ソフトウェア基礎論　17－6
（1986. 7. 3）

属性文法によるＵＮＩＸファイルシステムの記述

Specification of the UNIX File System based on Attribute Grammar

篠田陽一　　片山卓也
Yoichi SHINODA　　Takuya KATAYAMA
東京工業大学 情報工学科
Tokyo Institute of Technology, Department of Computer Science

あらまし　属性文法にもとづくオブジェクト指向的Ｕｎｉｘファイルシステムの記述について述べた。属性文法は元来プログラム言語の形式的意味記述の道具として導入され、コンパイラの自動生成に関連して研究が行われてきたが、その本質は木構造の上での属性値の関数的計算であり、この原理はもっと広い分野に応用可能である。本報告では、Ｕｎｉｘの階層的ファイルの木構造をこの手法を用いて記述することを試みた。階層的ファイルシステムとコンパイラ間にはどちらも属性つき木構造を主な計算対象とする大きな共通点がある一方、コンパイラでは構文木が一度構成されてしまえばそれ以後不変であるのに対し、ファイルシステムでは外部からのコマンドによって属性つき木の構造が変化する相違点がある。本稿では、オブジェクト指向的手法によってこれを簡潔に見やすく記述するための方法を考察した。これは、通常の属性文法に一時的属性の意味規則およびメッセージを追加し、さらに属性値の伝播的変化のメカニズムを取り入れたものである。まず属性文法にもとづく計算機構全般について述べた後、属性文法的オブジェクト指向計算モデルを導入し、つぎにそれを用いた階層的ファイルシステムの記述法について述べた。

Abstract　Specification of the Unix file system based on attribute grammar with object oriented extension is described. While the attribute grammer, introduced by D.Knuth as a method of specifying the formal semantics of the programming languages, has been exploited as compiler construction tools, it's important property is a purely applicative calculation of the value of the attributes over the tree structure, and this property can be applied to more general aspects in the program specification. The model which combines the attribute grammar and the object oriented programming is introduced to express the dynamic behavior of the system.

１．はじめに

属性文法[Knuth]はプログラム言語の意味記述のために、D.E.Knuthによって導入された形式システムであり、従来はプログラミング言語の仕様記述言語や、そのコンパイラの記述を中心に研究が進められてきた。

属性文法の基本原理は木の上での属性値の関数的計算であり、正統的な属性文法では構文木上での、また、ＡＧ［篠田・橘・片山］では計算木上での属性計算が記述されている。言語の処理以外への属性文法の応用としては、構造エディタの仕様記述[Reps]が知られているが、一般にソフトウェアの記述では木構造が用いられる場合が多く、このような一般的な局面で属性文法的な記述は有用であると考えられる。

例えば、階層的ファイルシステムやソースコード管理システム、ＣＡＤシステム、通信プロトコルの記述のように、全体が部分の集まりで記述され、かつ部分の間での情報の交換や、部分の所有する情報に操作を施したりすることによって表現されるシステム、あるいは、もっと一般的に段階的詳細化・抽象化の過程を属性文法的な

手法で記述することが考えられる。

本報告では、このような考察を背景に、一般的なシステム記述を対象とする属性文法に基づいた計算モデルを提案する。

2．属性文法概説

2.1. 属性文法の定義

属性文法Gは、次のような3つ組として定義される。

$G=<G_u, A, R>$

(1) 文脈自由文法(cfg) $G_u=(V_N, V_T, P, S)$ はGの基底文脈自由文法であり、Gの構文規則を定める。ここで、V_Nは非終端記号の集合、V_Tは終端記号の集合であり、$V=V_T \cup V_N$とする。またSは初期記号である。　Pは生成規則の集合であり、p　Pは、$X_0 \rightarrow X_1 X_2 \ldots X_n$の形をしているものとする。

(2) Aは属性の集合であり、記号X　Vの相続属性の集合をINH(X)、合成属性の集合をSYN(X)、$INH(X) \cap SYN(X)=\phi$とするとき、$A=\cup_{x \cup}(INH(X) \cup SYN(X))$である。ただし、初期記号Sについては$INH(S)=\phi$、終端記号X　V_Tについては$SYN(X)=\phi$とする。また、Xの属性aを$X.a$で表してもよいことにする。

(3) 生成規則p　Pに対し、意味規則の集合R(p)が定められる。各々の生成規則は、

(a) X_0の合成属性が$X_0, \ldots, X_n$の他の属性からいかにして定められるか、および、

(b) $X_1, \ldots, X_t$の相続属性が$X_1, \ldots, X_i$の他の属性からいかにして定められるか

を定めており、X_iの属性aが$X_{k(j)}$の属性b_j(j=1,...,r)を用いて計算されるとき、

$$X_{i.a}=F(X_{k(1).b_1}, \ldots, X_{k(r).b_r})$$

のような形で表すことができる。

$R=\cup_p R(p)$はGの意味規則の集合を与える。

2.2. 属性文法による記述の特徴

属性文法に基づく記述の特徴は、良くも悪くも、「rule-based」であること、および「関数的」であることに帰着される。すなわち、

(1) 意味記述が構文規則ごとに独立しており、全体を独立した断片的な記述の集合として構成できる。

(2) 意味規則によって与えられる各々の属性の値は、単一代入性を持ち、記述が関数的で明解である。

という利点とともに、これらの長所に対応して

(1)' ルールを構成するために導入した非終端記号において、属性値を必要とされる部分まで渡してやるための属性生起及びコピー規則が記述量を増し、全体の見通しを悪くする。

(2)' 属性の単一代入性が原因となって、記述しにくい処理がある。

等の問題点が指摘されている。

また(2)'から容易に想像できることであるが、属性文法は静的な意味の記述に向いており、動的な意味の記述は、(ア)構文木の構造を変化させることが出来ないこと、(イ)属性の計算は唯一回だけ行われ、その値を変更することはできない、という理由から困難であることが理解される。

従って、属性文法に基づいて動的な意味の記述を行うには、

(1) 属性つき木の変更

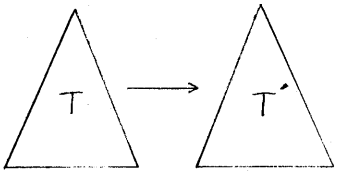
(a) 構文木の構造を変化させうること。

(b) 属性の値を変化させうること。

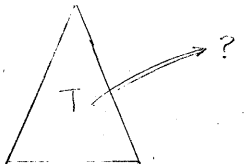
などの拡張を行うとともに、(1)'で指摘されている問題を解決し、システムの本質的な部分を見失わないようにするために、

(2) 属性つき木の内部状態に関する問い合わせの機能

を付け加える必要がある。(図2.1.)



(a)属性つき木の変更



(b)属性つき木の内部状態の問い合わせ

図2.1.属性文法に付加すべき機能

3．属性文法に基づく計算機構のモデル

本章では最初に、本稿で提案される計算モデルの基礎を形成する２つの計算モデルについて概説する。

3.1. 汎用プログラミング言語ＡＧ

属性文法の属性値評価過程は、少し見方を変えその形式に多少の変更を加えると、一般のプログラムの計算を記述するための計算モデルを与える。[片山] このモデルに基づいて設計された言語がＡＧである。

図3.1.に属性文法と属性文法型計算モデルの対応を示す。

属性文法	属性文法型計算モデル
終端記号	モジュール
生成規則 $X_0 \rightarrow X_1 X_2 \ldots X_n$	モジュールX_0のモジュール $X_1, X_2, \ldots, X_n$への階層的分割
相続，合成属性	モジュールの入出力属性
意味規則	関連するモジュール($X_0, X_1, \ldots, X_n$)の属性間の関数的記述
導出木	計算木

図3.1.属性文法と属性文法型プログラミングの対応

属性文法と、属性文法型計算モデルの最大の相違点は、構文木の作られかたにある。２章で触れたように、属性文法では、付随するcfgによる構文解析によって構文木が作られたあとで属性計算がおこなわれるが、属性文法型計算モデルにおける構文木（計算木）は、構文規則に付随する属性の上の述語（分割条件）により制御を受けながら計算の進行と共に構成されていく。

3.2. Synthesizer Generator

Repsによる構文エディタの属性文法による仕様記述では、基本的なエディティングのモデルとして、以下のようなものを想定している。

(a)展開されない非終端記号の処理

構文エディタによるエディティングの過程においては、展開されない非終端記号が出現する。これを、全ての非終端記号Ｘに対してＸ→⊥の形の生成規則と、Ｘの合成属性を定める適当な意味規則を付加することによって解決する。（図3.2.）

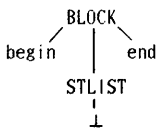


図3.2. Ｘ→⊥規則の例

(b)エディティングによる構文木の変化

属性つき木の変化は、カーソルが位置するノードにおいて、そのノードを頂点とするような木を刈り込んだり、free-standing tree（初期記号を根に持たないような木）を接ぎ木したりすることによっておこなう。当然のことながら、接ぎ木のおこなわれるノードにおいて非終端記号は一致していなければならない。free-standing treeの根において、その相続属性は任意の値を持っても良いことにするが、この木の内部の属性の値は無矛盾であるとする。一般にこのような木を接ぎ木して得られた木の属性値は矛盾を含むが、属性を再評価することによって全体を再び無矛盾な状態に戻すことができる。

この更新操作は、論理的には新しく与えられた終端記号列に対して属性つき木を新しく構成することと等価であるが、インクリメンタルに属性の評価を行うことで、効率の向上を図っている。

3.3. 属性文法的オブジェクト指向計算モデルの提案

本稿で提案する計算モデルは、基本的には属性付き木を「状態」とするシステムであり、その上での状態の変化を可能な限り属性文法を用いて表現しようとするものである。

「状態」として用いる属性付き木には、それをデータ構造としてとらえたとき、実際にそのデータに固有であると考えられる属性、及びそれらの間の意味関数のみを付加する。例えば、終端記号が表現する特定の値、ノード間の属性伝搬によってデータ構造全体の意味に一貫性を持たせるための属性や意味関数などがこれに相当する。

この「状態」を表す木の上での「操作」は、非終端記号において、その操作に特有な計算用の一時属性や計算を進めるための構文規則を与えることによって表現する。これは、非終端記号を１つのオブジェクトとしてとらえたとき、各々の「操作」に特有なメソッドを記述する事に相当する。

また、このように表現されるデータ構造の上での計算

は、非終端記号から非終端記号に送られる「操作」の要求、すなわちメッセージの送受と、そのメソッドに特有な属性の評価によって進んで行く。直感的には、静的（といっても、その構造や属性値は変化するのだが）な属性つき木の上に、計算が行われる瞬間だけオーバーラップして現れる計算木（3.1.節の属性文法型計算モデルにおける計算木）を考えればよい。

図3.3.は整数を表すデータ構造と、その値の計算をこの基本概念に沿って記述したものである。

```
N → I
    I.scale = 0

    :value(radix) → I:value
        I.radix ← radix
        N.value ← I.value

I → I D
    I2.scale = I1.scale + 1
    D.scale = I1.scale

    :value → I2:value D:value
        I1.value ← I2.value + D.value
        I2.radix ← I1.radix
        D.radix ← I1.radix
I → D
    D.scale = I.scale

    :value → D:value
        D.radix ← I.radix
        I.value ← D.value

D → ε

    :value
        D.value ← D.digit×D.radix^D.scale
```

図3.3.　整数の記述例

この記述例で、X→X0X1...Xnの部分と、等号で結ばれた式の部分は通常の属性文法において用いられている構文規則と意味規則の記述とほぼ同じ意味を持ち、それぞれデータ構造の静的な構造と、静的な属性間の意味規則を与える。前者を（データ）構成規則、後者を静的意味規則と呼ぶ。”：”で始まる行は、これが構成規則の左辺の記号が解釈することのできる操作、すなわちメソッドを定義することを表しており、”→”の右辺は、構成規則の右辺に現れる記号に対してどのような操作をメッセージを送ることによって要求するのかを記述する。操作名の後にある（）で囲まれた部分は、その内部の名前が相続属性として木の外部から操作に付随したパラメータとして与えられることを示している。また、”←”は対応する操作が適用される場合に現れる一時的な属性の意味規則を定義することを表す。例えば構成規則Ｉ→ＩＤにおいて、非終端記号Ｉはvalueという操作に対し、右辺のＩとＤにそれぞれvalueという操作を要求し、右辺のＩの相続属性であるradixを設定し、さらに自分自身の合成属性は右辺の合成属性Ｉ2.valueとD.valueから計算することを表している。

図3.4.は、この記述に基づいて構成された整数を表す構造（左側）とその値を求めた場合の計算の為の木の履歴（右側）を示したものである。この図では、二つの木が別々に書かれているが、対応するノードは同一のものであり、計算が行われている最中にはオーバーラップしている。

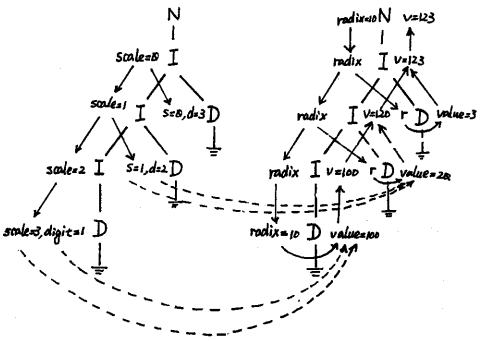


図3.4.　Ｎの構造とＮ:valueの評価

図3.4.で、非終端記号Ｄの属性として、構成規則でも一時属性でも与えられない属性digitが出現しているが、これについては後述する。

次に、属性付き木の属性の値を変更するような操作の記述と意味を示す。図3.5.は、指定された桁の数字を指定されたものに変更するような操作changeを実現するために、図3.3.に付加すべき記述を表したものである。

ここでは、構成規則Ｉ→ＩＤ及びＩ→Ｄにおいて、それぞれＤの属性digitに新しい値newdigitが設定されている。計算規則 :change → Ｄは、Ｄ.digitを視界に入れるために必要である。また、条件付きの計算規則を用いることによって、無駄な木の成長が抑制されるようにな

っている。図3.6.に操作:changeが評価されている様子を示す。

なお、この例では表現されていないが、変更された静的属性に依存するような属性が意味規則によって記述されているならば、それらの属性が一貫性を保つように、依存する属性が更新される。

```
N → I
    ...
    :change(changepos,newdigit) → I:change
        I.changepos ← changepos
        I.newdigit ← newdigit

I → I D
    ...
    :change
        → D
            when I1.scale == I1.changepos - 1
                D.digit ← I1.newdigit
        → I2:change
                I2.changepos ← I1.changepos
                I2.newdigit ← I1.newdigit

I → D
    ...
    :change → D
        when I.scale == I.changepos
            D.digit ← I.newdigit
```

図3.5. :changeの記述

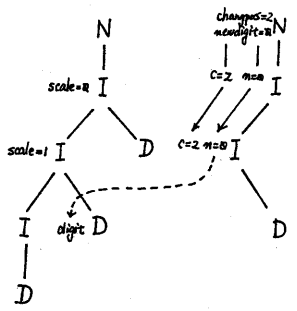


図3.6. N:changeの評価

さらに、属性付き木の構造を変化させる機構の内、木の延長を取りあげ、整数の例で説明しよう。木の延長は、延長が起こるノードにおいて、現在適用されている構成規則の代わりに用いる構成規則（の列）、および変更の前後において満たされるべき属性間の関係を記述することによって行うことができる。

図3.7.は、図3.3.に、操作:extendを組み入れるために必要な記述を示している。

```
N → I
    ...
    :extend(newdigit) → I:extend
        I.newdigit ← newdigit

I → I D
    ...
    :extend → I2:extend
        I2.newdigit ← I1.newdigit

I → D
    ...
    :extend
        substitue with I → ID, I → D
            I1.scale = I0.scale
            D1.digit = D0.digit
            D2.digit = I0.newdigit
```

図3.7. N:extendの記述

図3.8.は、N:extendの評価による効果を示しており、計算木が、変更の起こるノード（I_0）に達した後、替わりの構成規則I→ID、I→Dが適用され、新しい木が構成される様子を表している。新しく構成された木の属性には、元の木から得られるもの（I_1.scale,D_1.digit）、計算木から得られるもの（D_2.digit）、および意味規則によって伝搬して得られるもの（I_2.scale,D_2.scale）があることが理解される。

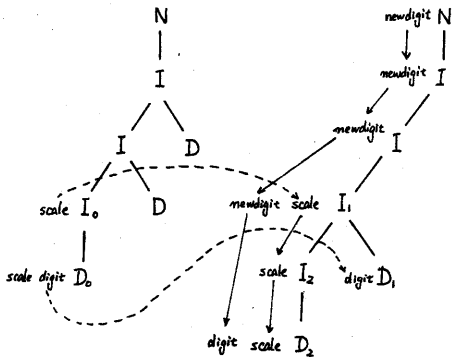


図3.8. N:extendの評価

また、先にDのdigitは構成規則でも一時属性としても与えられていないと書いたが、この属性はextendで与えられることが解る。

本章で提案した計算モデルをまとめると、以下のように表現される。

属性文法に基づくオブジェクト指向計算モデル

＝　静的な記述　＋　動的な記述

静的な記述　＝　静的属性　＋　静的意味規則

動的な記述　＝　一時属性　＋　一時意味規則

＋　メッセージ

動作のメカニズム

・メッセージの送受

・一時属性の評価

＋　静的属性の伝搬による変化

4．階層的ファイルシステムの記述

本章では、UNIXにみられるような階層的に構成されるファイルシステムを３章で提案するモデルに従って記述する。

ここで扱うファイルシステムは、概念的には図4.1.に示すような階層的なディレクトリ構造によって構成される。

ファイル／ディレクトリ名はディレクトリ・エントリで管理され、ファイル・システムの総容量が常に更新されている。

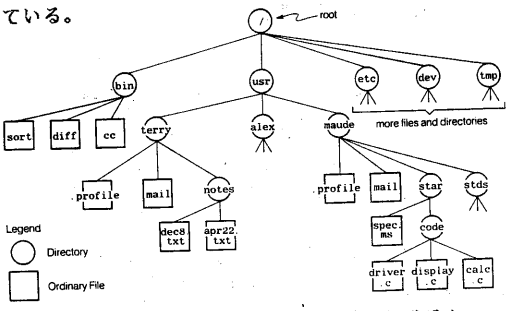


図4.1.階層的に構成されるファイルシステム

メソッドとしては、

rename　：　ファイル／ディレクトリ名の変更

set　　：　ファイル値の設定

mkdir　：　新しいディレクトリの作成

をとりあげる。また、操作が行われる位置は、必ずpathというパラメータで与えられるものとする。

以下に記述を示す。

```
Root → Dir
    Root.fssize = Dir.fssize

    :rename(path,newname) → Dir:rename
        Dir.path ← path
        Dir.newname ← newname

    :mkdir(path,dname)
        Dir.path ← path
        Dir.dname ← dname

    :set(path,fval)
        Dir.path ← path
        Dir.fval ← fval

Dir → DirEnt
    Dir.fssize = DirEnt.fssize

    :rename → DirEnt:rename
        DirEnt.path ← Dir.path
        DirEnt.newname ← Dir.newname

    :set → Dir:set
        DirEnt.path = Dir.path
        DirEnt.fval = Dir.fval

    :mkdir → DirEnt:mkdir
        DirEnt.path ← Dir.path
        DirEnt.dname ← DirEnt.dname

DirEnt → DirEnt File
    DirEnt1.fssize = DirEnt2.fssize + File.size

    :rename
        → DirEnt
            when DirEnt2.name == path
                DirEnt2.name ← newname

        → DirEnt:rename
                DirEnt2.path ← strip(path, DirEnt1.name)
                DirEnt2.newname ← DirEnt1.newname

    :set
        → File
            when DirEnt2.name == path
                File.fval ← DirEnt.fval

        → DirEnt:set
                DirEnt2.path ← strip(path, DirEnt1.name)
    :mkdir
        → ERROR
            when DirEnt.name == path

        → DirEnt:mkdir
                DirEnt2.path ← strip(path, DirEnt1.name)

DirEnt → DirEnt Dir
    DirEnt1.fssize = DirEnt2.fssize + Dir.fssize

    :rename
        → DirEnt
            when DirEnt2.name == path
                DirEnt2.name ← newname

        → DirEnt:rename
            when DirEnt2.name == top(path)
                DirEnt2.path ← strip(path, DirEnt1.name)
                DirEnt2.newname ← DirEnt1.newname

    :mkdir
        → ERROR
            when DirEnt.name == DirEnt.path
```

```
        → Dir:mkdir
            when DirEnt2.name <> path
                DirEnt2.path ← strip(path, DirEnt1.name)
                DirEnt2.dname ← DirEnt1.dname

        → DirEnt:mkdir
            DirEnt2.path ← strip(path, DirEnt1.name)
            DirEnt2.dname ← DirEnt1.dname

DirEnt → ε

    :mkdir
        substitute with DirEnt → DirEnt Dir, DirEnt → ε
            ...
File → ε
    File.size = sizeof(File.fval)
```

上の記述中で、path名を操作するために幾つかの補助関数が導入されている。strip(path,name)はpathの先頭からnameを取り除き、top（path)はpathの先頭から名前を一つだけ取り出す働きをしている。

5．考察

属性文法をいままで述べてきたように、汎用のシステム記述に用いようとする場合、解決しなければならない点が幾つか存在する。

(a) Interior Addressing

ユーザのコンテキストに対応する属性つき木の内部点をいかに効率よく指定するかという問題である。

ファイルシステムの記述では、pathで示される操作を行う場所がこの内部点に相当し、今回の記述では動的な計算木を構成する際にこのpathを解析することによって内部点を直接指定することを避けている。

しかしながら、このような状況は本報告で取り上げたファイル・システムにおけるワーキング・ディレクトリの他に、構造エディタにおけるカーソル位置、ソースコード管理システムにおける特定のバージョン、ＣＡＤシステムにおいては注目するエリア等、構造を持ったデータを扱うシステムにおいては一般的な概念であるといえ、効率的な内部点の指定は重要な課題であるといえよう。

(b) cfgの表現能力の限界

Underlaying Grammarがcfgであるという制限がもたらす木構造の制限がある。

この制限は、システムを記述する際に、設計者の意図するデータ構造と著しく異なった意味上の表現を強制する場合がある。例えば、あるノードから任意個の子ノードが生成されるといった状況、すなわち$X_0 \rightarrow X$のような文法規則に対応する状況は、頻繁に起こりうる。（属性文法型言語ＡＧには、繰り返しモジュール分割と呼ばれる、＊-ruleに相当する機能が導入され、＊-構造を持ったデータの処理において、記述の簡単化と意味上の整合を実現している。）

記述を行う際に、より柔軟な表現を許すためにも、＊-ruleに対する拡張を行う必要があるとおもわれる。

また、この計算モデルに基づいて記述されたシステムを実際に合成しようとするときの問題点として、

(c) 効率的な属性評価の方法

属性つき木の構造／属性の変更に関して、本モデルが参考としたRepsの構造エディタにおける編集モデルでは、属性伝搬を評価するためのアルゴリズムとして、time-optimalであるものと、space-efficientであるものが提案されているが、この編集モデルでは、木の変更が一度に一箇所でしか起こらないことが仮定されている。ところが、本モデルによれば、変更点は２つ以上同時に現れる場合があり、このような状況において上のアルゴリズム（またはその変形）が適用できることを確認する必要がある。

があげられる。

さらに、本計算モデル自体の課題として、

(d) オブジェクト指向の徹底化

オブジェクト指向プログラミングにおける重要な局面の一つに、クラス継承による重複記述の防止がある。システム記述において、その読解性を上げるためには重複した記述を減らすことが重要である。

また、オブジェクト指向プログラミングにおいて多用されるのが自分自身へのメッセージであり、この考えを適用することにより、本報告におけるpathの一致検査などは一箇所でまとめて記述できるようになるはずである。

(e) コンストレイント指向プログラミングとの関連

本計算モデルは、コンストレイント指向プログラミング[松田]と、とくに属性の伝搬の局面において深い関わりをもっていると考えられる。この関係を明らかに

し、モデルに組み込むことで、より読解性の高いコンパクトな記述が行えることが期待される。

があげられる。

6．まとめ

属性文法を一般のシステム記述と合成に応用するための計算モデルを提案し、その記述能力や実際のシステム記述／合成に用いる際の問題点を述べた。

本来の目的であったシステムの合成や、モデルの形式的定義についてはほとんど触れることができなかったが、属性文法による記述の利点を生かしながら、動的なシステムを記述するための基盤が提出された意義は大きいと思われる。

［参考文献］

[ICAD] ICAD System, Computer Aided Design Report, Vol5,No12,Dec.1985

[片山] 片山卓也：属性文法型計算モデル,情報処理, Vol24,No.2,pp147-155 (1983)

[knuth] Knuth.D.E.:Semantics of Context-Free Languages, Math.Syst.Th.,Vol2,No.2,pp.127-145 (1968)

[Reps] Reps,W.T.：Generating Language-Based Environments,The MIT-Press (1984)

[篠田・橘・片山] 篠田陽一,橘浩志,片山卓也：属性文法に基づいた関数型言語ＡＧのプログラム作成・実行支援システム,情報処理学会ソフトウェア基礎論／プログラミング言語研究会報告書 86-SF-16/86-PL-04 (1986)

[松田] 松田：属性伝搬によるコンストレイントの実現,情報処理学会ソフトウェア基礎論／プログラミング言語研究会報告書 86-SF-16/86-PL-04 (1986)