属性文法形式によるセマンティックス記述の問題点

松田裕幸　日本電気技術情報システム開発（株）

概要

本稿では属性変換文法、表示的意味論によるセマンティックス記述方式について簡単に解説を行ない、表示的意味論を属性文法の枠組みの中でとらえた属性文法形式において、表示として接続を許した場合の問題点について議論する。最後に、LEAG(Extended Attribute Grammar Based Language) による接続意味論の記述例を紹介する。

０．　はじめに　－セマンティックスフォーマリズムへの期待－

新たにプログラミング言語やインタフェース言語あるいはデザイン言語などを作成するにあたって、その言語仕様をどのように記述するかは設計者の任意であり、例えばシンタックスについては具体例を示すことによって、またセマンティックスは自然言語等によって与えることができる。そしてその仕様にもとづいて処理系作成者は仕様の内容を理解し、設計者は再設計の検討材料にする。しかし、再設計に対して修正が容易であり、また設計者自身が設計内容を整理しておきたい場合に、あるいは処理系作成者が仕様上のあいまいさに頭を悩まさないためにも、仕様をフォーマルに記述したい要求も当然起きてくる。（一方、利用者にとってのフォーマリティは一般に指摘されるほど重要ではないかもしれない。事実、フォーマルな仕様を見せられるよりは実際に使ってみて何気なく理解しているという場合が多く、かつその方が理解への早道になっていることもありうる。）

歴史的にみると、シンタックスのフォーマリズムが先行し、セマンティックスの研究はシンタックスの後を追いかけている。特に、シンタックスの場合は、バッカスのＢＮＦ[1] に始まり、Knuth が属性文法を提案し[2] 、さらに文脈自由文法の様々なサブクラスとそれらを入力とするパーザ自動生成機の研究[3] が進むことによって安定した理論を作り上げている。他方、セマンティックスに関しては、文脈自由文法や属性文法を変換（シンタックス要素をコードに変換）によって拡張した文脈自由変換文法[4] 、属性変換文法[5,6] があり、また、言語要素を数学的対象物としそのタイプ領域を自己適応が可能になるまで拡大したラムダカルキュラスによってセマンティックスを与える表示的意味論(Denotational Semantics)[7,8] が存在する。特に、その接続(continuation)の部分に焦点をあて、プログラムの制御に関するセマンティックスを記述する接続意味論(Continuation Semantics)[8] をあげることができる。この他、プログラムの実行の前提条件と後件条件の記述によってセマンティックスを規定する公理的意味論(Axiomatic Semantics)[9]の世界もある。しかし、いずれにしてもセマンティックスの研究分野ではシンタックスのように形式性と実用双方からみて有用な理論はまだ確立されていない。

では、セマンティックスに対するフォーマリズムは現状では無意味なのだろうか？そうではないと思う。かつて、フォーマルセマンティックスの研究はコンパイラの検証やプログラムの正当性の証明等に実際に役立つと考えられていた。しかし、現在ではそのような非現実的な希望はほとんど消え去り、むしろ言語設計者がセマンティックスを設計する際のフォーマリズムあるいは道具を提供することに存在価値を見いだすことができる。

現在、非常に多くのプログラミング言語が存在し、それらが持つ機能は多種多様である。Fortran 、Pascal、C などの言語の特徴はよく知られているが、この他にも、CLU での抽象データ型[10]、オブジェクト指向言語SmalltalK[11] 、LOOPS[12] 、ABCL[13,14] などにみられるメッセージパッシング、インヘリタンス機構、Prolog[15]でのユニフィケーション、バックトラック機構、関数型言語ML[16,17] でのタイプ理論にもとづくタイプインファレンス機構等々、一般にはまだよく知られていないが重要な機構が存在する。（ABCLでは通常の意味でのインヘリタンス機構は含んでいない。）そしてこれら多岐にわたる言語上の特徴に対してフォーマルなセマンティックスを与える研究はわずかしか行なわれていない（例えば、メッセージパッシングの理論[18]，インヘリタンス理論[19]など）。

筆者はすでにPascalのフルセットに対して属性変換文法を用いることでスタックマシン用コードを生成する言語仕様を記述している[20]。

さらに、属性文法を実行可能な仕様記述言語として拡張したLEAG[21,22] によってオブジェクト指向言語ABCL逐次版のコンパイラを作成した[22]。ABCLにはこれまでセマンティックスの研究の対象とされてきたPa

scalやC 、PL/Iなどにはみられない特徴－オブジェクト階層、メッセージパッシング、並列性等－がある。LEAGにおいては、これらの機構に対するセマンティックスはLispのコードで具体的に記述された。例えば、オブジェクトはクロジャーで表現されており、コード部の記述はそれを反映している。よって、オブジェクトの内部形式を理解するにはクロジャーの表現形式とそれへの操作関数を知らなくてはならない。オブジェクトへのメッセージもクロジャーへの操作関数の一部として理解される、等々。

ABCLの言語仕様が変化し、修正され、拡張されていくにつれて、コード部の記述への修正、追加がひんぱんに繰り返された。幸いにして、属性文法の特徴であるモジュール性と、LEAGのパターンマッチング機構、コードの属性への埋め込み機構とによって、大部分の対応は容易であったが、いくつかのルールに分散するコードからあるまとまったセマンティックスを抽出できるだけの透明さは少しずつ失われていった。

LEAGはABCLのような実験的言語を短期間でかつ多くの修正、変更を含む環境の中で作成するには有効であったが、上記の例にもみられるように、属性文法が持つ宣言性と表示的意味論が持つ抽象性のよさが部分的にしか活かされていない。そこで、現在、以下の３つの項目に対してLEAGの再検討を始めている：

1) タイプ属性の導入とタイプ操作関数の宣言機構。（例えば、メッセージ、オブジェクトなどをタイプとして扱い、メッセージの操作はメッセージタイプに対する操作関数によって表現する。）
2) プログラムを（Ｓ式で表現できる）ゆるく構造化された内部形式に変換するインタフェース。（LEAGは基本的にはＳ式上での解析とセマンティックスを記述できる。よって、ここで考えている内部形式は従来の構文解析が生成する構文解析木とは異なり、プログラムそのものがBEGIN,END,OBJECT等のキーワードによって暗黙に表現している構造をそのままＳ式に変換したものである。）
3) コード部での抽象的表現の記述能力。（例えば、バックトラック、メッセージパッシング、インヘリタンス、並列機構等の制御構造を形式的に記述したい。）

そして、本稿は3)にあげた内容の検討過程から生まれたものである。

１．　属性文法のセマンティックス表現への拡張

本章では属性文法を簡単に説明した後、属性文法を変換（構文要素をコードに変換）によってセマンティックスを表現する属性変換文法[5,6] と、LEAG[21,22] におけるセマンティックス記述形式について解説する。

1.1　属性変換文法

属性文法は文脈自由文法を属性によって拡張したものである。属性はプログラム中の宣言、手続き等から得られるタイプ、変数の束縛環境などの情報を保持し、タイプチェック、中間コード生成に利用される。属性文法によるプログラムの解析途中のイメージを図1-1に示す。

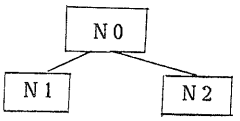
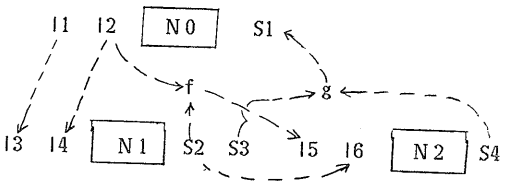


図1-1(a)　ルールと対応する解析木




```
<N0 ↓I1  ↓I2  ↑S1> ::=
    <N1 ↓I3  ↓I4  ↑S2  ↑S3>
    <N2 ↓I5  ↓I6  ↑S4>
    where I3 = I1, I4 = I2, I5 = f(I2, S2),
          I6 = S2, S1 = g(S3, S4)
```

図1-1(b)　属性文法の例と対応す解析木

図1-1(a)の左の図には解析木のノード(N0,N1,N2)とその依存関係を実線で表わし、右には対応するBNF の表記を示す。一方、図1-1(b)の上は、この解析木の各ノードに属性を付加したもので点線は属性の依存関係を示している。Ｉで代表される記号は相続属性(Inherited Attribute) を、Ｓで代表される記号は合成属性(Synthesized Attribute) を表わす。相続属性はこのノードに対する入力情報を保持し、合成属性は出力情報を保持する。例えば、ノードN1の相続属性I3はノードN0の相続属性I1の内容を受け継ぐ。また、ノードN2

の相続属性I5は、ノードN0のI2とノードN1の合成属性S2を属性評価関数fによって評価した結果に依存する。この関係をBNF の拡張形式で表現したものが図1-1(b)の下の式である。

ここで相続属性は↓の、合成属性は↑のプレフィックスによって示される。また、where 以下では、属性間の関係を表わしている。

ここではこれ以上、属性文法の説明は行なわないので、属性文法の一般的な定義については文献[4] を、属性文法記述からコンパイラ自動生成に関する研究に関しては文献[6,25,26,27,28] を、また、属性文法のクラス決定問題については文献[29,30] を、さらに、属性の評価式にまつわる順位、コスト、循環等の問題に対しては文献[30,31,32,33] を参考にされたい。

属性文法はすでにみてきたように、一般にはシンタックスの世界を対象としている。すなわち、属性の値としてはタイプなど本来シンタックスのカテゴリに属するものが用いられる。

一方、実際の処理系を作成するためにはコード部の記述が必要であり、コード部の記述に対応するものを属性文法の中に埋め込み可能にしたものが属性変換文法[42]である。図1-2 に属性変換文法の例を示す。

```
<S> ::= <E  ↑P> {ANSWER  ↓R}
    where R = P
<E  ↑P>  ::=  "+" <E1 ↑Q>  <E2 ↑R>
              {ADD  ↓A1  ↓A2  ↑R0>
    where A1 = Q, A2 = R, P = R0.
```

図1-2 属性変換文法の例

これは、前置形式で表現された加算プログラムに対するシンタックスとセマンティックス（コード部の記述）を記述している。{} で囲まれた部分がアクション部（コード生成もしくは実行）を表わし、属性はその引数である。アクションのタイミングは {} の位置によって決まり、完全に文脈に依存している。例えば、ADD 以下のコードは２つの式E1とE2の評価後に出力される。この文法をプログラムの解析とコード生成に活用するためには、一般にはプッシュダウンオートマトンを拡大する。すなわち、各非終端記号は属性用のフィールドを持ち、アクションシンボル(ANSWER,ADD 等)も非終端記号として扱われる。属性変換文法においてセマンティックスは、コード生成部の記述を属性文法の枠のなかに取り込みその解釈をオートマトンによって実現することでフォーマルに与えられる[4,6] 。

そして、セマンティックスの抽象性はコードのマクロ表現、今の例では、ANSWER, ADD などによって得られるが、実際には各種レジスタ割りつけ情報なども属性として与えなければならず[42]、特に、多種にわたる制御構造、引数呼び出し形式等を考えると、実用レベルの言語においては高い抽象性は得られにくい[22] 。

1.2　LEAGにおけるセマンティックス記述

(1) LEAGの概要と開発の動機

LEAGは属性文法の関数的解釈にもとづいており、コードを属性として埋め込むことが可能であり、最終的に生成されるコードは（連続した）属性間の評価関数の値として得られる。

LEAGはLisp-like な記法を持ち、属性は関数の引数と結果で受け渡される。このため、LEAGはパターンマッチングの機能と多値の機能を持つ。LEAGで記述されたものは、Lispの環境のもとでは、そのままインタプリタ、コンパイラとして実行可能であり、LEAGは言語のプロトタイプ[21,22] を作成する際に、言語仕様を記述し、その効果を実験するための強力なツールとなりうる。

Pascalのようにそのシンタックス、セマンティックスがよく知られている言語においては、言語仕様記述のフェーズと処理系作成のフェーズを分離することは言語設計者にとって容易であるが、ABCLのような実験的言語を作成する際には、事前にシンタックス、セマンティックスが明瞭になっているわけではなく、実際に動作するーすなわち、処理系が存在するー環境からのフィードバックによって言語仕様を固めていかなければならない。

このような状況のもとでは、仕様記述から処理系生成までの開発パスが短く、処理系使用から仕様記述へのフィードバックが容易となるプロトタイピング型の言語設計開発形態が望ましい。さらに言えば、言語仕様がそのままプログラム（コンパイラ、インタプリタ）として動作すればなお望ましい。LEAGはこうして方向をもって設計、開発された。

(2) LEAGにおけるセマンティックスの記述例

while 文に対するLEAGのルール記述を図1-3 に示す。（ただし、これは説明のために用意した例であり、実際にはこのように単なる言い換えにすぎない記述はほとんど存在しない。また、形式はすでに作成済みのもの[22]とは多少異なる。旧版の不備を修正して現在改

良を行なっており、それに対応している。)

```
(rule command (ENV)
  (select-on-syntax
    while   (while-statement ENV)
    if      (if-statement ENV)
    ... )
  (select-on-attribute
    (is-func <id> ENV)  (function ENV)
    (is-proc <id> ENV)  (procedure ENV)
    ...)
  )

(rule while-statement (ENV)                          -1
  (syntax "while <condition> do <command>")          -2
  (let  ((bool COND) (condition ENV))                -3
        ((TYPE BODY) (command ENV))                  -4
        (CODE (code                                  -5
               "(prog ()
                   loop:
                   (cond (COND
                          BODY
                          (go loop:))))"))
    in (CODE ENV) ))                                 -6
```

この例を完全に説明することは本節の目的の範囲を越えるので、詳細は現在作成中のLEAGに関する論文にゆずる。

ruleによってルールをひとつ定義する(1)。ルールは関数に対応する。ルールwhile-statement の入力属性はENV (1)、出力属性はENV とCODEである (6)。このルールに対応するシンタックスはsyntaxによって指定する(2)。< > はシンタックス要素であり、同時に< > 内の名前はそのまま本ルールを呼び出すルール名condition 、command に対応している。ルールcommand の入力属性はENV , 出力属性はbool、CONDであり(3)、boolは属性タイプ上の制約を表わす。CONDは属性変数である。コードCODEはcodeによって与えられる(5)。"" で囲んだものはLISPのプログラムであるが、COND、BODYの部分はすでに得られた内容が展開される。

属性変換文法ではコード部のタイミングはアクション部 { } によって制約されたが、LEAGでは各ルールを関数としてとらえ、コードを出力の一部としているため、ルール間でコードを自由に受け渡すことが可能であり、柔軟な記述が可能となっている。

2. 接続意味論のインプリメンテーション

2.1 表示的意味論

属性変換文法は一般的にスタックマシン等のコードをターゲットとしてそれへの変換ルールを文法の中にアクション部として記述することで形式的なセマンティックスのあつかいを可能にしているが、表示的意味論に代表される数学的意味論においては、ターゲットコードは特定のマシンに依存しないコードであり、その解釈実行によってセマンティックスをシミュレートする(ターゲットコードの抽象化)。すなわち、変換文法は実用レベルでの形式性を保っているが、表示的意味論においてはその実用性よりはセマンティックスの抽象的表現能力に力点がおかれている。

属性文法は基本的にはシンタックス上の理論であり、文法に従って記述されたプログラムの認識機(パーザ)を作成し、生成された解析木上に情報を属性として付加するために利用される。一方、表示的意味論はシンタックス要素に対してその表示(Denotation)を定義する。表示が、対応するシンタックス要素の意味となる。例えば、式(4+2)、(2*3)に対する表示(意味)は共に同じ数6である。このことを形式的に表わすと

$$M : [Syn \rightarrow Sem]$$

となる。Synはシンタックス要素のタイプ領域を、Semはセマンティックス要素のタイプ領域を表わし、MはSynからSemへの意味関数となる。表示的意味論は伝統的には、再帰関数の停止問題、関数の自己適応で生じる値域上の矛盾、未定義の扱いなどを数学的に解決したスコット理論[34]をもとにしているが、本稿で対象とする表示的意味論は、単に、シンタックス領域から数学的対象物によって表記される、セマンティックス領域への意味関数によって構成されるものとする。

表示的意味論によってプログラミング言語のセマンティックスを与えたものとして[7,8,35]などがある。

2.2 接続意味論

プログラムのセマンティックスはプログラム構成子(例えば、ステートメント、式など)に対する入力情報とその変化(結果)で与えられる。イメージを図で表現すると図2-1 のようになる。

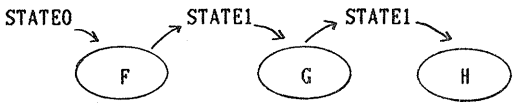


図2-1 直示意味論モデル

状態STATE は（名前　．値）の集合で、◯はプログラム構成子を表わす。そして、セマンティックスはSTATE の変化を記述することで与えられる（この他にも環境ENV も情報として渡されるが図では省略）。このようにして定義されるセマンティックスは直示意味論(direct semantics)と呼ばれる。

一方、実行の順序を何らかの理由によって途中でやめたり（例えば、実行時エラーの発生によって）、他の選択を行ないたい（例えば、ジャンプ）とした時、直示意味論では難しい。直示意味論では、実行過程はF 、G 、H 等の構成子上の意味関数によって直接的に受け渡されているからである。

GO TO 文を考えてみると、現時点でGO TO 行き先に対応するラベル位置を知っていなければならない。また、ジャンプがなければ続けて実行される次の構成子についても知っていなければならない。このセマンティックスを記述するためのモデルとしては次のようなものが考えられる（図2-2 ）：

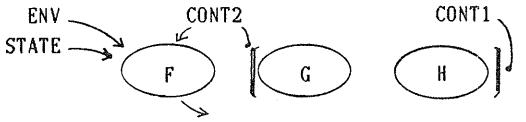


図2-2 接続意味論モデル

Fに着目するとFに対する入力は状態STATE 、環境ENV 、接続(continuation)CONT2 が与えられる。CONT 1 は〖F　G　H　．．．〗に対する接続であり、CONT2 はFの次を示す〖G　H　．．．〗に対する接続を表わしている。接続は環境、状態などを受け取り別の環境、状態を返す。また、時によっては別の接続を返す関数である。今、Fが"GO TO L1"であるとすると、もし[L1 / CONT] ENV ならば、ラベルL1に対応した接続CONTへ制御が移る。また、ラベルL1がまだ宣言されていなければ、前方参照に対するサーチをCONT2 に対して行なう。これが可能なのはCONT2 がFの残りのプログラム全部に対する接続を保持しているからである。このように、接続を導入することで構成子に対する情報は一端接続を通して間接的に次の構成子に渡される。もっと言えば、接続を入力として利用することで各構成子は独立に計算を行ない、結果を返す先を決定できる。このような考えにもとづいて出来上がっているセマンティックスの理論体系が接続意味論(continuation semantics)である。接続意味論を用いると、例えば、コルーチンやバックトラックは、一端抜けた手続きへの再入が可能であるようなセマンティックスによって記述することができる[8] 。

2.3　セマンティックスインプリメンテーション

表示的意味論は純粋にセマンティックス理論として発展してきたが、従来のシンタックス主導型の言語設計に対して、セマンティックス主導型[36]の言語設計の重要な柱となっている。そこでは、表示的意味論を直接記述している点に特徴がある。

Allison[37] はPascalの関数形式を巧みに用いて接続を直接インプリメントしている。Mosses[35]はセマンティック代数という表示的意味論を代数的に解釈したものを直接インタプリートするSIS というシステムを作成した。Wand[38]はラムダカルキュラスで表現された意味関数を特殊なコンビネータを導入することで変数フリーな形式にいったん変換し、その上でのオプティマイゼイションとインタプリテーションを与えている。また、Sethi[39] では、意味関数と接続の表現形式をYACC-like な記法とうまく融合させることでフォーマリティと実際性（コンパイラの自動生成）の両方を兼ね備えることができた。

Sethi のアイデアのポイントはプログラムフローの中継点に接続を対応させ、セマンティックスを接続間の関係によって定義している点にある。例えば、図2-3 はWHILE ループに対するプログラムフローであり、C0,C1 はそれぞれ接続に対応する。セマンティックスは次の式の {} で与えられる。

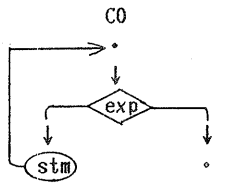


図2-3

```
stm: WHILE '(' exp ')' stm
  {$$ e c1 =
    cyclic c0=let e'=e[hbrk:=c1][hcom:=c0];
              in $exp | cond($stm e', c0, c1)}
```

詳細は論文[39]にゆずる。

3．　属性文法形式によって接続意味論をシミュレー

トすることの問題点

3.1 属性文法形式

属性文法と表示的意味論はそれぞれ対象とする領域がまったく同じではないが、古くから双方の関係が論じられている。特に、属性文法を代数的仕様などによって再定式化し、それを用いて表示的意味論をシミュレートする研究が行なわれている[29,40,41]。一方、defunctionalize されたインプリメント寄りの表示的意味論から属性文法モデルにあう形式に変換する研究がGanzinger[23] によって行なわれた。また、Paulson[24] は、拡張属性文法[41]の属性に表示を定義できるようにしたセマンティック文法を提案している。

Ganzinger は表示的意味論の属性文法的解釈という意味で属性文法形式(attribute grammar form)という用語を用い、Paulson では属性文法の形式はとっているがセマンティックスを記述しているのでセマンティック文法という呼び方をしている。本稿でも属性文法の形式だけを取り入れ、本来の定義を必ずしも厳密に守っている訳ではなく、かつセマンティックスを中心にした記述方式を提案するので特別な名称を採用することにする。Ganzinger の属性文法形式との混乱を招く恐れはあるが、本稿で以下に示す表現形式も”属性文法形式”と呼ぶことにする。理由は、ここで提案する形式は、基本的には属性文法モデルの枠組み内で属性タイプを拡張したものであり、かつ計算機構からみても共通点が多いからである。ただし、純粋な属性文法と区別をするために”形式”が付いている。

3.2 属性文法形式による接続意味論のシミュレート

属性文法形式は、属性文法の定義は保っているが、属性として表示(denotation)を許すことによりセマンティックスを記述できる点で属性文法と大きく異なる。また、計算モデルとしてみたとき、直接インタープリートできない点で表示的意味論とも異なる。Paulson のセマンティック文法によって簡単な算術式のセマンティックスを記述すると

```
expression<env、λs. exp1(s) + exp2(s)> =
  expression1<env, exp1> "+"
  expression2<env, exp2>                    (3.1)
```

となる。exp1とexp2は表示であり、このexpressionの意味はλs. exp1(s) + exp2(s)で表わされる。この文法の解析フェーズ（コンパイラ作成）、文法によるプログラムの解析フェーズ（コード生成）、実行（ラムダ式のインタプリテーション）は区別される。セマンティック文法は直示意味論だけを対象にしていて、接続を扱っていないので記述は非常に簡単であるが、一般に、接続まで許すといくつかの問題が生じてくる。今とりあげた例(3.1) で接続を考慮すると、

```
expression<env、K, K2> =                    (3.2)
  expression1<env,K1, K2> "+"
  expression2<env,λ v1.λv2. k(v1 + v2), k1.>
```

となる。接続属性間の依存関係に注目すると、

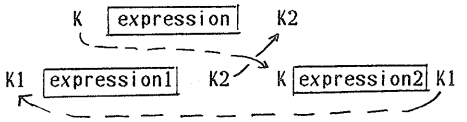


図3-1 属性文法形式での接続属性の依存グラフ

のような非循環ではあるが単純ではない属性依存グラフができる。また、(3.2) 式と意味は同じであるが形の異なる記述も考えられる：

```
expression<env、K, K2> =                    (3.3)
  expression1<env,λ V1.K1, K2> "+"
  expression2<env,λv2. k(v1 + v2), k1.>
```

いま示した例から２つの問題が考えられる：

1) 接続を属性として取り入れることにより、接続間の依存グラフが複雑になる(3.1) 。特に、再帰関数のセマンティックスを定義しようとすると依存グラフに循環性が生じる。

2) 実用レベルではラムダ式によって変数を完全に束縛できない場合も生じてくる。(3.3) では、expression2 のV1はどこからも束縛されていない。それは、表示的意味論では関数の入れ子で表示を管理できるのに対して、属性文法形式では、別々のノードで管理されるためである。

このような問題が生じるのは、属性文法の計算モデルとしての立場と、言語処理系を作成するために属性文法形式を用いる立場との間にギャップがあるからである。確かに、属性文法を合成属性のみによって再定式化し、その不動点定理によってセマンティックスを定義するアプローチ[41]を採用すれば理論上、上記

(1),(2) の問題は一部解決するが本質的解決にはつながらない。

3.3 LEAGによる接続意味論の記述例

前節を受けて、本節では、LEAGによる解決例を紹介することにする。ここでは、LEAGの計算モデルとしての立場は放棄し、LEAGでの各ルールは関数としてプログラムの解析、および必要な属性値の抽出も行ない、出力するコードの形式において接続意味論を実現する。

すなわち、LEAGのセマンティックスへの関与を一歩後退させて、解析時には、実行時に接続を利用するための情報を集める。また、セマンティックス属性はルールの引数としてではなく、生成されるコードの中でのみ用いる。更にいえば、コンビネータの導入によってそれらセマンティックスの属性に付随する変数を隠すこともできる。これは、タイプ属性とも関連する話題ではあるがここではこれ以上深くは立ち入らない。

既に取り上げた算術式を例にとると、LEAGでの記述は次のようになる。

```
(rule   expression (ENV CONTO)
  (syntax "<exp> + <exp>")
  (let  ((TYPE EXP1) (exp ENV CONTO))
        ((TYPE EXP2) (exp ENV CONTO))
        ((CONT
           (code
            "(M EXP1 ENV
              '(λ V1. (M EXP2 ENV
                    '(λ v2. CONTO (+ v1 v2))
               ))" ))
  in (ENV CONT)))
```

M は意味関数でありEXP はシンタクッス要素、ENV は環境、CONTは接続を表わす。ENV は式の実行では変化がないのでそのままコードの中に埋め込まれる。

このルールではプログラムの解析によってシンタクッス要素の部分を抽出し、セマンティックスにあたるコード部では意味関数M によって接続意味論をシミュレートしている。EXP1に対する接続は'(λ V1. ... であり、さらに中でのEXP2に対する接続は'(λ V2. ... によって与えられている。

このようなアプローチが実用レベルの言語で本当に有効かどうかは実験が済んでいない今の段階ではなんとも言えないが、LEAGのより広い分野への応用を目指して検討を続けていきたい。

文献

[1] Naur, P. ed.: Revised Report on the Algorithmic Language ALGOL 60., CACM 6 (1), 1-17, 1963.

[2] Knuth, D.E.: Semantics of Context-free Languages., Math. Sys. Theory 2 (2), 127-145, 1968. correction: in Math. Sys. Theory 5 (1), 95-96, 1971.

[3] Aho, A. V., Ullman, J. D.: Principles of Compiler Design., Addison-Wesley, 1978.

[4] Lewis, P.M. Rosenkrantz, D.J. Stearns, R.E.: Compiler Design Theory., Addison-Wesley,1978.

[5] Lewis, P.M. Rosenkrantz, D.J. Stearns, R.E.: Attributed Translations., J. Comput. Syst. Sci. 9, 279-307, 1974.

[6] Lewi, J. et all.: A Programming Methodology in Compiler Construction. (part 1), North-Holland, 1979.

[7] Tennent, R.D.: The Denotational Semantics of Programming Languages., CACM 19 (8), 437-453, 1976.

[8] Gordon, M.: The Denotational Description of Programming Languages., Springer-Verlag, 1979.

[9] Apt, K. R.: Ten Years of Hoare's Logic., ACM TOPLAS 3 (4), 431-483, 1981.

[10] Liskov, B. et all.: CLU Reference Manual., Lecture Notes in Computer Sience., 114, 1979.

[11] Goldberg, A., Robson, D.,:Smalltalk-80.,Addison-Wesley, 1983.

[12] Bobrow, D. G., Stefik, M.: The LOOPS Manual., Xerox PARC, August, 1981.

[13] 柴山、松田、米沢：並列オブジェクト指向言語ABCLによるプログラミング、「オブジェクト指向」鈴木則久編、共立出版、1985.

[14] 松田、米沢：ABCLユーザズマニュアル、東工大情報科学科内部資料、1984.

[15] Clocksin, W.F., Mellish, C.S.: Programming in Prolog., Springer-Verlag, 1981.

[16] Cardelli, L.: ML under Unix (pos3 version).,Bell Laboratory ,1983.

[17] Milner, R.: A Theory of Type Polymorphism in Programming., J. Comp. Sys. Sci. 17, 348-3

75 ,1978.
[18] Ward, S. A., Halstead, R.H.: A Syntactic Theory of Message Passing., J.ACM 27 (2), 365-383, 1980.
[19] Cardelli, L.: A Semantics of Multiple Inheritance., pp. 51-67, in Semantics of Data Types., Lecture Notes in Computer Science 173, 1984.
[20] 松田裕幸： 拡張属性文法に基づく言語の形式的定義について、静岡大学工学部情報工学科卒業研究 、1983.
[21] 松田裕幸： 拡張属性文法に基づく言語記述言語LEAGの概要と応用、第１０回情報処理学会ソフトウェア基礎論資料、10-2、 1984.9.
[22] 松田裕幸：A Language Based On Extended Attribute Grammars(LEAG): Its Theory, Implementation, and Applications. 東京工業大学情報科学科修士論文、1985.
[23] Ganzinger, H.: Transformational Denotational Semantics into Practical Attribute Grammars., 1-69, in [36]
[24] Paulson, L.: A Semantics-Directed Compiler Generator., POPL82, pp 224-233.
[25] 佐々、石塚： コンパイラ生成系 －Ｒｉｅ－、筑波大学 電子情報工学系 内部資料、1984.
[26] Riis, H. Madsen, M.: The NEATS System., DAIMI MD-44 (May 1982)., Computer Science Department, Aarhus University.
[27] Kastens, U. Hutt, B. Zimmermann, E.: GAG: A Practical Compiler Generator., Lecture Notes in Comp. Sci. 141 , Springer-Verlag., 1982.
[28] Kennedy, K. Warren, S.K.: Automatic Generation of Efficient Evaluators for Attribute Grammars., POPL76, pp 32-49.
[29] Mayoh, B.H.: Attribute Grammars and Mathematical Semantics., SIAM J. Comput. 10(3), 503-518., 1980.
[30] Kastens, U.: Ordered Attributed Grammars., Acta Informatica 13, 229-256.,1980.
[31] Bochmann, G.V.: Semantics Evaluation from Left to Right., CACM 19 (2), 55-62., 1976.
[32] Courcelle, B., Zannettacci, F-. P.: On the Expressive Power of Attribute Grammars., 161-172, IEEE Proc. of the 21th FOCS.
[33] Jazayeri, M., Ogden, W.F. Rounds, W.C.:The Intrinsically Exponential of the Circularity Problem for Attribute Grammars. CACM 18 (12), 697-706 , 1975.
[34] 高橋正子：スコット理論、情報処理解説、20 (11), 983-990, 1979.
[35] Mosses, P.: SIS: Semantics Implementation System., Aarhus University DAIMI MD-30 .,1979.
[36] Semantics-Directed Compiler Generation., Edited by Neil D. Jones Lecture Notes in Computer Science 94, Springer-Verlag .,1980.
[37] Allison, L.: Programming Denotational Semantics., Computer J. 26 (2), 164-174, 1983.
[38] Wand, M.: Deriving Target Code as a Representation of Continuation Semantics., ACM TOPLAS 4 (3)., 496-517, 1982.
[39] Sethi, R.: Control Flow Aspects of Semantics-Directed Compiling., ACM TOPLAS 5 (4), 554-595., 1983.
[40] Takeda, M. Katayama, T.: On Defining Denotational Semantics for Attribute Grammars., J. Inf. Proc. 5(1), 21-29 ,1982.
[41] Madsen, O.L.: On Defining Semantics by means of Extended Attribute Grammars., 259-299, in [36]
[42] Ganapathi, M. Fischer, C.N.: Description-Deriven Code Generation using Attribute Grammars.,POPL82, pp 108-119, 1982.