

Lorel-2 言語システムについて

榎本 進 片山卓也
(東京工業大学)

宮地利雄 榎本 肇
(工学部)

1. はじめに

Lorel-2は、グラフ等の組合せシステムの記述のために開発された高級問題向き言語である。我々は、数年前にLorel-1^{1)~4)}を開発し、種々の使用経験を積んだ結果、さらに豊富なデータ構成と強力な機能を備えたLorel-2を開発した。

Lorel-2の複合データとしては、set, tuple および再帰的なtupleであるtreeが用意されており、これらのデータを組合せることで、複雑なデータ間の関係を表現することができる。

Lorel-2のデータには、記憶域属性と呼ばれるものを指定することによって、setやtreeデータをハッシュ化したり、それらを種々の記憶領域に格納することができる。

Lorel-2に加えられた代表的な機能としては、ある条件を満たすsetをつくるset former、数学で用いられるsummationの一般形であるΣ式、および、連想検索式がある。

Lorel-2システムは、コンパイラ(オフティマイザを含む)と実行システムから構成されており、コンパイラがLorel-2ソースプログラムを基本命令と呼ばれる目的コードの列に変換し、これを実行システムが実行する。

コンパイラの開発は、構文図に意味づけを施した拡張構文図によって行われ、主としてデータ型の検査、エラー回復処理、および、setに関する目的コードの生成に工夫をこらしている。

実行システムは、マイクロ・プログラム化した基本命令を用いて、データ構造を処理しており、ゴミ集めもマルチ・タスク方式で実現している。

2. Lorel-2の言語仕様の概略

2.1. データとデータ型

- (1) integer: 整数で、データ型は i である。
- (2) real: 実数で、データ型は r である。
- (3) boolean: 論理値(true, false)を表わし、データ型は b である。
- (4) string: 8文字以内の文字列で 'ABC' のように ' ' で囲んで表わし、データ型は s である。
- (5) procedure: 手続き名であり、Lorel-2の手続きの他にCobol, Fortran, Hplのものが許される。オ1仮引数、..., オn仮引数、関数値のデータ型が各々、t₁, ..., t_n, t である関数のデータ型は [[t₁, ..., t_n]] → t である。
- (6) tuple: 固定長の線型リストであり、その構成要素である成分が e₁, ..., e_n であるtupleを次のように表わす。
(e₁, ..., e_n)

成分のデータ型が、t₁, ..., t_n であるtupleのデータ型は (t₁, ..., t_n) である。

また、成分のデータ型には、例えば、(first: i, second: r)のfirst, secondのように、成分識別子とよばれるセクタ名を付することができる。

tuple T の i 番目の成分の指定は、

(a) integer データによる T(i)

(b) 成分識別子による T.i

によって行うことができる。

tuple は、オ2成分以下に、特殊原子記号 nil を含ませることによって、木構造を表わすことができ、これをtreeと呼ぶ。節のデータ型が t の n 分岐treeのデータ型は、(t, *, ^{n個} *, *) である。

tree T の部分木である成分には、変数名を設定することができ、これを、subtree 変数 と呼び、そのデータ型は、

subtree T である。

また、empty tree は nil で表わされる。

(7) set: 可変長の線型リストであり、その構成要素である要素が $e_1, \dots, e_n$ である set を次のように表わす。

$\{e_1, \dots, e_n\}$

要素は全て同じデータ型でなければならず、そのデータ型を t とすると、この set のデータ型は $\{t\}$ である。

integer データ i, l, l を用いることによって、set X の左から i 番目の要素を $X[i]$ で、右から i 番目の要素を $X[\infty-i]$ で、 l 番目から l 番目までの要素からなる subset を $X[l, l]$ で指定できる。

set X の要素には、変数名を設定することができ、これを element 変数 と呼び、そのデータ型は element X である。

element 変数 E が表わす set 要素の n 個右隣の要素を $E[n]$ で、指定することができる。

また、empty set は $\{\}$ で表わされる。
[例 2.1] 変数 A のデータ型が、

$\{(first: s, second: i)\}$

element 変数 B が、 A の α 要素に設定されているとき、

$A[1, 2] \quad A[3] \text{ または } B[3]$
 $A: \{(ONE, 1), (TWO, 2), (THREE, 3)\}$
 $B: \quad A[3](2) \text{ または } A[3].second$

$\frac{1}{7} \frac{4}{3}$ を表わす変数 T のデータ型が、
 $(node: i, car: *, cdr: *)$ で、
subtree 変数 S が、 T の左部分木に設定されているとき、

$S \text{ または } T.car$

$S.car \text{ または } T.car.car$

$T: (1, (2, (3, \underline{nil}, \underline{nil}), \underline{nil}), (4, \underline{nil}, \underline{nil}))$
 $S:$

2.2. 変数の記憶域属性

変数には、宣言文において、normal, virtual, file, record, global および hash という 記憶域属性 を指定することができる。

(1) normal 属性 を与えられた変数は、常駐型記憶域 と呼ばれる常駐度の高いセグメント上にとられ、高速にアクセスすることができる。

(2) virtual 属性 を与えられた変数は、非常駐型記憶域 と呼ばれる、常駐度が通常のもので、比較的大きなセグメント上にとられる。

(3) file 属性 を与えられた変数は、ディスク上に ACOS-4 の V SAS ファイルとしてとられ、その編成も、順編成、相対編成、乱編成のものがある。

この変数は、file 制御文と、次の record 変数によってアクセスされる。
(4) record 属性 を与えられた変数は、前述の file 変数中の record を表わすことができる。また、“大域的”なデータでもあるため、Lorel-2 の外部手続きも含めて、他の言語 (Fortran, Hpl) の外部手続きからもアクセス可能な値を持つ。

(5) global 属性 を与えられた変数は、“大域的”なデータを表わす。

(6) hash 属性 は、set や tree 変数に対し、宣言文において、hash または key hash を指定することにより与えられる。

set 変数に対し、hash を指定することにより、要素全体がハッシュ化され、その set に対するデータの所属を速やかにきくことができる。

要素が n 成分 tuple である set 変数に対し、key hash を指定することにより、tuple の $\alpha 1 \sim \alpha n-1$ 成分がハッシュ化され、それらを key とした連想検索を速やかに行うことができる。

tree 変数に対し、hash を指定することにより、その節がハッシュ化され、部分木の所属や temperate matching による部分木の取り出しを速やかに行うことができる。

2.3. Lorel-2 の代表的な機能

(1) Σ 式 $(p | x_1, \dots, x_n \left\{ \begin{array}{l} \text{until} \\ \text{while} \end{array} \right\} B_\alpha : B_\beta) E$

・ x_i : くり返し条件 と呼ばれ、次の形

がある。

(a) $v = \alpha, \beta, \delta$ (b) $v \in X$ (c) $v \in X$

(a)は、変数 v に初期値 α 、終端値 β 、刻み δ でintegerを与える指定である。

(b)は、 X がsetなら、左から要素をとってきて、 v に与える指定で、 X がtreeなら、preorderに部分木をとってきて、 v に与える指定である。

(c)は、set X の要素を右からとってきて、 v に与える指定である。

• B_0, B_s : boolean 式

• E : 式

• p : 関数名または作用素

$x_1, \dots, x_n$ の指定によって、 x_n の方から先に変化するように値を v に取り出して行き、 B_s がtrueのときの E の値に次々と p を作用させて行き、このくり返しが終わったときの値を Σ 式の値とする。くり返しは、until指定があれば、 B_0 がtrueになったとき、while指定があれば、 B_0 がfalseになったとき終了する。

[例2.2]

$(+|I=1,7,2 \ J \in \{3,4,5\} \text{until } I>5:J \geq 4)(I+J)$
 $= (1+4) + (1+5) + (3+4) + (3+5) + (5+4) + (5+5) = 45$

(2) set former
 $\{E | x_1 \dots x_n \{ \text{until} \} \{ \text{while} \} B_0 : B_s \}$

Σ 式と同様にしてくり返しを行ったとき、 B_s がtrueのときの E の値からなるsetを、set formerの値とする。

[例2.3]

$\{I+J | I=1,7,2 \ J \in \{3,4,5\} \text{until } I>5:J \geq 4\}$
 $= \{1+4, 1+5, 3+4, 3+5, 5+4, 5+5\} = \{5, 6, 7, 8, 9, 10\}$

(3) 要素/部分木関係式

? $v : e_1 \in e_2$

e_1 がset/tree e_2 中に含まれるとき、この関係式はtrueとなり、element/subtree変数 v が満足する要素/部分木を指す。

e_1 には、成分として—を含むtupleも指定でき、このとき、—に対応する成分を無視して、 e_2 に対する所属をきくことができる。

この式を、要素/部分木関係式型の連想検索式と呼ぶ。

[例2.4]

$R: \{('TOKYO', 1), ('NAGOYA', 2), ('OSAKA', 3), ('NAGOYA', 4)\}$

のとき、 $?E: ('NAGOYA', _) \in R$ はtrueとなり、element変数 E は R の2要素を指す。

$T: (1, (2, \text{nil}, \text{nil}), (3, (4, \text{nil}, \text{nil}), \text{nil}))$

のとき、 $?S: (3, _, _) \in T$ はtrueとなり、subtree変数 S は下線の部分木を指す。

(4) 連想検索式

連想検索式には、前述の関係式型のもものと、次に述べる関数呼出し型のものがある。何れも、setまたはtree変数にhash属性を指定することで、高速の検索を行うことができる。

関数呼出し型の連想検索式：

$R[E_1, \dots, E_{n-1}]$

R は、 n 個の成分をもつtupleからなるsetで、この式は、 R の要素で、 $n-1$ 成分が $e_1, \dots, e_{n-1}$ に等しいtupleの n 成分からなるsetを値としてもつ。

[例2.5] 例2.4で $R['NAGOYA'] : \{2, 4\}$

(5) 代入文

代入文は4種類ある。

(a) $L := R$ (b) $L \downarrow := R$ または $\downarrow L := R$

(c) $L := \epsilon$ (d) $L := \equiv R$

(a)は、 R の値が L に代入される。

(b)は、 L の表わすset要素の右または左隣りに R が要素として挿入される。

(c)は、 L の表わすset要素が削除される。(d)は、 R の表わすset要素またはtree部分木を、elementまたはsubtree変数 L が指す。

[例2.6.] $A: \{1, 2, 3\}^B$ $X: \{10, 20\}$ のとき、

$X := A \Rightarrow X: \{1, 2, 3\}$ $\downarrow A[1] = 0 \Rightarrow A: \{0, 1, 2, 3\}^B$

$B := \epsilon \Rightarrow A: \{2, 3\}^B$ $B := A[3] \Rightarrow A: \{1, 2, 3\}^B$

(b) くり返し文

$\text{label}: (x_1 \dots x_n \{ \text{until} \} \{ \text{while} \} B_0 : B_s) \text{do } S_1 \dots S_n \text{od} [\text{label}]$

$x_1, \dots, x_n$ によって、 x_n の方から先に変化するようにくり返しが行われ、 B_s がtrueのとき文 $S_1, \dots, S_n$ が実行される。

くり返しは、until指定があれば、 B_0 がtrueになったとき、while指定があれば、 B_0 がfalseになったとき終了する。

くり返しの制御は、

exit(label), entrance(label), cycle(label)によって、各々、くり返しの終了、再実行、継続を行うことができる。

$\{l:(\dots)do \dots exit(l) \dots entrance(l) \dots cycle(l) \dots od\}$

2.4. フォログラム例

このプログラムは、undirected graph G に対して、そのdepth first searchを行った結果のspanning treeを T に与えるものである⁽⁵⁾。

G は、その要素であるtupleの第1成分がvertexを表わし、第2成分がそれに隣接するvertexのsetを表わす。

T は、その要素であるtupleがvertexの接続を表わし、 V は、その要素であるtupleの第1成分が G のvertexを表わし、第2成分が G をsearchするとき、既にvisitしたかどうかを表わす。

```
define Main ;
  type V_ELEM:element V ;
  variable G:{(i,{i})} key hash ,
           V:{(vertex:i,visit:b)} key hash ,
           T:{(i,i)} , E:V_ELEM ;

  define Search(v:i) ;
    variable E:V_ELEM , S:{i} , w:i ;
    if ?E:(v,_)eV  $\rightarrow$  E.visit:=true ,
       $\rightarrow$  /* error */...fi ;

    (S  $\in$  G[v])do
      (w  $\in$  S)do if ?E:(w,_)eV  $\wedge$   $\neg$ E.visit
         $\rightarrow$  T:=T U {(v,w)} ;
        Search[w] fi
      od
    od ;

    return ;
  end Search

  T:={} ;
  (E  $\in$  V)do E.visit:=false od ;
  (E  $\in$  V)do
    if  $\neg$ E.visit  $\rightarrow$  Search[v] fi
  od ;
  stop('STOP')
end Main
```

3. Lorel-2 システムの構成の概略

Lorel-2 システムは、コンパイラと実行システムを中心として、NEACシステム300 (利用者が使用可能な主記憶領域

は512 KB)上のオペレーティング・システムACOS-4の提供する機能を十分に利用した形で実現される。

Lorel-2 コンパイラは、Lorel-2 ソース・プログラムを読み込み、テープ型の検査とともに、解析、翻訳および最適化等を行い、Lorel-2 基本命令の列に変換して、ACOS-4のシステム記述言語Hplのソース・プログラムの形式で、出力を行う。

コンパイラによって出力されたHplプログラムは、Hplコンパイラによって、コンパイルユニットと呼ばれる機械語形式のモジュールに変換され、さらに、他言語のコンパイル・ユニットも含めて、スタティックリンクにより、リンクが行われ、実行可能なロード・モジュールが作り上

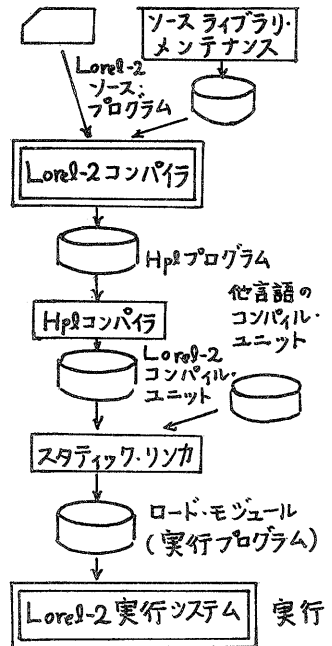


図3.1. Lorel-2プログラムの処理過程

げられる。このようにして作られたLorel-2実行プログラムは、概念的には、Lorel-2基本命令の列としてとらえることができ、これをLorel-2実行システムが実行する。

4. Lorel-2 コンパイラの構成

4.1. 拡張構文図によるLorel-2コンパイラの記述の概略

拡張構文図^{(6),(7)}は、従来の構文図に意味づけを施したもので、Lorel-2コンパイラは、拡張構文図と、そのnonterminalを1対1にrecursive programにおよぼすrecursive descentな技法によって記述される。Lorel-2の拡張構文図にお

いては、(1)BNF的構文検査、(2)データ型の検査、(3)中間コードの出力を記述している。(1)はnonterminalとterminalの接続によって指定される。(2)と(3)は、actionとassertionと呼ばれる、各々、その場所で必要とされる操作(代入等)と、条件を構文図に付けることで指定され、ここでは、それらをPascal-likeに書いている。(3)の中間コードとしては、ソースプログラムの木表現したものをを用いており、これをコンパイラが再びtraverseし、最適化等を行い、最終的に、Hplプログラムに翻訳する。

構文図で分岐が簡単に行えない場合は、先よみを利用して分岐を行う。

エラーは、3種類のクラスを設けて、適切なエラー回復処理を行っている。

4.2. データ型の検査

データ型の検査は、データ型のもつ2つの属性を考慮して効率よく行っている。ひとつは、この構文図を呼び出しているものによってデータ型が、既に与えられている相続的(inherited)データ型であり、他は、この構文図お

よびその呼び出す構文図でデータ型を構成する、構成的(synthesized)データ型である。

Lorel-2データでは、set, tuple, treeの複雑な入れ子があり、さらに、データ型がそれ自身で決まらない{ }, nilがあるため、そのデータ型を構成することは容易ではない。従って、多くの場合、データ型は相続的なものを利用した方が、その検査は速やかに行うことができる。Lorel-2では、変数名や手続き名のデータ型は、既に宣言文で与えられているため、それが関係する式に対しては、そのデータ型を相続的に与え検査することができる。例えば、代入文の左辺の変数→右辺の式、手続呼出しの仮引数→実引数がある。

構成的データ型にせざるを得ないものとしては、図4.1に与えられた関係式の左辺と右辺等がある。

4.3. 分岐における先よみ

一般に、分岐点の次がnonterminalであるとき、データ型の検査や中間コードの出力が一意に行えない場合が多

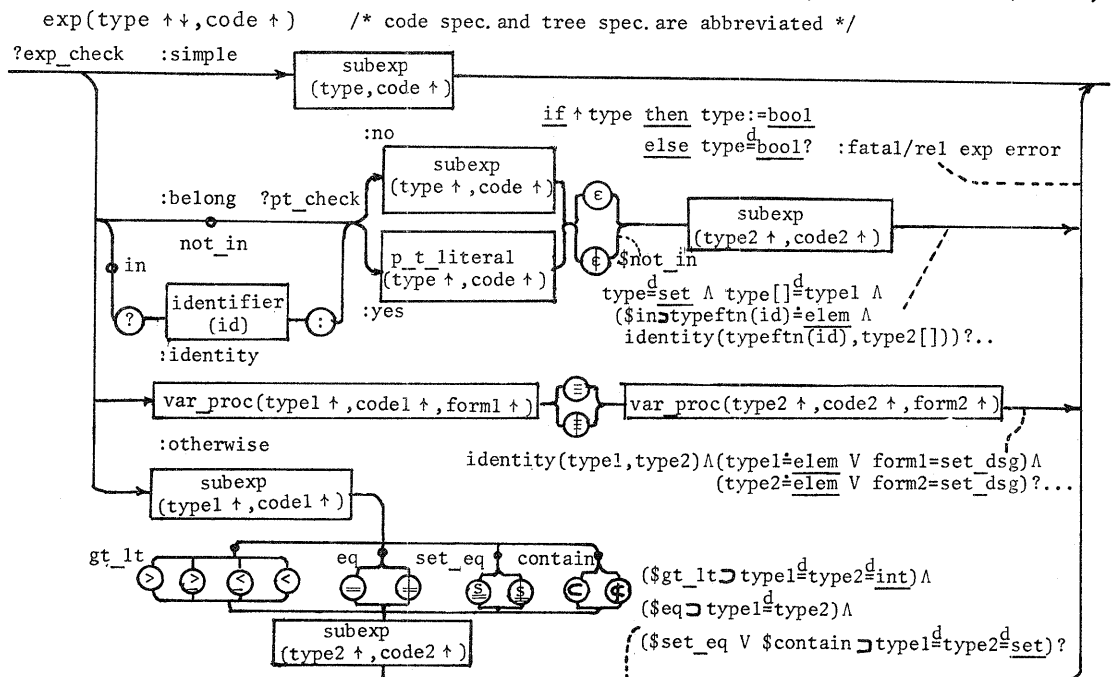


図4.1. 拡張構文図によるexpの記述

い。そこで、Lorel-2 コンパイラでは、不定長の先読み方式を採用して、それらを一意に行っている。

例えば、図1. 式 exp の先読み用構文図 exp-check は、非関係式型の式の次にくる演算子が $\in, \dots, \wedge$ であるかを調べにいき、各々に応じて、scalar 値 simple, $\dots$, otherwise をとるため、それを利用して、case 文で分岐する。

この先よみのおかげで、exp は、

- (1) subexp に直接おちるか、または、
- (2) subexp op subexp (op: $\in, \dots, \wedge$) におちるが、(1) では、exp が相続的データ型をもつ場合、それを直接 subexp に相続させることができる。

4.4. エラー処理

エラーのクラスには、(1) warning, (2) fatal, (3) abort があり、各々、(1) 適切な処置を行い、継続的にコンパイル、(2) エラーの起きた文を skip, (3) 即座にコンパイルを中止することを表わす。

4.5. set に関する only-one-copy 方式による目的コードの生成

Lorel-2 の set 式には、作用素として、 $\cup, \cap, -$ が含まれ、これに対し、単に、左と右の被演算子の set から新しい set をつくっていくコードは、領域的な損を招く。そこで、set 式の評価では、その構成要素の set を調べるだけで、copy は一度しか行わない方法を採用した。説明のため、処理プログラムも目的コードも Lorel-2 を採用する。原理的には、まず、生成される set は $\{\}$ に初期化し、(1) set 式を表わす binary tree Tree を preorder で巡っていき、変数に出会ったら、それから x にひとつずつ要素を取り出すコード ($x \in \text{Variable}$) do を出し、(2) Tree を root 方向に遡っていき、そのとき、(2-a) $\cup$ のノードなら、通過し、(2-b) $\cap$ のノードなら、その右部分木のノードである変数 Variable, $\cup, \cap, -$ を各々、($x \in \text{Variable}$), or, and, and-not におきかえることによって表わされる。論理式に対するコードを出力する。昇ってくる途中、初めて、ここで、 $\cap$ または $-$ に出会ったの

なら、Reg (レジスタ) にその評価値を入れ、さもなければ、前に出会った $\cap$ または $-$ の評価値 Reg と、ここでの評価値の and をとり Reg に入れるコードを出す。(2-c) のノードなら、 $\cap$ のときのようにして評価した値の否定を、(2-b) と同様にして Reg に入れる目的コードを出す。(3) root に達したなら、その変数に対する visit の終りを表わし、 $\cup$ のノードだけを遡ってきたなら、目的の set に x が含まれないなら、それを加えるコード "add x od" を出し、 $\cap, -$ を通ったのなら、if Reg $\rightarrow$ add x fi od を出す。末尾の od は、(1) での do に対応

```
Code[/* Target_set:={}; */...];
down: (T  $\in$  Tree)
do sel:=false;
  (until Variable[t.node.op])
  do if T.node.visit='left'  $\rightarrow$  T:=T.right,
    cycle,
    /* visit..'not' */  $\rightarrow$  cycle(down)
  fi
od;
Code[/* (x  $\in$  Variable)do */...];
mark:=true;
back: ( ) /* infinite loop */
do T:=father[T];
  if T=nil  $\rightarrow$ 
    if sel  $\rightarrow$  Code[/* if Reg  $\rightarrow$  add x
      fi od; */...],
       $\rightarrow$  Code[/* add x od; */...];
    fi;
    if mark  $\rightarrow$  exit(down),
       $\rightarrow$  entrance(down)
    fi
  fi;
  if T.node.op=' $\cap$ '  $\rightarrow$ 
    if mark  $\rightarrow$  T.node.visit:='all' fi;
    Select[T.right,B];
    if sel  $\rightarrow$  Code[/* Reg:=RegAB; */...],
       $\rightarrow$  Code[/* Reg:=B; */...];
    sel:=true
  fi,
  T.node.op='-'  $\rightarrow$ 
    if mark  $\rightarrow$  T.node.visit:='all' fi;
    Select[T.right,B];
    if sel  $\rightarrow$  Code[/* Reg:=Reg $\wedge$ B; */...],
       $\rightarrow$  Code[/* Reg:= $\neg$ B; */...];
    sel:=true
  fi,
  /* op.. $\cup$  */  $\rightarrow$ 
    if T.node.visit='not'  $\rightarrow$ 
      if mark  $\rightarrow$  T.node.visit:='left';
        mark:=false
      fi,
      /* visit..'left' */  $\rightarrow$ 
        if mark  $\rightarrow$  T.node.visit:='all'
        fi
      fi
    fi
  od back
od down;
```

する。

このようにして、 $\cap$, $-$ の右部分木に含まれる変数以外のもの全てをvisitしたとき、コード生成が終る。

プログラムにおいて、Treeのデータ型が、
(node:(op,visit:m),left,right:*)

op部が演算子または変数名を表わし、visit部が'not'なら、その部分木の変数は全てvisitしていないことを表わし、'left'なら、左部分木の変数だけvisitしたことを表わし、'all'なら、部分木の全ての変数をvisitしたことを表わす。visit部の初期値は全て'not'とする。

selは、tree上昇時に $\cap$, $-$ のノードを通過したかを表わし、markはtree上昇時にtrueなら、これからvisitする時のために、そのノードのvisit部を書き換えることを意味する。Variableは、ノードが変数名かを表わし、fatherは、subtree変数Tの親を表わす関係で、TがTreeのものなら、その値はnilである。

Codeは、その中のコメントの表わす目的コードを出力する手続きである。Select(T,B)は(2-b),(2-c)での論理式を生成するための次のような帰納的プログラムである。

```
/* Select(T,B)...B:new logical variable */
if Variable[T.node.op]
  → Code[/* B:=(x ∈ Variable); */...];
  return fi;
Select[T.left,L]; Select[T.right,R];
if T.node.op='U' → Code[/*B:=L V R; */...],
  T.node.op='∩' → Code[/*B:=L ∧ R; */...],
  /* op..'-' */ → Code[/*B:=L ∧ ~R; */...];
return;
```

図4.2に変数のvisitする道筋を表わし、その出力コードは次のようになる。

```
(x ∈ A)
do add x od;
(x ∈ B)
do
  B1:=(x in C);
  Reg:=B1;
  if Reg →
    add x fi
od;
(x ∈ D)
do
  B2:=(x in E);
  Reg:=~B2;
  if Reg →
    add x fi
od;
```

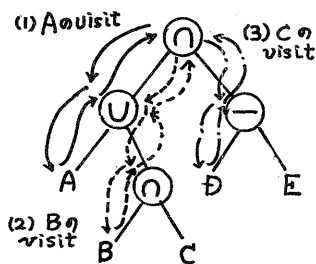


図4.2. $A \cup B \cap C \cup D - E$ に対するvisitの道筋

5. Lorel-2 実行システムの構成

5.1. 実行システムの概要

Lorel-2 実行システムは、Lorel-2 実行プログラムの実行を行い、また実行環境の管理を行うハードウェア/ファームウェア/ソフトウェアから構成された実行管理システムである。

Lorel-2 コンパイラにより翻訳され、スタティック・リンクにより結合された実行可能な Lorel-2 プログラムは、Lorel-2 実行システムにより実行される。

Lorel-2 実行システムは、概念的には Lorel-2 コンパイラが出力した基本命令の系列を実行する仮想機械であり、ACOS-4の実行管理の上で実現されており、その一部は拡張ファームウェアを利用している。

5.2. 実行システムの設計方針

実行システムの設計にあたっての方針は、次の通りである。

(1) 構成の形式化

記憶構造をセル構造として抽象化し形式的記述を行うこと、Lorel-2 実行プログラムを Lorel-2 基本命令の列として把握することなどの形式化を通して、概念的整理を行い実現を容易にした。

(2) 実行速度の高速化

基本命令のデコードをソフトウェアにより行うインタプリタ方式ではなく、ハードウェアが直接に基本命令のデコードを行うコンパイラ方式とし、リスト構造処理のための基本命令(copy, traverse等)をファームウェア化し、実行速度の向上をはかり、さらにソフトウェア部分についても実行効率の良いシステム記述言語 Hpl によって記述している。

(3) マシンやシステム機能の有効利用

NEAC システム 300 上のハードウェアおよび OS の提供している豊富な機能を実効に利用することにより、実行効率の向上をはかるとともに、実現を容易にしている。利用された機能の主要なものは、仮想アドレス空間、多重タ

スク環境、スタックによる手続き呼出し機構、VSAS ファイル編成、例外処理機構等である。

(4) デバック支援機能の強化

実行時におけるエラー発生位置の正確な指摘、エラー発生時の各変数の値の参照、プログラム実行経過の追跡、実行時における各変数の値のインタラクティブな参照および変更などの機能を備えて、LoveL-2 プログラムの開発中におけるデバックを支援する。

(5) システムの柔軟性

新しい周辺装置の導入などの処理系の稼働後に発生する動作環境の変化等にも柔軟に対応できるような余裕をもった設計を行い、また他言語との手続き単位でのリンクも許すように考慮している。

5.3. 実行システムの具体的構成

LoveL-2 実行システムは、構造的には図5.1に示すように、NEAC システム 300 上の固有ハードウェア/ファームウェア、LoveL-2 用拡張ファームウェア、LoveL-2 用基本ソフトウェア・サブルーチンから構成されている。

LoveL-2 実行プログラムは、概念的には LoveL-2 基本命令の系列であり、LoveL-2 実行システムは、これを実行するための仮想機械と考えられる。

LoveL-2 基本命令は、概念的な実行の機能単位であって、機械語様式の特定の形式を備えているわけではなく、手続きの呼出しであったり、拡張機械命令であったり、単一または一連の固有機械命令であったりする。そして、

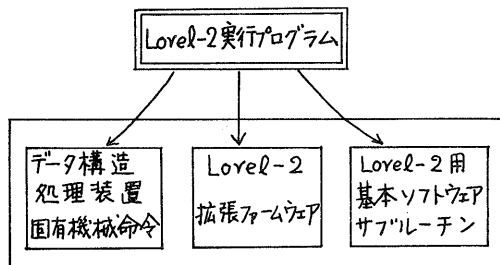


図5.1. LoveL-2実行システムの構成

LoveL-2 実行システムのソフトウェア部分が組込まれた、LoveL-2 実行プログラム全体としては、拡張機械命令を利用している部分を除き、ACOS-4により提供されている他言語の実行プログラムと同様の形式を持ち、同様に実行される。

拡張ファームウェア部分は、セル構造の処理を行う拡張機械命令を実行し、基本ソフトウェア・サブルーチン部分は一部の複雑な基本命令の実行や、プログラムの実行環境の管理、およびオペレーティング・システムとのインタフェース等の機能を担当し、オープンまたはクローズド・ルーチンの形式で実行プログラム中に組込まれる。

基本命令のデコードや固有機械命令により実現されている一部基本命令の実行は、固有ハードウェア/ファームウェアにより実行される。

また、機能的、論理的な観点からは、LoveL-2 実行システムは図5.2に示すように、基本命令実行処理部、実行管理、セルフォール管理、手続き呼出し管理、例外/エラー管理、ファイル入出力処理部、デバック支援部の7つの部分から構成されている。

基本命令実行処理部は、LoveL-2 実行プログラムから基本命令を読み出し、これを実行する。

実行管理は、プログラム実行のための環境を管理し、実行システムの初期化処理や終了処理などを行う。

セルフォール管理は、セルフォール上のフリーセルとカーベジセルをリストとして管理し、またカーベジセルからフリーセルへの再生処理を行う。

手続き呼出し管理は、LoveL-2 手続きの呼出しや退出に伴ない、スタック・セグメン等に置かれたその手続きに関連する情報の更新、および外部手続間におけるデータ型の検査などを実行する。

例外/エラー管理では、拡張ファーム

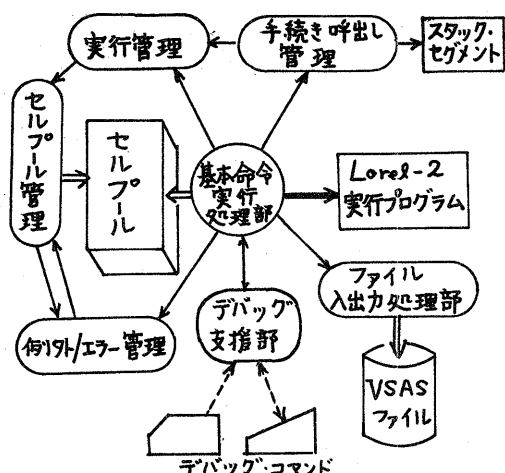


図5.2. Lorel-2実行システムの論理的構成

ウェアからソフトウェア処理を要求する機能的な割出し(トラップ的例外)や、基本命令実行処理部等から通知される Lorel-2 プログラム実行中に発生したエラーの検出、分類、処理を行う。

ファイル入出力処理部は、ファイル変数のアクセスのための file 制御文を実行し、OS のデータ管理とのインタフェースをとりながら、VSAS ファイルの処理を行う。

デバッグ支援部は、Lorel-2 利用者のデバッグ作業を援助するため、コンソールから対話的に、またはカードから実行時コマンドの指定に基づいて、プログラムの実行経過の追跡や変数の読出し/変更を行う機能を提供している。

5.4. 内部データ構造の概略

ここでは、Lorel-2 実行システムで用いる主要なデータ構造について述べる。

(1) セル構造

global, file 以外の属性を持つ Lorel-2 変数は、set や tree といった可変長データを含むことがあるため、その値はセル構造として記憶される。セル構造の一例を図5.3に示す。セル構造はスタック・セグメント上に置かれる V-エントリ と呼ばれる記述子を介してアクセスされ、セルプール上に用意されたセルを単位として構成され、セル間は

原則として双方向リンクにより結合されている。

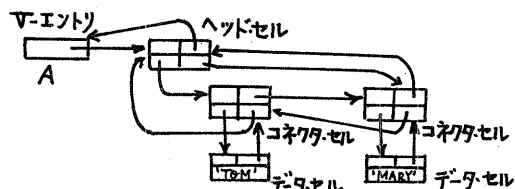
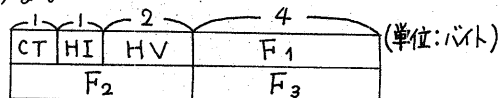


図5.3. セル構造の例 A: {'TOM', 'MARY'}

セルは、1個が16バイトの大きさで図5.4に示すフィールド構成を持つ。各セルには、その機能に従ってCTフィールドで定義されるセルタイプが与えられている。ポインタは通常のデータ記述子を使用しており、4バイト長である。



CT: セルタイプ HI: ハッシュ情報
F₁~F₂: ポインタフィールド HV: ハッシュ値
(中間ハッシュ値を含む)

図5.4. セルのフィールド構成

セルは4K個ずつで1個のセルプールを構成し、仮想記憶システム上のセグメント(64Kバイト)と対応づけられている。セルプールは複数個を用意しており、そのうち1個だけは主記憶内常駐度を上げる指定を持ったセグメントにより実現されており、常駐型記憶域と呼ばれる。フリーセルやカーベジセルのリストは、セル構造の局所性を向上させるためセルプール単位で管理される。また、カーベジセルからフリーセルへの再生処理を行うカーベジ・コレクタは、子タスクとして、主タスクとは非同期的に実行させることによりCPU効率の向上をはかっている。

(2) ハッシュ検索用データ構造

Lorel-2 では、ハッシュ検索される変数に対しても任意の書換えが許されており、このため、図5.5に示すようなデータ構造が作られる。

ハッシュ表は、Lorel-2 実行システム内に唯1個存在し、各ハッシュ値に対応

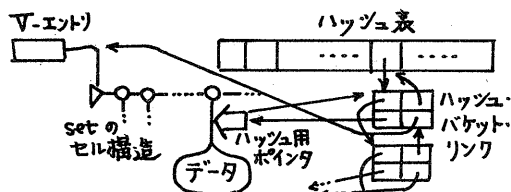


図5.5 ハッシュ検索のためのデータ構造の例

するバケット・リンクへのポインタ配列である。また、変数の書換えに伴うハッシュ値の再計算に際して無駄な計算を省き、効率を上げるために複合データに対するハッシュ値を、再帰的な形式で定義すると同時に、セルのHVフィールドにはハッシュ値計算の途中結果が保存されている。なお、ハッシングの計算は、データ構造処理装置の固有命令“HASH”をくり返し適用することにより実行される。

5.5. 実行速度の時間計測

(1) セルのtraverse, copyに要する時間、

traverse 30 μ 秒/要素

copy 600 μ 秒/セル

(全て、ソフトウェアの時の実験)

traverse, copy等の大きなセル構造に対する命令は、基本ソフトウェア・サブルーチンで構成されているため、その起動には、常に100 μ 秒必要とする。

(2) ファームウェア化による高速化

現在までに、フリーセルからのセルの切り出し、還元、カーベジセル構造へのセルの接続のファームウェア化を行った。その結果、ひとつのセルの切り出し、還元等に要する時間は、240 $\rightarrow$ 50 (μ 秒/セル)となった。従って、(1)のcopy命令も、600 $\rightarrow$ 410 (μ 秒/セル)となった。

(3) 要素関係式による連想検索

要素関係式 $? E: e_1 \in e_2$ に於て、 e_2 として、
 $\{(\{1,2,3\}, 1,2,1) \times 100,$
 $(\{1,2,3\}, 1,2,2) \times 100, (\{1,2,3\}, 1,2,3) \times 100\}$

を与え、 e_1 の検索にハッシュを用いた場合と、そうでない場合の比較を行う。

ここで、 $\times 100$ は要素が100個の並びを表わす。

(1) e_1 として、 $(\{1,2,3\}, 1,2,2)$ を用いて、101

番目の要素を検索すると、

ハッシュ : 6 m秒

非ハッシュ : 200 m秒

(2) e_1 として、 $(\{1,2,3\}, 1,2,3)$ を用いて、201番目の要素を検索すると、

ハッシュ : 5 m秒

非ハッシュ : 433 m秒

およそ、ひとつの要素を比較するのに、2 m秒かかると考えられるため、3~4 m秒がハッシュ値計算のための時間と考えられる。

6. おわりに

現在、LOREL-2は、コンパイラの開発を行っており、実行システムは既に稼働している。また、LOREL-2システムと関連して、マイクロシミュレータを開発しており、その完成によって、copy, traverse等の命令も、ファームウェア化を容易に行うことができると考えられる。

〔参考文献〕

- (1) 片山, 日比野, 榎本, 榎本: 論理関係処理言語 LOREL-1, 情報処理15, No.5
- (2) 榎本, 片山, 榎本: LOREL-1による記号処理の一例, 記号処理シンポジウム報告集(1974)
- (3) 榎本, 片山, 榎本, 吉田: LOREL-1による構造線の処理と接続, 第16回プログラミング・シンポジウム報告集
- (4) 榎本, 片山, 榎本: LOREL-1の性能評価, 情報処理学会第16回大会, 73
- (5) A.V.Aho, et al.: The Design and Analysis of Computer Algorithms, Addison-Wesley(1974)
- (6) 片山, 松本, 榎本: 拡張構文図を用いた recursive descent コンパイラの生成システム, 情報処理学会第18回大会, 306
- (7) 榎本, 片山, 榎本: 拡張構文図による recursive descent な LOREL コンパイラの構成, 情報処理学会第18回大会, 307
- (8) 宮地, 片山, 榎本, 榎本: LOREL-2 実行システムの構成, 電子通信学会, 計算機研資料 EC 77-4