Vol. 35 No. 7　　情　報　処　理　　July 1994

解　説

属性文法とその応用—V

ソフトウェア環境への属性文法の応用†

今泉貴史††　篠田陽一†††

属性文法は，コンパイラの記述とその自動生成を目的とした研究が進められてきた．しかし，最近は属性文法をソフトウェア開発環境の分野に応用する試みもいくつか行われている．本稿では，属性文法のソフトウェア開発環境への応用について紹介する．

属性文法のもともとの目的であるコンパイラの自動生成も，広くとらえればソフトウェア開発環境の一部とみることが可能であり，何をもってソフトウェア開発環境への応用とするかは判断が難しい．本稿では，ソフトウェア開発環境を記述するシステムと，そのための技術に関して解説する．

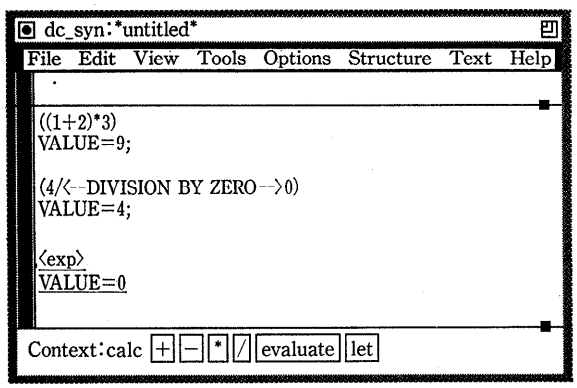


図-1　Synthesizer Generator による卓上計算機

1. 構造エディタ生成系

ソフトウェア開発環境への応用という点で忘れてはならないのが，構造エディタ生成系の **Synthesizer Generator** である[1)~4)]．Synthesizer Generator は米国 Cornell University の T. Reps と T. Teitelbaum らによって開発されたもので，属性文法に基づく記述言語を用いて構造エディタを生成するシステムである．

Synthesizer Generator は単なる構造エディタ生成系としてではなく，ソフトウェア開発環境を生成するツールとしてとらえることもできる．Synthesizer Generator が作成するエディタは，プログラムの作成だけでなく，コンパイラ，プログラムの静的なチェッカとしても使用できる．つまり，さまざまな情報をプログラマに与え，プログラマを助ける，ソフトウェア開発環境生成系である．

図-1 は，Synthesizer Generator に例題として附属する卓上計算機を実行した様子である．このプログラムはエディタであり，算術式の編集が可能となっている☆．また，エディタが扱う言語（この場合には式）の意味として，値を計算し出力する規則が指定されているため，計算機としても動作する．

構造エディタを生成する場合，エディタ生成系に編集対象である言語の構文を指定する必要があるが，通常は文脈自由文法を用いる．構文だけでなく言語の意味も指定することにより，高度な処理も可能となる．

Synthesizer Generator と同じく構造エディタ生成系である Gandalf システム[5)~8)]の ALOEGEN は，パーザ生成系 YACC[9)] と同様にアクションルーチンを用いて意味の記述を行う．これに対し Synthesizer Generator では，パーザ生成系 Rie[10)] と同様に属性文法を用いている．Rie では構文解析と同時に属性計算を行うため，対象とする属性文法のクラスが ECLR 属性文法[11),12)]となっている．しかし，Synthesizer Generator ではいったん

† On Application of Attribute Grammars to Software Environment by Takashi IMAIZUMI (Department of Computer Science, Tokyo Institute of Technology) and Yoichi SHINODA (School of Information Science, Japan Advanced Institute of Science and Technology).

†† 東京工業大学工学部情報工学科
††† 北陸先端科学技術大学院大学情報科学研究科

☆ 実際には，複数の式を編集するために，式の並びを編集することができる．

構文木を構築した後に属性計算を行うため，この制限はない．ただし，一般の属性文法すべてを受け付けるのではなく，属性評価器の速度の制限などから，ORDERED 属性文法[12),13)]を受け付けるようになっている（以前は well-defined 属性文法用の属性評価器も準備されていたが，最近のリリースでは準備されていない）．

先ほどの例題では，それぞれの「式」の意味としてその式のもつ値を計算し，Synthesizer Generator の出力構文（画面出力を決定するための構文．構文木が抽象構文木として不要な終端記号を排除して格納されているため，実際に出力を行う際にこれらを加えるためのもの）の中で計算した結果を表示するように指定している．そのため，すでに入力した式の一部を編集した場合にも属性計算により新しい式の値が計算され，新しく計算された値がただちに画面上に表示される．

プログラム1が，このエディタが扱う言語（算術式の並び）の構造や，その上での属性計算を指定しているものである．最初の root という宣言により，開始記号を calc と指定している．次に list の行は，calc という非終端記号がリスト形式のものであることを示している．ちなみに，Synthesizer Generator では構文規則の非終端記号をフィラと呼ぶ．次の数行が言語の構文規則を指定する部分で，

```
/* 抽象構文 */
root calc;
list calc ;
calc      : CalcPair(exp calc)
          | CalcNil()
          ;
exp: Null()
   | Sum, Diff, Prod, Quot(exp exp)
   | Const(INT)
   ;

/* 式の評価のための属性意味規則 */
exp {synthesized INT v;} ;
exp:  Null{ exp.v=0; }
   |  Sum{ exp$1.v=exp$2.v+exp$3.v; }
   |  Diff{ exp$1.v=exp$2.v-exp$3.v; }
   |  Prod{ exp$1.v=exp$2.v*exp$3.v; }
   |  Quot{ local STR error;
         error  = (exp$3.v == 0) ?
                  "<--DIVISION BY ZERO-->": "";
         exp$1.v= (exp$3.v == 0) ?
                  exp$2.v: (exp$2.v / exp$3.v);
         }
   |  Const{ exp$1.v=INT; }
   ;
```

プログラム 1　卓上計算機のソースファイル（抽象構文・意味規則）

表-1　卓上計算機の文脈自由文法

生成規則	ラベル（オペレータ）
calc → exp calc	CalcPair
calc → ∈	CalcNil
exp → ∈	Null
exp → exp exp	Sum
exp → exp exp	Diff
exp → exp exp	Prod
exp → exp exp	Quot
exp → INT	Const

これを文脈自由文法で表すと**表-1**となる．一番右の欄はそれぞれの構文規則に付けられたラベルで，Synthesizer Generator ではオペレータと呼ぶ．

次に書かれている部分が属性の定義と意味規則を記述した部分である．exp {synthesized INT v ;} ; という行は，exp というフィラに対して合成属性として INT（整数）型の属性 v を宣言している．この属性はそれぞれのフィラが表す式の値を保持するためのものであり，加算（Sum）の場合は右辺の exp の v の値を加えたもの*，定数（Const）の場合は右辺の整数そのものの値などと記述されている．

意味規則の中で特徴的なのが除算（Quot）の部分である．ここでは左辺の exp の属性 v の値を計算する場合に，除数の値が 0 であるかどうかを検査している**．もしこの値が 0 であれば，除算を行わずに被除数の値をそのまま式の値としている．その直前では，error という局所変数***を宣言し，除数が 0 の場合には "<--DIVISION BY ZERO-->" という文字列を割り当てている．この文字列は，エディタの画面上に 0 による除算に対するメッセージを出力するために，次の出力構文の中で用いられる．

プログラム 2の前半が，構文木をエディタ画面上に表示するための出力構文の定義である．出力構文は，フィラとオペレータを指定した後，角括弧でくくって実際の出力方法を指定する．CalcPair オペレータの場合，右辺の最初のフィラを出力した後，改行して "VALUE=" という文字列を出力し，その後に exp.v の値（CalcPair がもつフィラ exp の属性 v の値）を表示する．これによ

* exp$2 などの記法は，注目している構文規則の中に同じフィラが複数存在する場合にそれを区別するためのもので，exp$2 は 2 番目の exp（つまり右辺の最初の exp）を表している．

** ?: は条件演算子である．

*** オペレータに対して付けられる属性．計算の一時的な結果を保持するためなどに用いられる．

```
/* 出力構文 */
calc: Calcpair[ @ ::=
        @ "%nVALUE=" exp.v [";%n%n"] @];
exp: Null[ @ ::= "⟨exp⟩" ]
   | Sum[ @ ::= "(" @ " + " @ ")" ]
   | Diff[ @ ::= "(" @ " − " @ ")" ]
   | Prod[ @ ::= "(" @ " * " @ ")" ]
   | Quot[ @ ::= "(" @ " / " error @ ")" ]
   | Const[ @ ::= ^ ]
   ;

/* トランスフォームコマンド */
transform exp on "+" ⟨exp⟩ : Sum (⟨exp⟩, ⟨exp⟩),
              on "−" ⟨exp⟩ : Diff (⟨exp⟩, ⟨exp⟩),
              on "*" ⟨exp⟩ : Prod (⟨exp⟩, ⟨exp⟩),
              on "/" ⟨exp⟩ : Quot (⟨exp⟩, ⟨exp⟩)
              ;
```

プログラム 2 卓上計算機のソースファイル（出力構文・トランスフォームコマンド）

り，式を入力した場合にはその値が出力されることになる．その後，セミコロンと改行二つを出力して右辺の2番目のフィラ (calc) の出力を行うことを示している．セミコロンと改行二つの周りには角括弧が付けられているが，これはリスト形式のフィラにのみ許されている記法で，この部分はリストの終端でない場合にのみ出力される．また，出力指定の最初の::=は，左辺のフィラの出力を行うかどうかや編集可能かどうかなどを示している．

Quot の出力指定では，先ほど局所属性として指定した error の値を出力している．意味規則の部分で error に空文字列以外のものが割り当てられていた場合，この規則によりそれが画面上に出力される．また最後のトランスフォームコマンドは，構造エディタにみられる構造的な変更を支援するためのもので，“+”，“−”，“*”，“/”というコマンドにより，exp というフィラをそれぞれ加算，減算，乗算，除算に変更することを指定している．この例では用いていないが，現在のフィラの状態を用いて変換後の状態を指定することができるため，かなり複雑な変換も可能である．また，属性値を用いることも可能であり，この場合には文脈に依存した変換も可能となる．

エディタの仕様には，これ以外にも入力テキストをパーズするための入力構文，パーズした結果を抽象構文木と対応づけるためのエントリ宣言，意味規則中で用いることが可能な関数の定義などを指定する．

2. インクリメンタル属性評価器

Synthesizer Generator の作成するエディタにおいて，抽象構文木の上で属性計算を行う場合には，構文木が変更された場合の属性の再計算が問題になる．これについて，まずどのように変更が行われるのかを解説し，その後変更が起こった場合の属性の計算方法について解説する．

構造エディタの中には，構文木の変更として構造的な変更だけを許しているものもあるが，Synthesizer Generator ではより柔軟な編集モデルを提供している．すでに述べたように，構造的な変更はトランスフォームコマンドとして指定する．これに加えて，入力構文と呼ばれる構文を指定することによりエディタのユーザはテキスト形式での変更を行うことが可能となっている．この場合，編集可能な領域はセレクションと呼ばれる部分木に限られている．このセレクションは，構文木の上を移動するカーソルのようなもので，常にユーザが注目している部分木を示す．

ユーザがテキスト形式の編集を行う場合，セレクションを示す部分木が自動的に構文木から切り離される (Cut)．編集を終了し編集結果を登録する際には，その部分をパーズした結果の部分木が，もとの構文木に継ぎ木される (Paste)．このように Synthesizer Generator では，ユーザによる編集は Cut & Paste という編集モデルに基づくことになる(図-2)．ちなみに，構造的な変更を行うトランスフォームコマンドの場合にも，現在のセレクションを異なる部分木に変更する処理となる

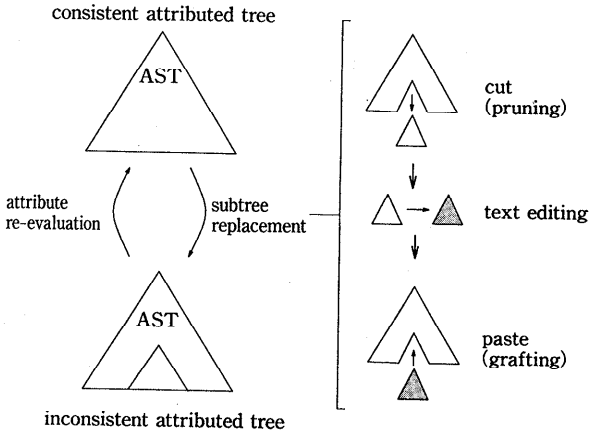


図-2 Cut & Paste 編集モデル

ため，同様な Cut & Paste 編集モデルとみることができる．

このように部分木の置換が行われた際，一般には上下の木の間で属性の値に矛盾が生じるため，属性の再計算が必要になる．このときほんの些細な変更に対しても構文木全体のすべての属性を再計算するのはコストも高く実用的とはいえない．変更の影響が局所的なものであれば，構文木の一部の属性値を再計算するだけで全体を一貫した状態に戻すことが可能である．このために，インクリメンタルな属性評価アルゴリズムが広く研究されている．インクリメンタル属性評価とは，部分木の置換が行われた際に再計算が必要な属性のみの値を計算して，値が変わらない属性に関しては処理を行わないことで属性の再計算にかかる時間を最小にしようとするものである．

Synthesizer Generator の属性評価器もこのインクリメンタルな評価器である[3),14)]．Synthesizer Generator のように，比較的大きな構文木に対して変更を加えるようなアプリケーションを考える場合，属性の再計算にかかるコストは無視できない．インクリメンタルな属性評価アルゴリズムの計算量を議論するときに用いられる尺度として *AFFECTED* がある．この *AFFECTED* とは，「置換により値が変更される属性インスタンスの集合」を示す．Reps の ORDERED 属性文法に対するアルゴリズムでは，$O(|AFFECTED|)$ で属性の再評価が可能となっている．インクリメンタルな属性評価器に関する研究は，たとえば well-defined 文法に対する評価器[3),15)] などこれ以外にも多くなされているが，インクリメンタルな属性評価アルゴリズムでは，どちらの木構造も正しく属性付けされていなければならないという制約がある．そのため，部分木の継ぎ木を行う前の段階で属性値を求める方法に関しても研究が行われている[16)]．

3. 非同期多値変更

Synthesizer Generator では属性評価アルゴリズムの制限から継ぎ木は1カ所でしか許されなかった．しかも，継ぎ木が行われてから属性計算が終了するまで次の処理は行えない．一つの構文木を複数のユーザで変更する場合や，属性の再計算に時間がかかる場合には，これらの点が問題となる．Mercury はこれらの点を解決することを主眼において G. E. Kaiser らにより開発されたシステムで，非同期多値変更を許すシステムである[17)]．この Mercury は Synthesizer Generator に対する拡張という形で実現されている．Synthesizer Generator の属性評価器部分は他の部分と独立して作られているためこのようなことが可能になっているのだが，そのため Mercury は Synthesizer Generator と同様なユーザインタフェースをもつ☆．Mercury で採用している属性文法のクラスは，Synthesizer Generator とは異なり well-defined となっている．

非同期多値変更について述べる前に，まず同期多値変更について簡単に説明しておく．同期多値変更とは，インクリメンタル属性計算において，継ぎ木が起こる場所が一カ所でなく，構文木の数カ所で継ぎ木が起こることを許すものである．これにより，一つの構文木を数人で編集することが可能になる．ただし，属性計算中は継ぎ木は許されず，その意味で継ぎ木は同期的である．もし複数箇所で継ぎ木が行われた場合には，一カ所で行われた変更による属性値の再計算が他の部分の属性値に依存している可能性もあるため，先ほどのインクリメンタルアルゴリズムをそのまま適用することはできない．そこで多値変更を許すアルゴリズムが必要となるのである．

非同期多値変更とは，このような複数箇所での変更が以前の変更による属性の再計算の最中に起こることを許すものである[18)]．変更の結果としての属性の再計算がすぐに終了する場合には問題とならないが，時間のかかる計算の場合には，同じ構文木を操作しているすべてのユーザはその計算が終了するのを待たなければならなくなってしまう．非同期多値変更を許すアルゴリズムでは，1カ所での変更による属性の再計算が終了する前に他の場所で変更することを許している．

非同期の変更を許す場合，インクリメンタル属性評価器で計算量の目安としていた *AFFECTED* の考え方も考え直さなければならない．Mercury では，すべての変更を順番にインクリメンタル属性評価器で評価した場合に，それぞれの変更に対して再計算が必要な属性すべてを合計

☆ Mercury でベースとしているのは Cornell 大学で開発されたバージョンであり，本稿で例に用いているものとは少し異なる．

したものを $ASYNC\text{-}AFFECTED_{SEQ}$ と呼び，これらの変更を完全に並列に処理したときに再計算が必要な属性の集合を $ASYNC\text{-}AFFECTED_{SIM}$ と呼んでいる．MERCURY の属性評価器では，$ASYNC\text{-}AFFECTED_{SIM} \leq ASYNC\text{-}AFFECTED \leq ASYNC\text{-}AFFECTED_{SEQ}$ となる $ASYNC\text{-}AFFECTED$ で評価を行うことが可能となっている．

これをソフトウェア開発環境に適用する例を考えてみる．大きな一つのプログラムを数人から構成されるプロジェクトチームで開発しており，それぞれのユーザの画面には，自分が担当しているプログラムの部分が表示されている．このプログラムはもちろん抽象構文木として管理され，その上の属性計算として変数や関数の定義と使用の関係の検査などが定義されているとする．このとき，ライブラリを担当しているプログラマがライブラリの仕様（入出力）を変更した場合を考える．他のユーザはこの変更とは関係なくプログラミング（つまり構文木の変更）を続けられるが，いつかはライブラリが変更されたことが属性の値として伝えられ，変更しなければならない部分に印が付けられることになる．そこでプログラマはライブラリの仕様が変更されたことを知り，正しく編集をすることができるのである．つまり，一人の変更により他のユーザの作業が中断されることはなくなるのである．しかし，この非同期多値変更に関してはモデルとしての完全な意味が与えられているわけではなく，研究の余地がある．

4. ソフトウェアデータベース

MERCURY は Synthesizer Generator のもつインクリメンタル属性評価器に対して非同期の多値変更を許す形で拡張を加えたものであったが，次に紹介する MAGE[15),19)] はまったく異なる概念で計算モデルを拡張することによりソフトウェア環境の記述を行おうとするシステムである．

MAGE の一つの応用例として，ソフトウェアデータベースがある．ソフトウェアデータベースでは，それぞれのデータを関連づけて保存し，データの一部が変更された場合には必要に応じて変更を伝播しデータベース全体を一貫した状態に保つ．インクリメンタル属性評価器の再計算のプロセスをこの変更伝播に用いることができる．

MAGE では，変更伝播だけでなく構造を変更する機構を提供しているため，容易にソフトウェアデータベースの記述に用いることが可能である．

MAGE の採用する計算モデル OOAG (Object Oriented Attribute Grammar) では，属性文法をオブジェクト指向の概念を用いて拡張することにより，メッセージパッシングの機構を用いて一時的な計算や構文木の変更を行うことが可能となっている．属性文法としてみた場合，MAGE の属性文法としてのクラスは well-defined となっている．

MAGE での属性計算 (MAGE の言葉ではオブジェクトの評価) は，

1. 静的計算
2. メッセージパッシング
3. 動的計算
4. 木の置換

という4つのフェーズを繰り返すことで行われる (**図-3**)．

最初の静的計算フェーズは，静的仕様と呼ばれる通常の属性文法と同様な記述に基づき属性の値を計算するもので，このフェーズだけを見れば通常の属性文法となんら変わりはない．

次のメッセージパッシングのフェーズは，ユーザや構文木の一部からメッセージが発せられた場

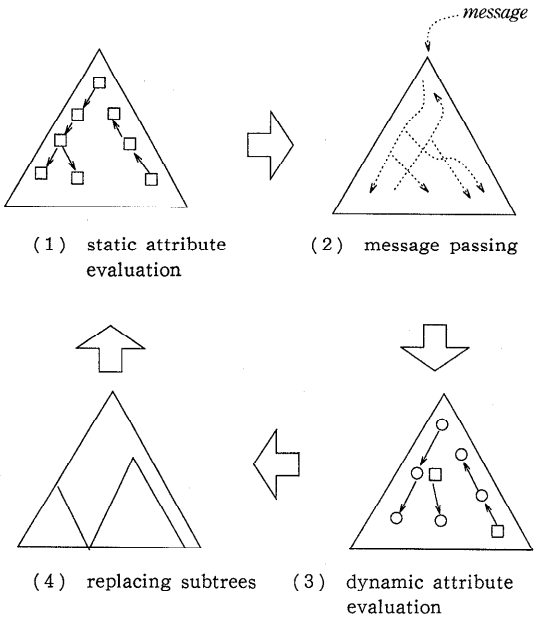


図-3 MAGE の評価ループ

合に，そのメッセージを構文木の関係する部分に渡していくためのフェーズである．メッセージのやりとりは，自分の親か子に対してのみに限られており，属性文法での属性の受渡しとよく似ている．しかしこのメッセージにより渡されるものは値ではなく，あらかじめ決められた意味規則（動的意味規則と呼ばれる）が構文木の上に張り付けられていく．メッセージには属性（動的属性と呼ばれる）も付けられており，張り付けられる意味規則は，この一時的に計算される動的属性の値を定義するものである．ここで張り付けられた意味規則は，次の動的計算フェーズで有効となる．

動的計算フェーズでは，メッセージにより張り付けられた意味規則により，動的属性の値を計算する☆．たとえば木構造の根に対して動的合成属性をもつようなメッセージを送ることは，その木構造に対する問合せ処理を表すことになる．

実は動的計算フェーズでは動的属性の値のほかに置換用の木の値も計算しておくことが可能である．最後の木の置換フェーズでは，計算された木構造を構文木の一部と置換することにより，構文木の構造を変更することができるようになっている．このようにして変更された構文木では継ぎ木が行われた点で属性の値に矛盾が生じるため，再度静的計算フェーズを行うことにより属性値を矛盾のない状態に再計算する．このようなループを繰り返すことにより，MAGE の評価は進められる．

MAGE のもつ特徴の一つは，動的属性計算と木の置換の機構を用いて構文木の構造を変更することができる点であろう．Synthesizer Generator などでは，現在のセレクションに対応する部分木をテキスト編集やトランスフォームコマンドという形で変更することを許していた．言い換えると，現在注目しているセレクションに対応する部分木だけを変更することができた．これは基本的には Cut & Paste モデルに基づくものである．しかし MAGE の場合には，これに加えて，メッセージや動的仕様記述に記述さえされていれば，現在の注目点とは関係なく，構文木のどの位置に対しても変更を行うことが可能となっている．この機構は，データベースなどを構築する際に特に有効となる．データベースにデータを登録する場合には，データの内容によりデータベース内部の構造に基づいて実際の登録が行われる（つまり構文木が変更される）位置が決定される．Synthesizer Generator の場合には変更する位置を明示的に指定する必要があったが，MAGE ではその必要がない．

☆ 実際には，メッセージパッシングのフェーズと動的評価のフェーズはインタリーブされる．

5. 展 望

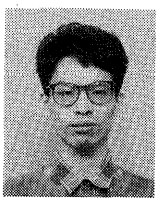
属性文法は対象とする言語の意味を記述するための形式的手法としてとらえられてきた．これまで述べてきたように，インクリメンタルな属性評価器と Cut & Paste やメッセージパッシングの機構を用いることでシステムの動的な側面の記述も可能となっている．ソフトウェア開発環境にみられるさまざまなオブジェクトを静的な木構造としてとらえることに異論を唱える向きもあるが，この点に関しても高階属性文法[16),20)]，MAGE にみられる Named Object, DR-Thread[21)] などさまざまな研究が行われている．属性文法には記述の局所性やそれにともなう読解性という特徴があり，これらを生かすことでソフトウェア開発環境のような比較的大規模なソフトウェアを記述することも可能になったと言えるであろう．

参 考 文 献

1) GrammaTech, Inc. One Hopkins Place, Ithaca, NY 14850, USA: *The Synthesizer Generator Reference Manual*, 1993. Release 4.1.
2) Reps, T. W. and Teitelbaum, T.: *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer-Verlag (1989).
3) Reps, T. W.: *Generating Language-Based Environments*, The MIT Press, Cambridge, Massachusetts 02142, London, England (1984).
4) 渡辺喜道，今泉貴史，徳田雄洋：構造エディタ生成系，情報処理，Vol. 32, No. 3, pp. 282-294, (Mar. 1991).
5) Habermann, A. N. and Notkin, D.: Gandalf: Software Development Environments, *IEEE Transactions on Software Engineering*, Vol. 12, pp. 1117-1127 (Dec. 1986).
6) Notkin, D.: The GANDALF Project, *The Journal of Systems and Software*, Vol. 5, No. 2 (May 1985).
7) Habermann, A. N. et al.: The Second Compendium of Gandalf Documentation, Technical report, Department of Computer Science, Carnegie

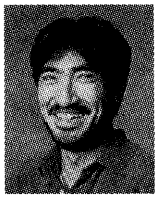
-Mellon University (May 1982).
8) Staudt, J., Barbara, K., Charles, W., Habermann, A. N. and Ambriola, V.: *The GANDALF System Reference Manuals*, Department of Computer Science, Carnegie-Mellon University (May 1986).
9) Johnson, S.: YACC: Yet Another Compiler Compiler, Technical Report 32, Computing Science (1975).
10) 佐々政孝, 石塚浩志, 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 (1993).
11) Sassa, M., Ishizuka, H. and Nakata, I.: ECLR-Attributed Grammars: A Practical Class of LR-Attributed Grammars, *Information Processing Letters*, Vol. 24, No. 1, pp. 31-41 (Jan. 1987).
12) 佐々政孝: 属性文法, コンピュータソフトウェア, Vol. 3, No. 4, pp. 73-91 (377-395), October 1986. チュートリアル.
13) Kastens, U.: Ordered Attribute Grammars, *Acta Informatica*, Vol. 13, pp. 229-256 (1980).
14) Reps, T. W. and Teitelbaum, T.: *Incremental Attribute Evaluation for Ordered Attribute Grammars*, chapter 12, pp. 246-277, In Gries 2 (1989).
15) Shinoda, Y. and Katayama, T.: Object Oriented Extension of Attribute Grammars and its Implementation Using Distributed Attribute Evaluation Algorithm, In *Proceedings of the International Workshop on Attribute Grammars and their Applications*, pp. 177-191, Springer-Verlag, 1990. Lecture Notes in Computer Science 461.
16) Teitelbaum, T. and Chapman, R.: Higher-Order Attribute Grammars and Editing Environments, In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 197-208, ACM, June 1990. White Plains, New York.
17) Micallef, J. and Kaiser, G. E.: Support Algorithms for Incremental Attribute Evaluation of Asynchronous Subtree Replacements, *IEEE Transactions on Software Engineering*, Vol. 19, No. 3, pp. 231-252 (March 1993).
18) Micallef, J.: Incremental Attribute Evaluation for Multi-User Semantic-Based Editors, Ph. D. dissertation CUCS-023-91, Columbia University, Department of Computer Science, New York, NY 10027, USA (May 1991).
19) Gondow, K., Imaizumi, T., Shinoda, Y. and Katayama, T.: Change Management and Consistency Maintenance in Software Development Environments Using Object Oriented Attribute Grammars, In Shojiro Nishio and Akinori Yonezawa, editors, *Object Technologies for Advanced Software* (*Proceedings of the First JSSST International Symposium*), pp. 77-94. JSSST, Springer-Verlag, November 1993. Lecture Notes in Computer Science 742.
20) Vogt, H. H., Swierstra, S. D. and Kuiper, M. F.: Higher Order Attribute Grammars, In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 131-145. ACM, June 1989. Portland, Oregon.
21) Vorthmann, S. A.: Coordinated Incremental Attribute Evaluation on a dr-Threaded Tree, In *Proceedings of the International Workshop on Attribute Grammars and their Applications*, pp. 207-221, Springer-Verlag, 1990. Lecture Notes in Computer Science 461.

(平成 6 年 4 月 28 日受付)



今泉 貴史 (正会員)

1965 年生. 1987 年東京工業大学工学部情報工学科卒業. 1989 年同大学院理工学研究科修士課程修了. 1992 年同博士後期課程修了. 同年同工学部情報工学科助手, 博士(工学). この間, 属性文法に基づく言語のプログラミング環境の開発に携わる. 属性文法を用いたアプリケーションの開発などに興味を持つ. 日本ソフトウェア科学会会員.



篠田 陽一

1959 年生. 1983 年東京工業大学工学部情報工学科卒業. 1985 年同大学院理工学研究科修士課程修了. 1988 年同博士後期課程単位取得退学. 同年同工学部情報工学科助手. 1991 年北陸先端科学技術大学院大学助教授. 工学博士. この間, 情報環境, ソフトウェア工学, ソフトウェア開発環境などの研究に従事.