# インクリメンタルな解析による空間解析器の高速化

飯塚和久[†] 亀山裕亮[†]
志築文太郎[††] 田中二郎[††]

図式はある規則の基に描かれており，これを文法として定義することができる．この文法を，プログラミング言語の文法に対して図形文法と呼び，矩形や線などの基本図形要素がトークンとして扱われる．また，プログラミング言語と同様に，図形文法に基づいて図式を解析する空間解析器を作ることができる．これをインタラクティブな図形処理システムに適用した場合，ユーザの入力に応じて，トークンが動的に追加・削除されることになる．空間解析器は，これに応じて解析をリアルタイムにやり直す必要が生じる．我々は，すでに提案されている，図形文法を与えることで空間解析器を利用できるシステムを改良し，この解析を高速に行うインクリメンタルな解析器を実現した．高速化のポイントは，解析時に探索するトークンの組合せ数を減少させることにある．我々は，組合せを求める際に，ルールの適用条件に関するグラフを利用した．これにより，ルールが適用できる組合せを効率的に求めることができる．また，前処理を行うことで効率的なグラフの探索を行うことができる．さらに，属性値に関するテーブルを用意することでさらなる高速化が可能であることを述べる．

# A Faster Incremental Parsing Algorithm for Spatial Parser

Kazuhisa Iizuka,[†] Hiroaki Kameyama,[†] Buntarou Shizuki[††]
and Jiro Tanaka[††]

Diagrams are drawn based on certain rules. Such rules can be defined as “visual grammars.” A spatial parser is used to analyse a diagram whether the diagram follows a given grammar or not. Previous algorithms that are used in the spatial parsers try to test all the combination of tokens consisting of the diagrams whether the rules can be applicable or not. Therefore, the algorithms are problematic to be used to process diagrams that are inputed and edited dynamically by users because interactivity becomes low. We propose a new spatial parsing algorithm that realizes a faster incremental parsing by improving the existing algorithms. The algorithm examines the rules in a given grammar to construct a dependency graph between conditions of the rules. The graph is used to eliminate combinatorial test of tokens. Moreover, our algorithm examines the graph to find efficient strategies for searching. These improvements decrease the order of analysis in the average case.

## 1. はじめに

フローチャート，E-R 図，オブジェクト図などの図式は，ある規則の基に，矩形や線などの基本図形要素（トークン）を組み合わせて描かれる．この規則は文法として定義することができ，この文法を，テキストのプログラミング言語の文法に対して “図形文法” と呼ぶ．図形文法としては，Positional Grammars [1]，Relational Grammars [2]，Constraint Multiset Grammars（CMG）[3),4)]などが提案されている．

テキストのプログラミング言語と同様に，図形文法に基づいて図式を解析する “空間解析器” を作ることができる．文法から空間解析器を生成するシステムも提案されており，VLCC [5]，Penguins [6]，SPARGEN [7]，恵比寿[8]，Rainbow [9)]などがある．

図式は，一般に計算機上でインタラクティブに入力・編集されるため，空間解析を編集中に行うことによって，その解析結果を利用した様々な処理[10)~12)]を行うことが期待される．しかし，これまでの空間解析器では，トークン数が多くなった場合に解析が非常に遅くなり，リアルタイムに処理を行うことが困難であった．本論文では，この問題を解決するための，インクリメンタルな解析アルゴリズムを提案する．

本論文の構成は次のようになっている．まず，2 章

† 筑波大学大学院工学研究科
Doctoral Program in Engineering, University of Tsukuba
†† 筑波大学電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba

では CMG について説明し，既存の解析アルゴリズムとその問題点について述べる．3 章では，解析をグラフ探索に帰着して処理を行う解析アルゴリズムを提案する．4 章では，グラフ探索を効率的に行うための前処理について述べる．5 章では，制約条件によってはさらに効率的な探索が行えることを示す．6 章では実験結果について述べる．

## 2. CMG とその解析

### 2.1 CMG

CMG に基づく解析はボトムアップに行われる．また，ルール（生成規則）を注意深く記述することで，バックトラックなしに解析を行うような文法を定義できる．このため，インタラクティブなシステムにおいて，動的にトークンが追加・削除された場合の再解析における計算量を削減できる．また，CMG では広い範囲の図形言語を扱うことができる．このような理由から，我々は空間解析器に対する図形文法として CMG を利用する．

CMG におけるルールは，以下のような形で定義される．

```
T ::= T1, T2,...,Tk where (
  Constraints
) {
  AttributeAssignments
}
```

これは，RHS の記号列 $T_1$, $T_2$,...,$T_k$ に対応するトークンが存在し，それらトークンの属性が *Constraints*（制約条件）を満たした場合に LHS の T に還元されることを示している．T の属性は *AttributeAssignments*で決定される．

### 2.2 図形言語の例

例として，**図 1** に示すような計算の木を表す図形言語について説明する．この図形言語は，次の 2 つのルールで記述される．

```
<< 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.midx
  n.val := t.text
}
```

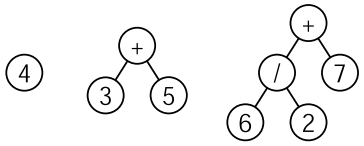


**図 1** 計算の木の例
Fig. 1 Examples of calculation tree.

```
<< ルール 2 >>
n:Node ::= c:Circle, t:Text, l1:Line,
    l2:Line, n1:Node, n2:Node where (
  c.mid == t.mid &&
  isOperator(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)
}
```

ここで，ルール 1 は木の葉にあたるノードの定義を行っている．円と文字列が存在し，その中心点が一致していて，テキストが数字であった場合は，ノードに還元されることを示している．また，ルール 2 は節となる演算子と，その 2 つの葉をまとめてノードを再帰的に定義している．

### 2.3 既存の解析アルゴリズム

Chok らの提案した CMG に基づく解析アルゴリズム[4),13)]は以下のようなものである．

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

ここで，D は現在のトークンテーブルを表す．解析は，D に対してルールを適用していくことで行われる．D に対して適用できるルールがなくなった時点で解析が終了する．EvaluateRule() は，引数で与えられたルールについて処理を行う．ルールが適用された場合は true を返し，そうでない場合は false を返す．AllCombination() は，T で与えられる RHS のトークンの種類に対応した組合せを，D におけるすべてのトークンの組合せから生成する．このアルゴリズムは，RHS に該当するすべてのトークンの組合せに対して制約条件をチェックすることにより，ルールをチェックしている．すべての組合せに対して探索を行うため，D のトークン数を $N$，RHS の記号数を $k$ とすると，$O(N^k)$ の組合せを探索することになる．

### 2.4 インタラクティブシステムへの応用

インタラクティブな図形処理システムでは，図形が動的に追加・削除される．解析器では，これに対応するトークンが D に対して追加・削除されることになる．このときは，再び解析が行われるが，解析は最初からやり直されるのではなく，現在のトークンテーブルに対して行われる．つまり，すでに構築された解析木をそのまま利用することでインクリメンタルに解析が行われる．

しかし，一般のルールでは RHS の記号数 $k$ は 2 以上となる．また，計算の木の例のように $k$ が 6 程度になることもよくある．よって，このようなアルゴリズムでは，トークン数が増えた場合，インタラクティブなシステムでリアルタイムに処理することは困難である．

## 3. 制約条件グラフを利用した解析

### 3.1 ルールと制約条件

ルールにおける制約条件は様々なものが記述できるが，一般には，ある 1 つのトークンの属性が特定の値を持つ制約条件と，2 つのトークンの属性の比較に関する制約条件の 2 つが多く利用される．ここでは，前者を単一トークンに関する制約と呼び，後者を 2 つのトークンに関する制約と呼ぶことにする．単一トークンに関する制約の例として，矩形が白で塗りつぶされている，円の半径が 20 ピクセルであるといったようなものがある．2 つのトークンに関する制約としては，直線の始点が円の中心と一致している，2 つの矩形の線の色が等しいといったようなものがある．

図形言語では，ほとんどの場合これら 2 種類の制約条件で記述される．つまり，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になる．Chok らのアルゴリズムでは，それぞれの組合せが制約条件を満たすかチェックしていた．我々は，この逆のアプローチをとり，制約条件を満たすようなトークンから組合せを求めるようにする．

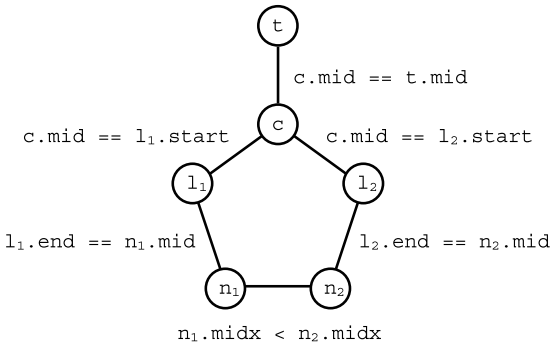


**図 2** 制約条件グラフ
Fig. 2 Constraint graph.

たとえば，円の半径が 20 ピクセルであるといった制約条件を利用することですべての円に関する組合せをチェックする必要がなくなる．同様に，直線の始点が円の中心と一致しているような，直線と円の組合せは，すべての直線と円の組合せよりも少なくなることが期待される．

### 3.2 制約条件グラフ

制約条件を満たすトークンから，効率的に組合せを求めるために制約条件とトークン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 のトークンをノードとし，2 つのトークンに関する制約をエッジと考えることで，RHS のトークンと制約条件に関するグラフが構成できる．計算の木の例でみると，ルール 2 の場合は**図 2** のようになる．たとえば，トークン n1 は l1 と n2 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` という制約条件をエッジとしている．

### 3.3 探索方法

制約条件グラフにおいて，制約条件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，組合せ探索を，グラフ探索に置き換える．このようにすることで，制約条件を満たす組合せをインクリメンタルに求めることができ，効率的に組合せを求めることができる．

グラフの各ノードを訪問し，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．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．

各ノードを訪問した際に，そのノードに関する制約条件をチェックする．この制約条件は大きく3種類に分けられる．1つは，そのトークンに関して単一トークンに関する制約である．2つめは，2つのトークンに関する制約である．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，このエッジに対応する制約条件を，このトークンと，1つ前に訪問したノードに対応するトークンと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すでに訪問したノードにつながっている場合は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．これはすでに訪問したノードで構成される3つ以上のトークンの属性に関する制約条件である．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，そのトークンの組合せは，ルールのすべての制約条件を満たすことになる．

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．

各ノードの訪問順序は，開始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，計算の木では，ルール2の訪問順序は**表1**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る．

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，2つのトークンに関する制約条件であるので，前述のとおり，これを満たす組合せは非常に限られてくる．多くの場合では，1つであると考えられる．このような場合では，バックトラックは発生しない．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．

**表1** 訪問順序の例
Table 1 Example of visiting order.

| 開始ノード | 探索順序 |
|---|---|
| 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 |

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，エッジが存在しない場合でも，3つ以上のトークンの属性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．

### 3.4 探索開始ノード

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．

2.3節で述べたように，解析が終了した時点でDに含まれるトークン（古いトークン）はどのルールも適用できない状態になっている．このことは，新しいトークンが追加された場合，古いトークンのみで構成される組合せでは，どのルールも適用できないということである．つまり，ルールが適用できる場合は，その組合せの中に必ず新しいトークンを含むということである．よって，探索開始ノードとして新しいトークンに対応するノードを選び，そのノードのトークンを新しいトークンとして探索を開始すれば，Dにおけるすべての可能な組合せをチェックしたことになる．

なお，新しいトークンに対応するノードは複数のノードに対応する場合がある．計算の木の例においては，Circleの場合は，ルール1のcとルール2のcの2つがある．Nodeの場合はルール2のn1とn2の場合がある．このような場合は，それぞれのルールにおいてそれぞれのノードから探索を行えばよい．

複数のトークンがまとめて追加される場合があるので，これらをDとは別に保持する必要がある．これらのトークンを未処理トークン集合と呼び，Uで表現することにする．Uのトークンはルール適用がチェックされていないトークンの集合となる．新しくトークンが追加された場合には，そのトークンをDとUに追加し，その後に解析をはじめる．解析では，Uの中から1つのトークンを取り出し探索を開始する．この探索で組合せが見つかった場合はルールが適用され，このトークンで開始した探索がすべて失敗した場合には，このトークンを含むような組合せではどのルールも適用できないと分かる．このようにしてUのトークンを1つずつチェックし，Uが空集合となるまで解

析を行うことにより解析が終了する．

以上のような探索開始ノードの選び方をすることにより，探索する組合せの範囲を限定させ，さらに適用するルールも最低限のルールに対してチェックを行うだけで解析を行うことができる．また，Chok らのアルゴリズムと同様に，すでに構築された解析木をそのまま利用しているので，インクリメンタルに解析が行われている．

### 3.5 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．

```
routine Parse()
  while NotEmpty(U)
    S := Pop(U)
    for each P in DependRule(S)
      if EvaluateRule(P, S) then
        break
      endif
    end
  end
end

routine EvaluateRule(P, S)
  for each A in StartNode(P, S)
    M := InitialTokenList(A, S)
    if Visit(P, M, A) then
      return true
    endif
  end
  return false
end

routine Visit(P, M, A)
  C := Constraints(P, M, A)
  if !SatisfiesConstraint(M, C) then
    return false
  endif
  if Finished(M) then
    N := NewNonTerminal(M, P)
    D := (D\M) ∪{N}
    U := (U\M) ∪{N}
    return true
  endif
  A := NextNode(P, M)
  for each S in GetTokens(A, D)
    M := SetToken(M, A, S)
    if Visit(P, M, A) then
      return true
    endif
  end
  M := UnsetToken(M, A)
  return false
end
```

Parse() は解析の最初のフェーズで，U の中から 1 つのトークンを抜き出して，そのトークンが利用されるルールをそれぞれチェックしている．EvaluateRule() では，各ルールにおける探索開始ノードを StartNode() で決定し，それぞれに対して探索を行う．ここで利用されている M は，各ノードに対応するトークンの列を保持する．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 M を初期設定しており，開始ノードの対応した位置に開始トークンを設定し，その他の部分には nil を設定する．

Visit() は，制約条件グラフを探索する部分である．最初に，こ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，制約が成り立つかチェックする．ここで制約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訪問元のノードに戻る．

制約が成り立った場合は，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，M 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は終了し，その M はルールを適用できるので，新しい非終端記号を生成する．この記号は，D に追加されるほかに，U にも追加される．また，D とともに U から M に含まれるトークンが削除される．

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．次の訪問ノードに対応するトークンを GetTokens() で取得する．それぞれのトークンに対して SetToken() で M に設定した後，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，M に設定されたトークンを UnsetToken() で取り消した後に訪問元ノードに戻る．

### 3.6 トークンの追加・削除

トークンの追加は，ユーザがキャンバスから図形を描画した場合に生ずる．このときの処理は以下のようになる．

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

つまり，トークンテーブル D と未処理トークン集合 U に新しいトークンを追加し解析を行う．

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，対応する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)
  if T ∈ D then
    D := D\{T}
    U := U\{T}
    Parse()
  else
    X := {}
    P := Parent(T)
    do
      X := X ∪ (Children(P)\{T})
      T := P
      P := Parent(T)
    while P != nil
    D := D\{T}
    U := U\{T}
    InsertTokens(X)
  end
end
```

削除されたトークンが D に存在する場合は，D から削除する．同様に U からも（含まれていれば）削除される．一方，削除されたトークンが，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 に利用されていることになる．この非終端記号は RHS のトークンを失うため存在できなくなるので削除する必要がある．このとき他の RHS トークンは D に戻され，U にも追加される．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再帰的に処理を行う．

### 3.7 計 算 量

EvaluateRule() 本体は，開始ノードに対する繰返しがあるだけなので，トークン数 ($N$) に影響されない．Visit() に関しては，次のノードを探索する際に，GetTokens() で得られたトークンすべてに対して訪問しているので $O(N)$ のコストがかかる．また，Visit() は RHS の記号数 $k$ から開始ノードを除いた $k-1$ 回だけ再帰的に訪問される．以上をまとめると，EvaluateRule() 全体の計算量は，最悪の場合 $O(N^{k-1})$ になる．Chok らのアルゴリズムでは $O(N^k)$ であったので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，平均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あり，この場合はバックトラックが発生しないためである．すべての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．計算の木のルール 2 では，n1 と 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．これらに対応するトークンは，一般に 1 対 1 に対応するため，バックトラックが発生しない．よって，表 1 で示した訪問順序の場合の計算量は $O(N^2)$ となる．なお，訪問順序を適切に選び，n1 と 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. 前処理を加えた探索

前章で述べたように，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，以後の探索で訪問可能なノード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．各エッジの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，そのトークンに対して EvaluateRule() が呼ばれるので，この中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 GetTokens() で求めるために利用できる．さらに，訪問順序を決定する際にも利用する．Visit() で NextNode() で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．

## 5. 属性値テーブルの利用

前章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，たとえば円の中心と線の開始点が等しいといったようなものであり，多用される制約条件である．

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．すべての訪問がこのようなエッジで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は EvaluateRule() は $O(1)$ で処理が行える．

なお，このような手法は，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，2 点間の距離がある値以下であるという条件や，一方のトークンの点がもう一方のトークンの矩形領域に含まれるといった条件などでも応用することが可能である．

## 6. 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．実験に利用した図形言語は，計算の木，折れ線，リスト構造の 3 つである．

**計算の木**　2.2 節で示した計算の木を表す図形言語である．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**　複数の線分で構成される折れ線を扱う図形言語である．入力図形として，1 つにつながった折れ線を利用した．なお，付録 A.1 に定義を示してある．

**リスト構造**　データを矢印でつなぐことでリスト構造を表した図形言語である．入力図形として，一列につながったリストを利用した．なお，付録 A.2 に定義を示してある．

これらの図形言語に対して，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．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．比較する解析器として，2.3 節で示した Chok らのアルゴリズム，制約条件グラフを利用した解析アルゴリズムの 2 つに加え，前処理を加えたものと，ハッシュ表を利用したものについても扱った．

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 1800+，512MB Memory
- Java 2 SDK 1.4.1，interpreter mode

計算の木に対する総解析時間に関する実験では，**図 3** に示すような結果になった．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．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．

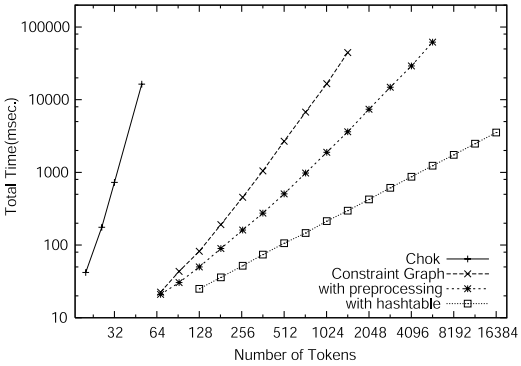


**図 3** 計算の木に対する総解析時間
Fig. 3 Parsing time of calculation tree.

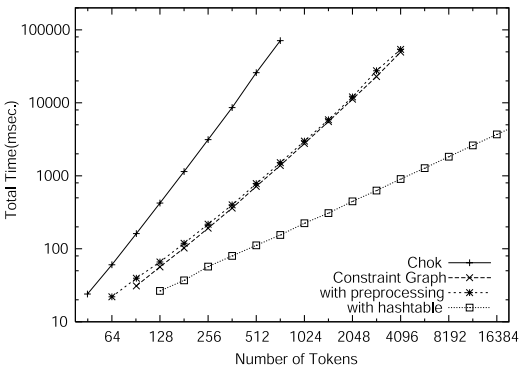


**図 4** 折れ線に対する総解析時間
Fig. 4 Parsing time of line.

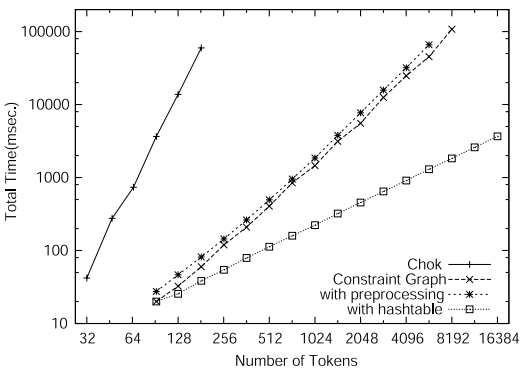


**図 5** リスト構造に対する総解析時間
Fig. 5 Parsing time of list.

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．

折れ線に対する実験は**図 4**，リスト構造に対する実験は**図 5** 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がほぼ同じ結果になった．これは，制約条件がすべて 1 対 1 の関係であるので，訪問順序に探索時間が影響がなかったためである．

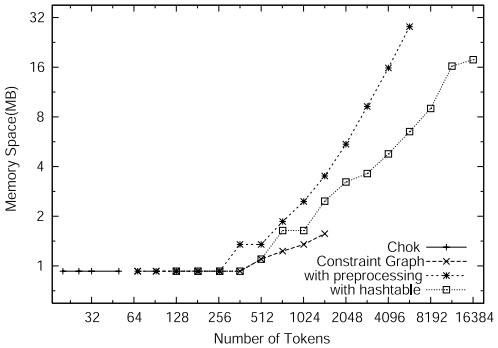


**図 6** 計算の木の解析における記憶領域
Fig. 6 Memory space in calculation tree parsing.

図 3 の実験における記憶領域については，**図 6** のような結果になった．前処理を行った場合の記憶領域は，$O(N^2)$ 必要であった．ハッシュ表を利用した場合は $O(N)$ であった．なお，実験環境の影響で，記憶領域について 0.93MB 以下の情報が取得できなかった．よって，解析が低速であったその他の解析器では，有効な情報を取得することができなかった．

## 7. 関 連 研 究

CMG に基づく解析器[13]は，Chok らによって提案された．このアルゴリズムでは，ルールを階層化して管理することにより無駄なルール適用を削減させている．しかし，この手法でも無駄なルール適用が発生する場合があった．我々が提案したアルゴリズムでは，ルール適用はグラフ探索の開始トークンにより決定される．これにより，ルール適用を必要最小限に抑えている．Balt によって提案された CMG に基づく解析器[14]は，Chok らと同じルール適用方法に加え，単一トークンに関する制約条件を利用している．我々は，単一トークンに関する制約条件に加えて 2 つのトークンに関する制約条件を利用することで高速な解析を実現している．また，Wittenburg は CMG に比べて記述力の弱い Relational Grammars に対して Earley-style [15]の解析を用いて[16]いる．

## 8. 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，空間解析の高速化手法について提案を行った．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，これまでの解

析結果を利用することで，探索開始ノードを適切に選択した．さらに，前処理を加えることで高速な探索を実現できることを示した．以上のようなインクリメンタルな解析手法により，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．

## 参 考 文 献


1) Costagliola, G., Lucia, A.D., Orefice, S. and Tortora, G.: Positional Grammars: a Formalism for LR-like Parsing of Visual Languages, *Visual Languages Theory*, Marriott, K. and Meyer, B. (Eds.), pp.171–191, Springer (1998).
2) Ferrucci, F., Tortora, G., Tucci, M. and Vitiello, G.: A Predictive Parser for Visual Languages Specified by Relation Grammars, *Proc. IEEE Symposium on Visual Languages*, pp.245–252 (1994).
3) Helm, R., Marriott, K. and Odersky, M.: Building Visual Language Parsers, *Conference Proceedings on Human Factors in Computing Systems*, pp.105–112 (1991).
4) Chok, S.S. and Marriott, K.: Parsing Visual Languages, *Proc. 18th Australasian Computer Science Conference*, pp.90–98 (1995).
5) Costagliola, G., Tortora, G., Orefice, S. and Lucia, A.D.: Automatic Generation of Visual Programming Environments, *IEEE Computer*, Vol.28, No.3, pp.56–66 (1995).
6) Chok, S.S. and Marriott, K.: Automatic Construction of Intelligent Diagram Editors, *Proc. ACM Symposium on User Interface Software and Technology*, pp.185–194 (1998).
7) Golin, E.J. and Maglierry, T.: A Compiler Generator for Visual Languages, *Proc. IEEE Symposium on Visual Languages*, pp.314–321 (1993).
8) 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 (1998).
9) Joung, S. and Tanaka, J.: Generating a Visual System with Soft Layout Constraints, *Proc. International Conference on Information*, pp.138–145 (2000).
10) 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 (2001).
11) Iizuka, K., Tanaka, J. and Shizuki, B.: Describing a Drawing Editor by Using Constraint Multiset Grammars, *Proc. International Symposium on Future Software Technology*, pp.119–124 (2001).
12) 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 (2002).
13) Chok, S.S. and Marriott, K.: Automatic Construction of User Interfaces from Constraint Multiset Grammars, *Proc. IEEE Symposium on Visual Languages*, pp.242–249 (1995).
14) Balt, L.R.: Full CMG parsing, Master's Thesis, Leiden University, The Netherlands (1996).
15) Earley, J.: An Efficient Context-free Parsing Algorithm, *Comm. ACM*, Vol.13, No.2, pp.94–102 (1970).
16) Wittenburg, K.: Earley-style Parsing for Relational Grammars, *Proc. IEEE Symposium on Visual Languages*, pp.192–199 (1992).


## 付　　録

### A.1 折れ線の定義

折れ線を表す図形言語は，次のようなルールで定義される．

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

### A.2 リスト構造の定義

リスト構造を表す図形言語は2つのルールから定義される．まず，リストの終端を定義するルールは，次のようになる．

```
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
}
```

また，リストを再帰的に定義するため，次のルールを利用する．

```
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
  }
```

(平成 14 年 12 月 24 日受付)
(平成 15 年 7 月 1 日採録)

**飯塚 和久**（学生会員）

1999 年筑波大学大学院理工学研究科修士課程修了．現在，同大学院工学研究科博士課程在学中．図形言語の処理系，空間解析器とその応用に興味を持つ．日本ソフトウェア科学会，ACM 各会員．

**亀山 裕亮**（学生会員）

1975 年生．1998 年筑波大学第三学群工学システム学類卒業．同年同大学院工学研究科博士課程入学．プログラミング言語や図形言語の処理系に関する興味を持つ．日本ソフトウェア科学会会員．

**志築文太郎**（正会員）

1971 年生．1994 年東京工業大学理学部情報科学科卒業．2000 年同大学大学院情報理工学研究科数理・計算科学専攻博士課程単位取得退学．博士（理学）．現在，筑波大学電子・情報工学系講師．ヒューマンインタフェースに関する研究に興味を持つ．日本ソフトウェア科学会，ACM，IEEE Computer Society，電子情報通信学会，ヒューマンインタフェース学会各会員．

**田中 二郎**（正会員）

1975 年東京大学理学部卒業．1977 年同大学大学院理学系研究科修士課程修了．1978 年から米国ユタ大学に留学．1984 年同大学計算機科学科博士課程修了，Ph.D. in Computer Science．1993 年より筑波大学に勤務，電子・情報工学系助教授を経て，現在，電子・情報工学系教授，第三学群情報学類長．また，筑波大学先端学際領域研究センター（TARA センター）の実世界インタラクション研究プロジェクト研究代表者を併任．プログラミング言語やヒューマンインタフェースに興味を持つ．ACM，IEEE Computer Society，日本ソフトウェア科学会，電子情報通信学会，人工知能学会，ヒューマンインタフェース学会各会員．2002 年から ACM 日本支部において，CACM 日本語版編集長をつとめる．