プログラミング－言語・基礎・実践 12－6
（１９９３．５．２８）

並列動作記述に対する文法プログラミングからのアプローチ

松田裕幸
東京工業大学総合情報処理センター
hmatsuda@cc.titech.ac.jp

属性文法を基にしたプログラミングシステムの研究を行なっている．文法プログラミングとは文法をプログラムと見なす立場の総称として用いている．本報告では，文法プログラムを記述するため新たに開発した言語 Leag により，並列に動作する各プロセスから発行されたイベントをグローバル時間で整列されたものを文として認識することで並列動作をシミュレートする．現在 Leag 自身には並列機能はなく，機能拡張に対するアイデアについても触れる．

GRAMMAR-BASED PROGRAMMING APPROACHES CONCURRENT BEHAVIOR DESCRIPTION

Matsuda Hiroyuki
Computer Center, Tokyo Institute of Technology
Oh-okayama 2-12-1, Meguro-ku, Tokyo 152, Japan
E-MAIL: hmatsuda@cc.titech.ac.jp

This paper proposes a programming system based on attribute grammar. Interpreting and extending a grammar as a program is generalized with the term *Grammar-based Programming*. A newly-designed language *Leag*, which realizes the grammar-based programming, describes the behavior of of concurrent processes by treating a sequence of events issued by them as a sentence. The sequential events are assumed to be interleaved into a sentence according to a global clock.

1 文法プログラミング

文法プログラミング (Grammar-based Programming) の概要を述べる．詳細は文献 [4] に譲る．**文法プログラミング**とは，入力データを文として認識できる文法を定義し，その文法構造を反映したプログラムを書くことである (図 1)．文法として LL(1) 文法を仮定し，LL(1) 文法によって解析され抽出された情報に対する操作を属性文法に基づいて記述する．

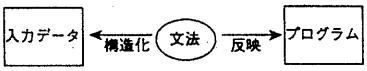

図 1: 文法プログラミング

文法プログラミングの目的は，明解な論理構造を持ったプログラムを書くための枠組を提供することにある．明解さの目安として，可読性 (readability) と，表現性 (expressiveness) をあげる．可読性は，プログラムの実行という動的な挙動がプログラムテキストから読みとれることと定義し，表現性はプログラマが意図した問題解決戦略をプログラム構造が反映していることと定義する．

可読性に関しては入力データの構成 (解析) 構造とプログラム構造を一致させることで，表現性に関してはプログラムの実行過程を入力データの構成過程に沿わせることで実現する．さらに文法プログラミングでは，文法規則と属性それぞれをプログラムの実行過程と情報操作に対応させることにより，制御の流れと実際の計算を分離する．このことも，可読性の向上に大きな役割を果たしている．

文法プログラミングを行なうためのプログラミング言語 **Leag** を設計した．Leag プログラムは関数型言語 Gofer[2] のプログラムに変換され，Gofer 上で動作する．

Leag は属性の遅延評価，属性同士のパターンマッチング，属性主導制御 (ガード機構)，属性値の型推論等の強力な機能を有する．特に，遅延評価によって，属性の依存関係にループが存在する属性クラスに対しても，解が不動点を持つ場合には属性評価が可能なクラスに対しプログラム可能となる．

以下，Leag による記述例をもとに文法プログラミングの概要を理解する．なお，Leag の構文規則を付録 A に載せる．最初に 2 進数列を 10 進数に変換するプログラムを示す．2 進数列は整数の列からなる文と考える．図 2にこの文の解析木と属性値の求まる過程を示す．

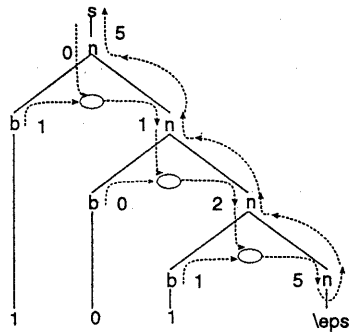

図 2: 2 進数列 101 に対する属性文法 AG による解析木

Leag プログラムは次の通りである．

```
GCode b2d is
type Term=[Int]
s(→v1) = n(0→v1)
n(sum→sum) = \eps
n(sum→v2) = b(→v1) n(sum*2+v1→v2)
b(→0) = 0
b(→1) = 1
GEnd
```

括弧 () を除いた部分が生成規則に対応し，括弧内に属性を記述する．\eps は空列を表す．「→」の左側が入力属性，右側が出力属性である．`type Term=[Int]` は入

力文が整数列(リスト)であることを宣言している. このプログラムはGoferの関数に変換されるが, その際の関数名はプログラム名をもってする. 今の場合は`b2d`になる. 従って, プログラムの実行は例えば,

```
b2d [1,1,0,1]
```

となる. これは, 2進数列1101の10進数値を求めており, 値13を得る.

入力文の構造は何も整数列や文字列に限らない. 次に, 2進木に対し, 葉の最小値でもって木全体の葉の値を置き換えるプログラムを見てみる. 最初にプログラムを示す:

```
data Tree = Tip Int | Fork Tree Tree
GCode repmin is
type Term=Tree
s(→tree) = t(min→(tree, m))
  where min=m
t(m→(Tip m, n)) = Tip n
t(m→(Fork t1 t2,min m1 m2))
  = Fork t(m→(t1, m1)) t(m→(t2, m2))
GEnd
```

`data`部分はGoferが提供するユーザ型定義を表している. このようにLeagプログラムはGoferプログラムと共存できる.

`repmin (Fork (Tip 3) (Fork (Tip 2) (Tip 5)))`

の結果は

`(Fork (Tip 2) (Fork (Tip 2) (Tip 2)))`

である. 上記プログラムは実は葉の最小値を求めるプログラムと, その最小値を使って葉の値を置き換えるプログラムの合成によって得られたものである. しかし, 詳細は文献[3]に譲る.

プログラム`repmin`において, 図3に示すように後方に対する属性依存が生じている. これはLeagが, 通常LL(1)文法と組で使用されるL-属性より広いクラスになっていることを示している. この依存関係では, 属性

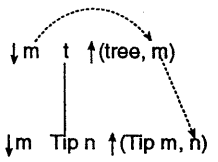

図3: プログラム`repmin`における属性依存

間のループは生じないが, 次の例のように属性間にループがあっても(図4), ループ内に不動点を持つ場合には計算は停止し, 解は求まる.

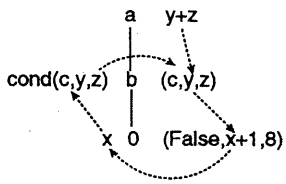

図4: プログラム`cag`におけるループ属性

```
GCode cag is
type Term=[Int]
a(→y+z) = b(cond(c,y,z)→(c,y,z))
b(x→(True,7,x+2)) = 1
b(x→(False,x+1,8)) = 0
GEnd
cond (c,y,z) = if c then y else z
```

例えば, `cag [0]`の答は17である. 最初`c,y,z`, 従って`x`の値が未知であるが, 3番目の規則を実行した時点で, `c, z`の値`Flase, 8`は確定するので, `cond`の計算により`x`の値8も確定する. 最後に`y`の値9も確定して答17を得る.

また, Leagでは空文を扱うこともできる. この場合には, Leagプログラムは関数型プログラミングのパラダ

イムに従う. これは, 次に示す階乗のプログラムで説明する.

```
GCode fact is
type Term=[]
fact(0→1) = \eps
fact(n→n*v) = fact(n-1→v)
GEnd
```

type Term=[] は入力文が空, すなわち無いことを表している. fact に関する生成規則と見るとこの実行は停止しない. しかし, 属性に関する収束条件 (n=0) によってプログラムは停止する. 例えば, fact 3 は結果6を返して停止する.

2 並列動作記述

本章では Reader-Writer 問題を例にとり Leag による並列動作の記述を行なう. Leag 自体には今のところ並列機能はなく, 並列に動作する各プロセスから発行される各種イベントをグローバル時間で整列したものを文として扱うことにより, 並列動作をシミュレートする.

以下に示す問題と解答はすべて Hemmendinger[1] に負うている. ただし, 文法プログラミングの形式に移し変えるために表現上の変更を行なっている.

最初に Reader-Writer 問題を整理しておく:

> 共有資源に対し複数の読み手と複数の書き手が存在する. 読み手は資源に対し同時アクセスが許される. 書き手は相互排除のもとでのみ資源にアクセスする. 共有資源に対する同期記述を行なう. ■

図5にあるように, 読み手/書き手からの呼び出し(call)を管理するための同期システムを仮定する. 同期システムは呼び出しを受け,「要求」,「開始」,「終了」のイベントの管理する. 図では, 読み手/書き手と同期システムと

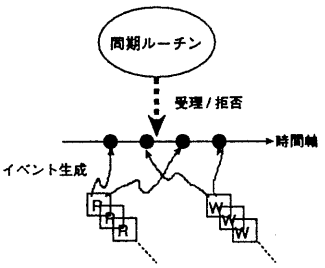

図 5: Reader-Writer 問題

の間の相互関係, イベントを生成する実体については述べられていない. また, 呼び出しが受け付けられて要求イベントが生成される (図6) までの機構に関しても一切説明していない. なお,「要求が待たされる (waiting)」とは, 要求イベントが受理された後, 開始イベントが発行されない状態を指す.

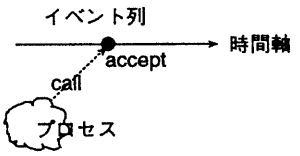

図 6: イベント

以下に述べるプログラムは同期システムの働きを記述したもので, 時間軸にそって生成されたイベント列を受理/拒否することで資源の割り当てに関する管理を行なう. すなわち, 受理されたイベント列は資源に対する正しい割り当てになっている.

資源への読み出し開始/終了, 書き込み開始/終了に対応して4つのイベントを定義する:

```
data Event = RS |RF |WS |WF
type Term = [Event]
```

読み出し要求, 書き込み要求に対するイベントは存在するがシステムの働きにおいて don't care の時は, null イ

ベントとしてあたかも存在していないかの扱いをする.

イベントに関し, 1トランザクション内での時間経過は無視, イベント間の状態遷移には1単位時間が経過, イベントの発生は要求−開始−終了の順, トランザクションの間プロセスは失敗しない, 等の仮定をおく. この仮定は文法モデルにおける終端記号の扱いと等価である. 構文解析を状態遷移で考えると, 1つの終端記号は状態をある状態から次の状態に遷移させる. この終端記号を1つのイベントの発生と捉えると受理は1単位時間で終了されると仮定できる. また, 生成規則を考えると適用される規則列は一定の順序に従った記号列を受理する. 例えば, [要求, 開始, 終了] という列を受理する規則を書くことは容易である.

最初に以下の性質を満たすminimal版の解を示す.

> 【minimal版】
> 読み手には同時アクセスを許し, 書き手を相互排除する. 資源が空いている時はRS, WSのいずれも受け付ける. WSの後はWFしか受け付けない. つまり, あるプロセスの書き込みの間, 他の書き込みイベントや読み出しイベントは受け付けない. また, RSの後はRSがいくつか続き, 同数のRFを処理する. つまり, 読み出しの間は書き込みは許さない.

次にこの仕様を満たすプログラムrwを示す:

```
GCode rw is
type Term=[Event]
reader_writer(n→) = \eps
    | /n >= 0/ RS reader_writer(n+1→)
    | /n > 0/ RF reader_writer(n-1→)
    | /n == 0/ WS reader_writer(-1→)
    | /n == -1/ WF reader_writer(0→)
GEnd
```

nは読み出しプロセス数を表し, n == -1の時は書き込み処理中, n == 0の時は読み出し, 書き込み可能状態, n > 0の時は読み出し中のプロセスがいることに対応する. /.../で囲まれた部分はガードとして働く. このプログラムの実行は例えば次のようになる:

```
rw [RS, RF, WS, WF] 0
```

最初は読み出しプロセスはないと仮定しているので, プロセス数に対応する属性nの値は0となる. このイベント列は読み出し開始, 終了, 書き込み開始, 終了なので正しく終了する. もし, [WS, RS] のようなイベント列が来たらこれは排除されなければならない.

さて, minimal版では資源が空いている場合, 書き込み, 読み出し, いずれのプロセスが優先するかは決めていなかった. そこで書き込み要求を優先するようプログラムを拡張する (writer-preference版). この場合, 書き込み要求のみを管理する必要があるので, 読み出し要求はnullイベントして扱う.

> 【writer-preference版】
> 書き込み要求のある時は, 読み出しを待ちにする[1].

書き込み要求に対する新しいイベントWRを追加:

```
data Event = RS |RF |WR |WS |WF
```

ここで, 拡張に関しプログラムrwを直接変更するのではなく, 上記性質を満たす別のプログラムwprefを考え, rwとwprefをマージするという方法をとる. ただし, その前にWR部分に関する規則をrwに追加しておく: プログラムrw'.

```
GCode rw' is
type Term=[Event]
reader_writer(n→) = ...
        | WR reader_writer(n→)
GEnd
```

[1] これだけの規則では書き込み終了後, すべてのイベントを受け付けることはできず, starvationを防ぐことはできない.

rw' は rw 同様, 資源への書き込み/読み出しからなる資源管理を行ない, 以下に示すプログラム wpref は, 書き込み要求を優先するような制御を担当する.

```
GCode wpref is
type Term=[Event]
writer_pref(qw→) = \eps
    | /qw == 0/ RS writer_pref(qw→)
    | RF writer_pref(qw→)
    | WR writer_pref(qw+1→)
    | /qw > 0/ WS writer_pref(qw-1→)
    | WF writer_pref(qw→)
GEnd
```

qw は書き込み要求プロセス数を表し, qw > 0 は書き込み要求あり, qw == 0 は書き込み要求無しに対応する.

最後にプログラム rw' と wpref を合成する. ここでは rw' と wpref は構造的に等価であるので, 文法の合成アルゴリズム[2] に従い合成を行なう: プログラム rw_wpref.

```
GCode rw_wpref is
type Term=[Event]
reader_writer((n,qw)→) = \eps
  | /n >= 0 && qw == 0/
    RS reader_writer((n+1, qw)→)
  | /n > 0/ RF reader_writer((n-1, qw)→)
  | WR reader_writer((n, qw+1)→)
  | /n == 0 && qw > 0/
    WS reader_writer((-1, qw-1)→)
  | /n == -1/ WF reader_writer((0, qw)→)
  GEnd
```

この新しいプログラムでは次の最初の例は正しく受理されるが, 2 番目の例は書き込み要求の後に読み出し開始を行なおうとしたので受理されない:

```
rw_wpref [RS, RS, WR, RF, ...]  (0,0)
rw_wpref [RS, RS, WR, RS, ...]  (0,0)
```

プログラム rw_wpref では, 書き込み終了後, 書き込み要求が発行され続けると待たされている読み出し要求が starvation 状態に陥る. 次に改良版を示す LimitWait.

【LimitWait 版】

書き込み終了後は, 待たされている読み出し要求がすべて実行終了するまで次の書き込み要求は待たされる.

イベント WF 受理後の rstart への遷移がこの役目を果たしている. ここで, 属性 rr は待たされている読み出しプロセス数を管理するために追加された. なお, 書き込み開始から終了までは一単位のイベントと見なすことができるので, ここでは WRITE イベント一つに集約させている.

```
data Event = RR | RS | RF | WR | WRITE
GCode limwait is
type Term = [Event]
limwait((r,qr,qw)→) =
    /r==0 && qr==0 && qw==0/ \eps
  | RR limwait((r,qr+1,qw)→)
  | /qr>0 && qw==0/
      RS limwait((r+1,qr-1,0)→)
  | RF limwait((r-1,qr,qw)→)
  | WR limwait((r,qr,qw+1)→)
  | /r==0 && qw>0/
      WRITE rstart((0,qr,qw-1)→)
rstart((r,qr,qw)→) =
    /qr==0/ limwait((r,qr,qw)→)
  | /qr>0/ RS rstart((r+1,qr-1,qw)→)
  | WR rstart((r,qr,qw+1)→)
  | RR rstart((r,qr+1,qw)→)
```

[2]文献 [4] 参照. ここでは直観的な説明にとどめる. 構造等価とは比較するプログラムにおいてすべての生成規則部分の「形」が一致することと定義する. 合成アルゴリズムでは同じ生成規則上に, 双方の属性をマージする. ガード内の述語は AND 条件によって結びつける.

```
GEnd
```

プログラム

```
limwat [RR,RR,WR,WRITE,RS,RS,RF,RF] (0,0,0)
```

は正しく終了するが，入力文

```
[RR,WR,WRITE,WR,RS,RF]
```

に対しては，書き込み終了後に書き込み要求が発行されているのでこのイベントは受理されない．

最後に，単純化されたモニターの記述を行なう．

【モニタ】

・モニタでは要求をいったん受理した後は，そのイベントに対し直ちに実行権を与えなければならない．

・書き込み終了後，待たされていた読み出しプロセスに権限が移るが，この読み出しプロセスがすべて終了した後のことはプログラム `limwait` では記述していない．ここでは，すでに要求のあった書き込みプロセスのうち一つに特権モードを与え読み込み終了後このプロセスの開始を認める．

以下にプログラムを示す．ここで，属性 `pw` は書き込みプロセスへの特権状態を表す論理値を値としてとる．なお，このプログラムはモニタ機構を記述したプログラムとすでに述べた `minimal` 版 `rw` の合成結果になっている．入力属性の 1 番目 `r`(資源読み出し中のプロセス数) とそれに対するガード条件が `minimal` 版から受け継いだ部分である．

```
data Event = RR | RS | RF | WR | WS | WF
GCode rw_limw is
type Term = [Event]
limwait((n,0,0,False)→)= \eps
limwait((n,0,0,False)→) =
  RR limwait((n,1,0,False)→)
limwait((n,qr,qw,True)→) =
  RR limwait((n,qr+1,qw,True)→)
limwait((r,qr,qw,False)→) =
  /r>=0 && qr>0/
    RS limwait((r+1,qr-1,qw,False)→)
limwait((r,qr,qw,pw)→) =
  /r>0/ RF limwait((r-1,qr,qw,pw)→)
limwait((r,0,0,False)→) =
  WR limwait((r,0,1,True)→)
limwait((r,qr,qw,True)→) =
  WR limwait((r,qr,qw+1,True)→)
limwait((0,qr,qw,True)→) =
  /qw>0/ WS limwait((-1,qr,qw-1,True)→)
limwait((-1,qr,qw,True)→) =
  WF limwait((0,qr,qw,False)→)
limwait((r,0,qw,False)→) =
  /qw>0/ limwait((r,0,qw,True)→)
GEnd
```

プログラム

```
rw_limw [WR,RR,RR,WR,WS,WF,RS,RF,RS,RF,WS,WF]
(0,0,0,False)
```

は正しく終了するが，次のデータ

```
[WR,RR,RR,WR,WS,WF,RS,RF,RS,RF,RR,RS,RF]
```

は待たされているすべての読み出しプロセス (2 個) が終了した後に，再び読み出し要求を発行しているためこのイベントは受理されない．

3 並列システム

前章では Leag プログラムはイベント列からなる文を認識する認識器として働きしか示さなかった．しかし，Leag が有する属性遅延評価機構を利用することによって，Leag 自身を並列システムに機能拡張することは可能である．まだ実現はされていないが，Leag を並列システムに拡張するための基本アイデアを述べる．

通常の文認識システムにおいては，Leag 同様，入力文を一度に与えることを考えると認識すべき文はすべて同時に揃っていなければならない．しかし，入力文を遅延属性で受けとれる[3]Leag では，文の解析を行ないながらある時点以降の認識を「中断」することが可能である (図 7)．

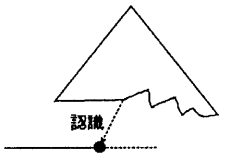

図 7: 文認識の中断状態

次にこの中断状態を利用して，文法自身がイベントを「生成」する．(図 8)．

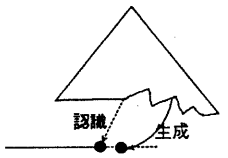

図 8: 中断状態からの生成

生成対象を自分自身ではなく，他文法の入力文の中に埋め込むことも可能である (図 9)．

今まで述べてきた文法を並列に動作する「自立」したシステムと見なすと，各文法から生成されるイベントは他に存在する管理用の文法に対する制御イベントと見なすこともできる．例えば，(P1, Req)[4] というイベントはプロセス P1 からの要求イベントとして解釈する．

管理用文法は対応すべき管理状内容に応じた複数の文法からなり，「協調」してそれぞれに応じた入力文の

[3] ここの機能に関しては本論文では説明していない．

[4] 2 つの要素からなるタプル

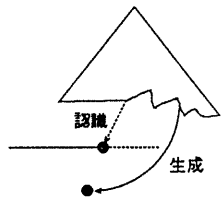

図 9: 中断状態からの生成 (2)

処理を行なう (図 10)．

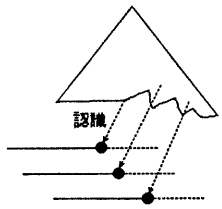

図 10: 制御文法

あるタイミングでどの入力文を認識するかは各イベントの種類とこの文法を構成する部分文法の記述によって決定される．

ここに述べたアイデアは現段階では非常にラフなものであるが，従来のシステムが，文法自身を手続化し，その中に種々の機能を追加することであたかも文法同士が解析途中で交信しあうような方式を採用しているのに対し，Leag の並列システムへの拡張はあくまで「文」を中心とした拡張になっている点が大きく異なる．文を中心とした通信/制御に依拠することにより，並列動作に対する理論の構築が明晰になることが期待される．

参考文献

[1] Hemmendinger, D.: Specifying Ada Server Tasks with Executable Formal Grammars, *IEEE Trans. Softw. Eng.*,

Vol.16, No.7(1990), pp.741-754.

[2] Jones, M.P.:
An introduction to Gofer, included as part of the distribution for Gofer version 2.28, Yale Univ. (1993).

[3] 松田裕幸: 文法プログラミングにおけるプログラム合成について, 日本ソフトウェア科学会プログラム合成変換研究会, 1992.

[4] 松田裕幸: 文法プログラミング, 日本ソフトウェア科学会論文誌投稿中.

A Leag の構文

Leag 言語の構文定義を行なう. メタ記号として, 定義::=, あるいは|, 省略可能{}, 0個以上の繰り返し{}*, 1個以上の繰り返し{}+を用いる. なお, メタ記号そのものを構文要素として用いる時は,「"」で囲む.

【構文規則】

Leag プログラム ::=
  GCode プログラム名 is
  { 入力文型宣言 }
  プログラム本体
  GEnd

【説明】

Leag プログラムは Gofer 関数に変換されるが関数名には''プログラム名''をあてる.

【構文規則】

入力文型宣言 ::= type Term = 入力文の型
入力文の型::=
  [] | [Int] | [Char] | [String] |
  [ユーザ定義の型] | ユーザ定義の型

【説明】

入力文型宣言は, Leag プログラムを Gofer に変換する際のコード生成に必要であり, 大きく「計算(空文)」,「文」,「構造」の3つに分類される. [] はプログラムに入力文がないことを表す. また, [Int], [Char], [String], [ユーザ定義の型] はそれぞれ, 入力文が整数列, 文字列, 文字列のリスト, ユーザ定義データのリストであることを表す. 入力文の型がユーザ定義の型そのものの時は「構造」に分類される. ここでは構造全体を入力文と見なす. 入力文型宣言を省いた時は, 入力文の型は [Char] と見なす.

【構文規則】

プログラム本体 ::= {G-規則}*
G-規則 ::= G-左項 = G-右辺 { "|" G-右辺}*
        {where 属性等式+ }
G-右辺 ::= \eps
        | { ガード } 終端要素 {G-右項}*
        | { ガード } {G-右項}+
終端要素 ::=
        入力文型宣言で指定した型の要素データ
G-左項 ::= G-項
G-右項 ::= G-項
ガード ::= / 述語 /
属性等式 ::= 適用属性 = 定義属性からなる式

【説明】

プログラム本体は複数の G-規則からなる. さらに, G-規則は, G-左項, G-右項, 終端要素などに分割される. 終端要素としては, 整数, 文字列などの他に, 構成子も含まれる. \eps は空列を表す. 現在の仕様では, 終端要素は右辺第一項以外には出現しないことを仮定している.

n_i を G-項または終端要素とすると,

$$n_0 = n_{i1}\, n_{i2} \cdots | \, n_{j1}\, n_{j2} \cdots$$

は

$n_0 = n_{i1}\, n_{i2} \cdots$

$n_0 = n_{j1}\, n_{j2} \cdots$

と等価である．

ガードは Gofer のガードと同じ働きをする．ただし，ガード中で用いられる述語（これも Gofer 関数）の引数は G-左項の入力属性に限る．

【構文規則】

G－項 ::= 項名(入力属性→出力属性)

入力属性 ::= (属性*)

出力属性 ::= (属性*)

属性 ::= Gofer で許される式．

式に現れる変数は属性変数とみなす．

【説明】

入力属性は 0 個以上の属性のタプル，出力属性は 1 個以上の属性のタプルからなる[5]．一般の属性文法では出発記号の入力属性は空としているが，Leag では，そうした制限は設けない．なお，Leag プログラムでは簡単のために 1 行目の G-左項が出発記号に相当するという仮定を設けている．

[5]要素 1 個のタプルは括弧を省略できるのは Gofer の場合と同じ．