プログラミングー言語・基礎・実践ー 13－9
（１９９３．８．２０）

並列システム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，文法による並列システム記述の分類を行ない，文法プログラミングがプロセスの外部観察挙動だけではなく，プロセス内部の記述もできることを示す．また，文法プログラミング言語 Leag による記述例も示す．

GRAMMAR-BASED PROGRAMMING APPROACHES TO CONCURRENT SYSTEMS

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

Authors have studied a programming system based on attribute grammar. Interpreting and extending a grammar as a program is generalized with the term *Grammar-based Programming*(GP). This paper classifies grammar specifications of concurrent systems and shows that the GP may not only describe the external behavior of concurrent processes but also the internal behavior of them. A newly-designed language *Leag*, which realizes the grammar-based programming, demonstrates these two kinds of description of concurrent systems.

1 文法プログラミング

本稿の主要テーマである並列システムの仕様記述を議論する前に，仕様システムの一つとしてここで提案する文法プログラミングについて概説する．文法プログラミング (Grammar-based Programming)[9] とは，直観的には「文法をベースにアルゴリズムを記述する」と言い変えることができるが，正確に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．

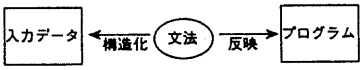

図 1: 文法プログラミング

2 進数列 10 進数変換を例に文法プログラミングを説明する．図 2は 2 進数列を文と見立てた時の，解析木と属性値 (10 進数) の求まる過程を示す．

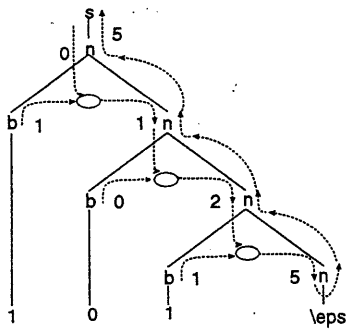

図 2: 2 進数列 101 に対する属性文法 AG による解析木

次に，文法プログラミング言語 Leag[9] によるプログラム例を示す．

```
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．すなわち，Term の要素 Int(今の場合 0,1) が終端記号として扱われることを表す．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 (今の場合 b2d) をもってする．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．

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[1]．Leag プログラムは関数型言語 Gofer[6] のプログラムに変換され，Gofer 上で動作する．

文法プログラミングの目的は，明解な論理構造を持っ

[1]Leag は基底文法として LL(1) を仮定しているが，属性主導制御によって ALL(1)[2] と同等のクラスを実現する．ALL(1) は LR(1) と同等であることが知られている．また，属性文法としても遅延評価機能により L-属性だけではなく，より広いクラス–絶対非循環–を対象とすることも示せる．

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様々なものが考えられるが，ここでは可読性 (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，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．

次節では，実行可能仕様としての文法プログラムについて述べる．

2 実行可能仕様

実行可能仕様 (executable specification) には形式仕様が持つ宣言的表現に対し手続き的解釈 (procedural interpretaion) [7] が必要となる．仕様を手続きとして読むためには，実行できるプログラムとして仕様自身にデータ定義と制御構造が含まれていなければならない (あるいはなんらかの形で与えられなければならない)[2]．

しかしその一方で，単にインタプリタあるいはトランスレータによって実行可能であることだけで仕様本来の宣言性を欠いては，実行可能仕様は単なるプログラミング言語と変わらないものになってしまう．文法プログラミングはこの宣言性と実行可能性の橋渡しを，データ型の定義と文法構造によって行なう．

[2]文法プログラミングにおける文法と Leag の関係は論理プログラミングにおけるホーン節と Prolog の関係に似ている．

文法プログラミングにおける文法構造はすでに見たように文法本来が持っている宣言的性質の他に，その手続き的解釈としてのプログラム構造および制御構造をも表している．かつ，この制御構造は文として見た場合のデータの (構成) 構造と一致する．さらに，属性文法によってプログラム構造と計算は分離される．特にこの最後の特徴は，従来仕様記述において主張されてきた「仕様の抽象化」よりある意味では重要である．それは，どんな抽象物であっても内部仕様において構造と計算の入組みは仕様としての良い性質を失わせるものだからである．仕様を積み上げ大規模な仕様を用意しなければならない段階において始めて仕様の抽象化が有効となる．

3 文法による並列記述の分類

文法をベースにした並列システム記述は，プロセス[3]そのものの「挙動 (behaviour) 」を記述する場合 [1, 4][立場 1] と，プロセスが発行/受理するメッセージ列をイベント[4]の列を見なし，この列 (文) を観察事象 (observal behavior) として認識する文法を記述する場合 [3, 5][立場 2] に分けられる．

立場 1 の場合，文法記述の中に並列操作を含むよう拡張する必要があり，Haas[4] は request,send,wait 等の文法要素を追加し，Anderson 等 [1] は real-time 属性文法という名前で，やはり送受信に対応した文法要素によって拡張した属性文法を提案している．立場 2 では，Hemmedinger[5] は文法記述を Ada のタスクに変換する方法を提案し，一方 Chapman[3] は属性評価器そのものを拡張する方向で文法記述に対する実行機構を得ようとしている．

本稿で提案する文法プログラミングによるアプロー

[3]立場によってエージェント，オブジェクト等様々な呼び方があるがここではプロセスに統一．

[4]本来イベントは挙動との関連で使われる用語であるが，ここではメッセージ列の要素を情報単位とは見ないで，ある動作の結果という立場に立ち，メッセージ列をイベント列と捉える．

チでは，立場 2 に立つ一方，文法自らがイベント要素を生成することによって立場 1，すなわち文法それ自身がプロセスの記述になる，をも実現しようとする [立場 3]．しかも，Leag を実行可能仕様と見なすことで，Hemmedinger, Chapman と比べ仕様記述から実行機構へのギャップを大幅に短縮している点も特記すべき点である．

プロセスを○，文法を△，認識を破線⋯>，生成を実線→で表すと立場 1 は図 3で表される．ここで，イベント列 (文) は複数プロセスからのメッセージ (イベント出力) のインタリーブを仮定している．従って，イベント事象に対してはグローバル時計を仮定する．

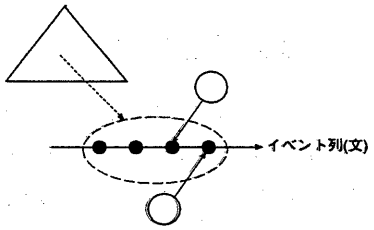

図 3: interleaved イベントと文法による認識

同じ立場 1 であるが，複数の文法が一つの文に対し補完あるいは制約として互いに協力しながら文認識作業を進める場合がある (図 4)．この時，文として見た時の終端記号が相互の同期条件として働く場合にはこれら文法を単に合成することで同等の働きを持った文法を手に入れることができる．この例は次節で紹介する．

次に，文法自身がイベントを生成する場合を考える．図 5は一方が生成を行ない他方が認識を行なうという単純な場合を表している．文を介在させない疑似的なコルーチン機構を本節の後半で例示する．

図 6は単に認識するだけではなく，同時に生成も行なう場合である．この際，今認識している文に対し生成イベントを終端記号として追加する機構については現在

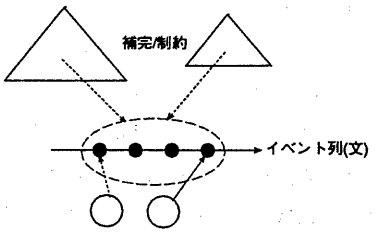

図 4: 複数の文法による認識

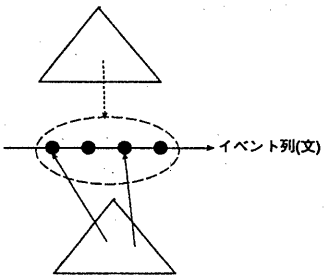

図 5: 文法間のイベント生成と認識

研究中である．なお，記述例を次節に示す．

最後に複数の文を扱う場合も想定できる (図 7)．これは分散協調関係の場合に対応させることができる．この場合も複数の文を対象とした同期/非同期制御が必要になってくる．

本節の例題として，2 つの文法間で一方が文を生成しながら他方がその文を受理し続けるという，文を間にはさんだ形での疑似コルーチンについて説明する．ここでは簡単のために，文は登場せず，リストデータを用いる．例として，与えられた整数を初期値とする (無限) 整数列を生成する文法 from と，その整数列を受けとって (無限) 素数列を返す文法 sieve を考える．Leag によるプログラムを次に示す:

```
GCode from is
type Term=[]
from(n→n:v) = from(n+1→v)
```

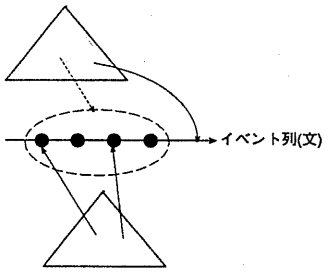

図 6: 同一文法によるイベント認識と生成

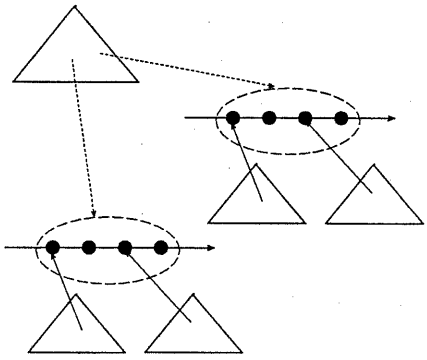

図 7: 複数の入力文を処理する文法

```
GEnd
GCode sieve is
type Term=[]
sieve(a:x→a:s) = sieve(sift a x→s)
  where
    sift a x =
      filter (not .  (multipleof a)) x
    multipleof a b = (mod b a) == 0
GEnd
primes n = take n (sieve (from 2))
```

どちらのプログラムにも文は存在せず(type Term=□), from の出力属性と sieve の入力/出力属性のみにより計算が進む.「:」は cons を表す. また,「. 」は関数合成を表す. 基底文法だけを見るとどちらも収束しないが, 属性 v, s に着目すると遅延 cons からなるリストとして収束する. 従って, (from 2), (sieve (from 2)) はそれぞれ遅延 cons からなるリストを返し, take によって遅延 cons 部分の評価が進む. 通常のコルーチンのようにプロセス間の制御の交替は明示されていないが, take, sieve 間で受け渡される遅延 cons リスト内には, take によって次整数を求める手続きが埋め込まれていると考えることができ, 一方, このリストを評価する seive の方も順次, from 側に次の整数を要求することで疑似コルーチンを実現している.

今後実際に文を介在させた形でのコルーチンの実現を考えている.

4 並列システムの記述

本節では前節で分類した並列記述のうち, 立場 2 に立った互いに補完し合う複数の文法によるイベント列認識の例 (図 4に対応) と, 立場 1 に立った文法自身プロセス挙動を記述した例 (図 6に対応) を示す.

4.1 補完しあう文法によるプロセスの外部観察記述例

共有資源に対し複数のプロセス (読み手, 書き手) がアクセスする Reader-Writer 問題 [5] を考える[5]. 読み手は資源に対し同時アクセスが許されるが, 書き手は相互排除のもとでのみ資源にアクセスを許される. 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 つのイベントを仮定する:

```
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

[5]本稿では細かい議論は省いているので, 詳細は [8] を参照されたい.

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.

これらイベントに関し補完しあう 2 つの文法を用意する. 一つは以下の性質を満たす文法で,

【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.

もう一方は, 次の性質を満たす文法である:

【writer-preference 版】
書き込み要求のある時は, 読み出しを待ちにする[6].

2 つのプログラム rw, wpref を次に示す:

```
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
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
```

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. プログラム rw 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.

プログラム wpref では, 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. こちらは, minimal 版において, 資源が空いている場合の書き込み, 読み出し, プロセスの優先順位を書き込み優先にするための文法である.

詳細な議論は文献 [8, 9] に譲るが, 上記 2 つのプログラムを属性のマージ, ガード内述語の AND 接続によって合成する. この際, イベント要素に WR と追加し, プログラム rw に

```
| WR reader_writer(n→)
```

を追加する.

```
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)→)
```

[6] ただし, これだけの規則では書き込み終了後, WR 以外のイベントを受け付けることはできず, starvation を防ぐことができない.

```
| /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)
```

4.2 プロセス挙動の記述例

ここでは, 文法自身によるプロセス挙動の記述例を示す[3]. メッセージの送り手と受け手を考える. メッセージの種類には通常の情報の他にAck等の制御情報も含む. 従って, 送り手, 受け手それぞれにメッセージを受ける/送るためのキューが必要である. 以下のプログラム sender では遅延consリストでこれを実現している. また, メッセージと共に1ビット情報も一緒に交換され, 次のメッセージにおいて必ず反転することが要求されている. 反転ビットが受け渡されない時はエラーである. さらに, メッセージ送受信の失敗も記述する(終端記号 SError, RError).

プログラム sender は, 送り手の挙動を記述したものである. 中括弧{}で括られた部分は, 属性変換文法でいうところの動作記号に対応するが, 文法プログラミングにおいては, 動作記号列を文要素として生成することを意味する. すなわち, 受け手(receiver)が受理するメッセージ列を形成する.

```
data Msg = ...
data Protocol =
    S(Msg,Int) | R(Int)| SError | RError
alt 0 = 1; alt 1 = 0
GCode sender is
type Term=[Protocol]
sequence((i:inq,outq,b)→) =
  seq((i,b)→)
  sequence((inq,outq++[i],alt b)→)
seq((m,b)→) = {S(m,b)} R(b)
  | {S(m,b)} RError  seq((m,b)→)
  | {SError}  bad((m,b)→) seq((m,b)→)
bad((m,b)→) = R(alt b)
  | RError
GEnd
```

なお, 動作記号から文生成までの機構に関しては現在のところLeag処理系では対応していない. これに関しては早急に実現する予定である.

5 最後に

文法による並列システムの記述はその研究成果が散発的に報告されているのみで活発な議論には至っていない. その最大の理由の一つが, 記述したものを実際に動作させ検証するためのテストベッドが用意されていないためである. 著者等が開発した文法プログラミング言語Leagは実行可能仕様として優れた検証・実行環境を提供する. Leagの属性値に対する型推論機構もこれを補完する. しかしながら, 文法によってプロセス自身の挙動を記述するには, 動作記号と並列制御との関係を明確にする必要がある. 現在この点に関し研究を続けている.

参考文献

[1] Anderson, D.P. and Landweber, L.H.: Protocol Specification by Real-Time Attribute Grammars, *Proc. IFIP WG6.1 4th Int. Workshop on Protocol Specification, Testing and Verification*,

Y.Yemini, R.Strom and S.Yemini(Eds), North-Holland, 1984, pp.457–465.

[2] op den Akker, R., Melichar, B. and Tarhio, J.: Attribute Evaluation and Parsing, *Attribute Grammars, Applications and Systems*, Alblas, H. and Melichar, B. (Eds), Lecture Notes in Computer Science **545**, Springer-Verlag, 1991.

[3] Chapman, N.P.: Defining, Analysing and Implementing Communication Protocols Using Attribute Grammars, *Formal Aspects of Computing*(1990)2, pp.359–392.

[4] Haas, O.: Formal Protocol Specification Based on Attribute Grammars, *Proc. IFIP WG6.1 5th Int. Workshop on Protocol Specification, Testing and Verification*, M. Diaz(Ed.), North-Holland, 1985, pp.39–48.

[5] Hemmendinger, D.: Specifying Ada Server Tasks with Executable Formal Grammars, *IEEE Trans. Softw. Eng.*, Vol.16, No.7(1990), pp.741–754.

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

[7] Kowalski, R.:*Logic for Problem Solving*, North Holland, 1979.

[8] 松田裕幸: 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-チ, 情処研報**PRG-12-6**(1993), pp.47–56.

[9] 松田裕幸: 文法プログラミング, ソフトウェア, (採録決定).