加えることができる。さらに、図式の解析とその処理について同一の枠組みで記述することにより、統一的な記述が可能で、システム全体が見渡しやすくなる。加えて、空間解析器の解析に関しては、まったく変更していないため、4 章で示した空間解析器の高速化がそのまま利用することができる。逆に、導入した特別な終端記号が RHS の記号として利用される場合が多くなり、これまでの解析器では解析速度に問題があったが、提案した解析手法では、RHS の記号数に依存しない高速な解析が実現されているため実用的なシステムが実現できる。

図式に対する処理を図形文法で記述することにより、その処理や GUI コンポーネントの扱いが統一化される。これにより、実行環境による動作の違いを吸収することが可能であり、実装言語や GUI ツールキットの違いを気にせずに処理が記述することができる。

Handragen では、手書き入力を扱うために、Gesture トークンを導入している。これは、手書き入力のストローク情報とともに、ジェスチャ認識システムの結果を含んだトークンである。この Gesture トークンは、本研究で提案した特別な終端記号の拡張であると言える。これまでは、図形エディタ側で処理していた手書き入力ストロークを、空間解析器で処理できるようにするため終端記号として扱っている。これにより、手書き入力に対する処理を図形文法で処理を記述することが可能となっている。また、図式の解析結果を利用して、他の図形との関係により処理を変更することが可能となり、コンテキストに応じたジェスチャの認識を行い、認識精度を向上させている。

第6章　まとめ

インタラクティブなシステムで空間解析器を利用する際に求められる、空間解析の高速化と、図形エディタと空間解析器の統合について述べた。

空間解析器の高速化については、制約条件グラフを用いることで組合せの探索をグラフの探索に置き換え、制約条件を利用しながら漸進する探索を実現した。また、前処理を加えることで効率的な探索順序を決定できるようになった。さらに、属性値に関するテーブルを用意することで、より高速な探索を実現できることを示した。これらにより、これまでのアルゴリズムに比べ高速な解析を実現できた。

図形エディタと空間解析器の統合を実現するため、特別な終端記号を導入し空間解析器でマウスなどの情報を扱えるようにした。また、図式に対する処理の記述を行うための図形文法 i-CMG を提案した。これにより、インタラクティブシステムにおける処理の記述と、図形言語の定義を統一的に扱えるようになった。加えて、空間解析器の新たな利用法を示唆した。この提案手法を利用した応用例を示し、これまでのシステムでは実現できなかった、図式に対するインタラクションを簡単に実現できることを示した。

謝辞

本研究を進めるうえで多くの御指導をいただきました、筑波大学電子・情報工学系の田中二郎教授に深く感謝します。

本論文の審査にあたって数多くの貴重な助言・討論をいただきました、筑波大学電子・情報工学系の北川博之教授、西原清一教授、安永守利教授、加藤和彦助教授に深く感謝します。

筑波大学電子・情報工学系の志築文太郎講師、三浦元喜助手には、研究について多くの助言をいただきました。また、酒寄保隆氏、亀山裕亮氏、小川徹氏、宮下貴史氏、糸賀裕弥氏をはじめとする田中研究室の皆様及び OB の皆様には研究を進める上での議論に参加・協力していただきました。ここに感謝の意を表します。

参考文献

[Bab97a] 馬場昭宏, 田中二郎. Spatial Parser Generator の Tcl/Tk を用いた実装. インタラクション'97 論文集, pp. 71–78, February 1997.

[Bab97b] 馬場昭宏, 田中二郎. GUI を記述するためのビジュアル言語. インタラクティブシステムとソフトウェア V 日本ソフトウェア科学会 WISS '97, pp. 135–140, December 1997.

[Bab98a] 馬場昭宏. Spatial Parser Generator を持ったビジュアルシステム. 筑波大学大学院工学研究科修士論文, January 1998.

[Bab98b] 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, pp. 1385–1394, May 1998.

[Bab98c] A. Baba and J. Tanaka. Eviss: A Visual System Having a Spatial Parser Generator. *Proceedings of Asia Pacific Computer Human Interaction*, pp. 158–164, July 1998.

[Bab99] 馬場昭宏, 田中二郎. 「恵比寿」を用いたビジュアルシステムの作成. 情報処理学会論文誌, Vol. 40, No. 2, pp. 497–506, February 1999.

[Bad01] G. J. Badros, A. Borning, and P. J. Stuckey. The Cassowary Linear Arithmetic Constraint Solving Algorithm. *ACM Transactions on Computer-Human Interaction*, Vol. 8, No. 4, pp. 267–306, December 2001.

[Bal96] L. R. Balt. Full CMG parsing. Master's thesis, Leiden University, The Netherlands, July 1996.

[Bor97] A. Borning, K. Marriott, P. Stuckey, and Y. Xiao. Solving Linear Arithmetic Constraints for User Interface Applications. *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp. 87–96. October 1997.

[Cho95a] S. S. Chok and K. Marriott. Parsing Visual Languages. *Proceedings of the 18th Australasian Computer Science Conference*, pp. 90–98, February 1995.

[Cho95b] S. S. Chok and K. Marriott. Automatic Construction of User Interfaces from Constraint Multiset Grammars. *Proceedings of IEEE Symposium on Visual Languages*, pp. 242–249, September 1995.

[Cho98] S. S. Chok and K. Marriott. Automatic Construction of Intelligent Diagram Editors. *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp. 185–194. November 1998.

[Cho99] S. S. Chok, K. Marriott, and T. Paton. Constraint-based Diagram Beautification. *Proceedings of IEEE Symposium on Visual Languages*, pp. 12–19, September 1999.

[Cho03] S. S. Chok and K. Marriott. Automatic Generation of Intelligent Diagram Editors. *ACM Transactions on Computer-Human Interaction*, Vol. 10, No. 3, pp. 244–276, September 2003.

[Cos95] G. Costagliola, G. Tortora, S. Orefice, and A. D. Lucia. Automatic Generation of Visual Programming Environments. *IEEE Computer*, Vol. 28, No. 3, pp. 56–66, 1995.

[Cos98] G. Costagliola, A. D. Lucia, S. Orefice, and G. Tortora. Positional Grammars: A Formalism for LR-like Parsing of Visual Languages. *Visual Languages Theory* (Eds. by K. Marriott and B. Meyer), pp. 171–191. Springer, 1998.

[Cos99] G. Costagliola, F. Ferrucci, G. Polese, and G. Vitiello. Supporting Hybrid and Hierarchical Visual Language Definition. *Proceedings of IEEE Symposium on Visual Languages*, pp. 236–245, September 1999.

[Ead84] P. Eades. A Heuristic for Graph Drawing. *Congressus Numerantium*, Vol. 42, pp. 149–160, 1984.

[Ear70] J. Earley. An Efficient Context-free Parsing Algorithm. *Communications of the ACM*, Vol. 13, No. 2, pp. 94–102, 1970.

[Fer94] F. Ferrucci, G. Tortora, M. Tucci, and G. Vitiello. A Predictive Parser for Visual Languages Specified by Relation Grammars. *Proceedings of IEEE Symposium on Visual Languages*, pp. 245–252, October 1994.

[Fuj99] K. Fujiyama, K. Iizuka, and J. Tanaka. VIC: CMG Input System Using Example Figures. *Proceedings of the International Symposium on Future Software Technology*, pp. 67–72, October 1999.

[Fuj00] 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, September 2000.

[Fuj01] 藤山健一郎. 例示入力図を用いた Spatial Parser Generator. 筑波大学大学院工学研究科修士論文, February 2001.

[Gol91] E. J. Golin. Parsing Visual Languages with Picture Layout Grammars. *Journal of Visual Languages and Computing*, Vol. 2, pp. 371–394, 1991.

[Gol93] E. J. Golin and T. Magliery. A Compiler Generator for Visual Languages. *Proceedings of IEEE Symposium on Visual Languages*, pp. 314–321, August 1993.

[Gro96a] M. D. Gross and E. Y.-L. Do. Demonstrating the Electronic Cocktail Napkin: a paper-like interface for early design. *Conference on Human Factors and Computing Systems*, pp. 5–6. 1996.

[Gro96b] M. D. Gross and E. Y.-L. Do. Ambiguous Intentions: A Paper-like Interface for Creative Design. *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp. 183–192. November 1996.

[Har97] Y. Harada, K. Miyamoto, and R. Onai. VISPATCH: Graphical Rule-Based Language Controlled by User Event. *Proceedings of IEEE Symposium on Visual Languages*, pp. 162–163, September 1997.

[Hel91] R. Helm, K. Marruitt, and M. Odersky. Building Visual Language Parsers. *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 105–112, 1991.

[Hir91] M. Hirakawa, Y. Nishimura, M. Kado, and T. Ichikawa. Interpretation of Icon Overlapping in Iconic Programming. *Proceedings of IEEE Symposium on Visual Languages*, pp. 254–259, October 1991.

[Hon00] J. I. Hong and J. A. Landay. SATIN: A Toolkit for Informal Ink-based Applications. *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp. 63–72. 2000.

[Hos99] 細部博史, 松岡聡, 米澤明憲. HiRise: GUI 構築のためのインクリメンタルな制約解消系. コンピュータソフトウェア, Vol. 16, No. 6, pp. 33–45, 1999.

[Hos00] 細部博史, 松岡聡, 米澤明憲. GUI を対象とした線形計算による制約階層解消系の高速化. コンピュータソフトウェア, Vol. 17, No. 2, pp. 25–29, 2000.

[Iga97] T. Igarashi, S. Matsuoka, S. Kawachiya, and H. Tanaka. Interactive Beautification: A Technique for Rapid Geometric Design. *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp. 105–114, October 1997.

[Iiz01a] 飯塚和久, 志築文太郎, 田中二郎. 図形言語処理システムにおける図形エディタと空間解析器の統合. 日本ソフトウェア科学会第 18 回大会, September 2001.

[Iiz01b] K. Iizuka, J. Tanaka, and B. Shizuki. Describing a Drawing Editor by Using Constraint Multiset Grammars. *Proceedings of the International Symposium on Future Software Technology*, pp. 119–124, November 2001.

[Iiz03a] 飯塚和久, 志築文太郎, 田中二郎. 空間解析器における効率的な探索手法. 日本ソフトウェア科学会第 20 回大会, September 2003.

[Iiz03b]　飯塚和久, 亀山裕亮, 志築文太郎, 田中二郎. インクリメンタルな解析による空間解析器の高速化. 情報処理学会論文誌：プログラミング, Vol. 44, No. SIG13 (PRO18), pp. 100–109, October 2003.

[Joh79]　S. C. Johnson. Yacc: Yet Another Compiler Compiler. *UNIX Programmer's Manual*, Vol. 2B, pp. 353–387. Bell Telephone Laboratories, seventh edition edition, January 1979.

[Jou00a]　丁錫泰, 田中二郎. Rainbow: ビジュアルシステム生成系におけるレイアウト制約の実現. 情報処理学会論文誌, Vol. 41, No. 5, pp. 1246–1256, May 2000.

[Jou00b]　S. Joung and J. Tanaka. Generating a Visual System with Soft Layout Constraints. *Proceedings of the International Conference on Information*, pp. 138–145, October 2000.

[Jou01]　丁錫泰, 田中二郎. 空間パーサにおける木構造レイアウト制約の実現とその評価. 電子情報通信学会論文誌, Vol. J84-D-I, No. 9, pp. 1350–1361, September 2001.

[Kam02]　H. Kameyama, K. Iizuka, B. Shizuki, and J. Tanaka. GIGA: Graphical Definition of Production Rules in a Spatial Parser Generator. *Proceedings of the International Symposium on Future Software Technology*, October 2002.

[Kam03a]　亀山裕亮, 志築文太郎, 田中二郎. 直接操作を用いたグラフィカルな図形文法編集システム. 日本ソフトウェア科学会第 20 回大会, September 2003.

[Kam03b]　亀山裕亮, 飯塚和久, 志築文太郎, 田中二郎. GIGA: 空間解析器生成系におけるグラフィカルな文法編集システム. 情報処理学会論文誌, Vol. 44, No. 11, pp. 2565–2574, November 2003.

[Lue78]　G. S. Lueker. A Data Structure for Orthogonal Range Queries. *Proceedings of IEEE Symposium on Foundations of Computer Science*, pp. 28–34, 1978.

[Mar94]　K. Marriott. Constraint Multiset Grammars. *Proceedings of IEEE Symposium on Visual Languages*, pp. 118–125, October 1994.

[Mar96]　K. Marriott and B. Meyer. Towards a Hierarchy of Visual Languages. *Proceedings of IEEE Workshop on Visual Languages*, pp. 196–203, September 1996.

[Mar98]　K. Marriott and B. Meyer. The CCMG Visual Language Hierarchy. *Visual Languages Theory* (Eds. by K. Marriott and B. Meyer), pp. 129–169. Springer, 1998.

[Mas00]　増井俊之. スナッピングの活用, インターフェイスの街角, pp. 164–168. Unix Magazine, December 2000.

[Nak96]　中井央, 佐々政孝, 山下義行, 中田育男. LR 属性文法に基づいたインクリメンタルな属性評価. 情報処理学会論文誌, Vol. 37, No. 12, pp. 2254–2265, December 1996.

[Nak97]　中井央. コンパイラにおけるインクリメンタルな解析法. 筑波大学大学院工学研究科博士論文, July 1997.

[Nis99]　西名毅, 田中二郎. ビジュアルシステム生成系 Eviss における Action の視覚化. 日本ソフトウェア科学会第 16 回大会, September 1999.

[Oga02]　小川徹, 田中二郎. ドラッグ＆ドロップを用いたビジュアルプログラミングシステム. 情報処理学会論文誌：プログラミング, Vol. 43, No. SIG1 (PRO13), pp. 36–47, January 2002.

[Rek95]　J. Rekers and A. Schuerr. A Graph Grammar Approach to Graphical Parsing. *Proceedings of IEEE Symposium on Visual Languages*, pp. 195–202, September 1995.

[San94a]　M. Sannella. *Constraint Satisfaction and Debugging for Interactive User Interfaces*. Master's thesis, Department of Computer Science and Engineering, University of Washington, August 1994.

[San94b]　M. Sannella. SkyBlue: A Multi-Way Local Propagation Constraint Solver for User Interface Construction. *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp. 137–146. November 1994.

[Sas93]　佐々政孝, 石塚治志, 中田育男. 1 パス型属性文法に基づくコンパイラ生成系 Rie. コンピュータソフトウェア, Vol. 10, No. 3, pp. 20–36, May 1993.

[Shi03]　B. Shizuki, H. Yamada, K. Iizuka, and J. Tanaka. A Unified Approach for Interpreting Handwritten Strokes using Constraint Multiset Grammars. *Proceedings of 2003 IEEE Symposium on Human-Centric Computing Languages and Environments*, pp. 180–182, October 2003.

[Shi04]　志築文太郎, 田中二郎, 飯塚和久. 文法を用いた手書きストローク認識のための枠組み. インタラクション 2004 論文集, May 2004. （掲載予定）.

[Sug94]　三末和男, 杉山公造. マグネティック・スプリング・モデルによるグラフ描画法について. 情報処理学会研究報告「ヒューマンインタフェース」, Vol. 55, pp. 17–24, 1994.

[Tar72]　R. E. Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, Vol. 1, No. 2, pp. 146–160, June 1972.

[Tsu01]　土屋洋夢. ビジュアルシステム恵比寿における制約解消の高速化. 筑波大学第三学群情報学類卒業論文, February 2001.

[Ull88] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, Vol. 1. Computer Science Press, 1988.

[Wal90] J. Q. Walker. A Node-positioning Algorithm for General Trees. *Software Practice and Experience*, Vol. 20, No. 7, pp. 685–705, 1990.

[Wei93] L. Weitzman and K. Wittenburg. Relational Grammars for Interactive Design. *Proceedings of IEEE Symposium on Visual Languages*, pp. 4–11, August 1993.

[Wit90] K. Wittenburg and L. Weitzman. Visual Grammars and Incremental Parsing for Interface Languages. *Proceedings of IEEE Symposium on Visual Languages*, pp. 111–118, October 1990.

[Wit92] K. Wittenburg. Earley-style Parsing for Relational Grammars. *Proceedings of IEEE Symposium on Visual Languages*, pp. 192–199, September 1992.

[Yam02] 山田英仁, 飯塚和久, 田中二郎. ビジュアルシステム生成系「恵比寿」におけるジェスチャの実現. 日本ソフトウェア科学会第 19 回大会, September 2002.

[Yam03] 山田英仁. ビジュアルシステム生成系における手書き入力と図形文法との統合. 筑波大学大学院システム情報工学研究科修士論文, February 2003.

[Yos86] I. Yoshimoto, N. Monden, M. Hirakawa, M. Tanaka, and T. Ichikawa. Interactive Iconic Programming Facilities in HI-VISUAL. *Proceedings of IEEE Symposium on Visual Languages*, pp. 34–41, June 1986.

著者論文リスト

- 志築文太郎, 田中二郎, 飯塚和久. 文法を用いた手書きストローク認識のための枠組み. インタラクション 2004 論文集, May 2004. （掲載予定）.
- 亀山裕亮, 飯塚和久, 志築文太郎, 田中二郎. GIGA: 空間解析器生成系におけるグラフィカルな文法編集システム. 情報処理学会論文誌, Vol. 44, No. 11, pp. 2565–2574, November 2003.
- B. Shizuki, H. Yamada, K. Iizuka, and J. Tanaka. A Unified Approach for Interpreting Handwritten Strokes using Constraint Multiset Grammars. *Proceedings of 2003 IEEE Symposium on Human-Centric Computing Languages and Environments*, pp. 180–182, October 2003.
- 飯塚和久, 亀山裕亮, 志築文太郎, 田中二郎. インクリメンタルな解析による空間解析器の高速化. 情報処理学会論文誌：プログラミング, Vol. 44, No. SIG13 (PRO18), pp. 100–109, October 2003.
- 飯塚和久, 志築文太郎, 田中二郎. 空間解析器における効率的な探索手法. 日本ソフトウェア科学会第 20 回大会, September 2003.
- 山田英仁, 飯塚和久, 田中二郎. ビジュアルシステム生成系「恵比寿」におけるジェスチャの実現. 日本ソフトウェア科学会第 19 回大会, September 2002.
- H. Kameyama, K. Iizuka, B. Shizuki, and J. Tanaka. GIGA: Graphical Definition of Production Rules in a Spatial Parser Generator. *Proceedings of the International Symposium on Future Software Technology*, October 2002.
- K. Iizuka, J. Tanaka, and B. Shizuki. Describing a Drawing Editor by Using Constraint Multiset Grammars. *Proceedings of the International Symposium on Future Software Technology*, pp. 119–124, November 2001.
- 飯塚和久, 志築文太郎, 田中二郎. 図形言語処理システムにおける図形エディタと空間解析器の統合. 日本ソフトウェア科学会第 18 回大会, September 2001.
- K. Fujiyama, K. Iizuka, and J. Tanaka. VIC: CMG Input System Using Example Figures. *Proceedings of the International Symposium on Future Software Technology*, pp. 67–72, October 1999.
- 飯塚和久, 田中二郎. KLIC と Java のメッセージインターフェース. 日本ソフトウェア科学会第 15 回大会論文集, pp. 237–240, September 1998.