スケーラブル統合プログラミング言語モデル LMNtal の実用化

(課題番号 16300009)

平成 16 年度～平成 18 年度科学研究費補助金
(基盤研究 (B))
研究成果報告書

平成 20 年 3 月

研究代表者

上田 和紀

(早稲田大学理工学術院教授)

はじめに

LMNtal (elemental と発音) は，多重集合や並行処理やモビリティなどの概念を持つさまざまな計算モデルを簡明な形で統合することを目的として，研究代表者と研究分担者が提案したプログラミング言語モデルである．

LMNtal における計算は，階層構造と接続構造の双方を表現する「階層グラフ」の書換えに基づいている．階層構造と接続構造は，計算システムにおいてはもちろんのこと，社会組織から生体に至るまで，あらゆる場面で見ることのできる構造化のしくみであり，しかも両者はしばしば併存する．そこで，階層グラフ書換えに基づく実用的なプログラミング言語を具体的に提示してその有用性を示すことは重要なチャレンジであると考えた．

本基盤研究は，計算モデルとして確立しつつあった LMNtal をプログラミング言語に成長させるために，実用プログラミングに必要な機能を設計し，実装技術を確立し，実際に処理系を構築して広く公開することを目標に推進した．

本研究によって得られた主要成果は以下の通りである．

(a) 階層的多重集合書換え計算モデルに基づく統合プログラミング言語の確立

階層グラフ書換えモデルに基づくプログラミングの実用化に必要な機能を特定して，LMNtal のための新たな言語機能の設計を行った．主な機能としては，組込みの型の概念を提供する型付きプロセス文脈，モジュールシステム，他言語インタフェースなどがあげられる．さらに，型付きプロセス文脈の概念に基づいて算術演算などの基本的な組込み機能を付与するとともに，モジュールシステムと他言語インタフェースを用いて多様なライブラリの構築を行った．

また，LMNtal に今後付与する予定の機能の一部について，検討と試作を行った．与えられた初期状態からの簡約経路をすべて求める非決定的実行機能を設計・試作し，探索問題や検証問題への応用への道を開いた．また，データの流れとプロセスの規模を同時に扱う静的型体系を構築し，これに基づいて型検査・型推論を行うアルゴリズムを設計・試作した．

(b) 実装技法の確立と処理系の作成・公開

本基盤研究の準備として，逐次型実行時システムの核部分の試作実装，実装の観点からみた言語仕様の再検討と詳細化，将来の分散実装につながる規準実装方式の詳細な文書化等

を平成 14〜15 年度に実施した．基盤研究期間中は，それらの成果を発展させる形で，処理系の作成と改良に当たってきた．

本処理系は Java 処理系上で稼動し，大きく分けて，LMNtal のために新たに設計した中間言語へのコンパイラと，中間命令列を実行をつかさどる実行時処理系から構成されている．

LMNtal は細粒度の並行性を持つが，膜による階層化・局所化機能とルール移送機能をもつことから、正しくかつ効率的な実装方式は自明でない．そこで，複数の非同期タスクで実行でき，並列分散実装への拡張が可能なスケジューリング方式と排他制御方式を設計・実装した．またこれがデッドロック不発生，漏れのないルール適用検査などの要件を満たすことを確認した．

また，各タスクの逐次実行の効率化のために，LMNtal 特有の最適化技術を検討し，ルールの共通部分の編み上げや，マッチングコスト見積もりに基づくルール左辺の命令列並び替えなどを実装した．さらに，中間言語の解釈実行機能に加えて中間言語から Java への変換 (トランスレータ) 機能を実装した．

並列分散処理については，将来の本格導入に向けて機能の試作を行い，ライブラリの拡充と最適化を行った．

実用面では，統一されたインタフェースで LMNtal から Java オブジェクトを利用できる仕組みを作り，Java 標準のクラスライブラリのほとんどを LMNtal から利用できるようにした．また実行可視化機能を設計・実装して，グラフ自動整形機能をはじめとする機能拡充および性能向上を実現した．さらに宣言的グラフィクスおよび GUI 機能の実装を行い，広範な用途への対応可能性を確認した．

(c) 記述能力の検証

LMNtal は，並行論理プログラミングとその解析・実装技術に関する研究代表者らの研究を背景に，

- 並行論理プログラミング自身
- そこから派生した制約プログラミング言語 CHR (Constraint Handling Rules)
- 多重集合概念をもつ多数の理論計算モデルのエッセンス

を簡明な形でまとめることを目指した言語モデルであり，多くの関連する並行分散計算モデルをもつ．

そこで，計算モデルとしての LMNtal の基本的記述能力の検証のために，既存の代表的な計算モデルを LMNtal でエンコードして，開発した処理系上で動作確認を行った．並行計算モデルの代表である π 計算，階層やモビリティの概念をもつ Ambient 計算，関数型言語のモデルである λ 計算，制約プログラミングのための CHR (Constraint Handling Rules) など，多様な計算モデルのエンコードに成功した．

*　　*　　*

このように本研究では，階層グラフ書換えに基づく言語モデル LMNtal に対して，

- 実用言語としての機能の検討・設計
- Java へのトランスレーション機能を含むコンパイラと実行時処理系の作成
- 実行時処理系内部の非同期実行方式の設計と実装
- 並列分散実装
- プログラミング環境の実現

を行い，それと同時に，並行・分散・制約処理分野のさまざまな計算モデルを LMNtal にエンコーディングすることによって，言語の広範な記述能力を実証した．教員と高いプログラミング能力をもつ学生とからなるチームが一致協力して，5 万行規模の言語処理系を構築し継続的に発展させることに成功し，大学における言語処理系のチーム開発の経験とノウハウを蓄積することができた．

また本基盤研究は，今後の新たな展開への基盤を与えることとなった．平成 19 年度には，標準の実行時処理系よりも高い空間効率と時間効率をもつ実行時処理系 SLIM の開発を始めるとともに，LMNtal のシステム検証への応用を目指して，SLIM をベースとするモデル検査器 LMNtalMC の試作を行った．理論面では，最も古典的かつ重要な計算モデルである λ 計算の LMNtal へのエンコーディングが，λ 計算自体に対して新たな知見を与えつつある．このように LMNtal とその処理系は，プログラミング言語としてだけでなく，計算モデル研究やシステム検証のための統合的なワークベンチとしても有望であることがわかってきた．

本成果報告書ではこれらの研究開発成果のうち，基盤研究の期間中に成果として確立した部分について，論文誌や国際会議の発表論文に基づいて詳述する．なお，本基盤研究の最大の成果は，ソフトウェア成果，すなわち開発した LMNtal 処理系である．

`http://www.ueda.info.waseda.ac.jp/lmntal/`

で公開しているので，参照・活用していただきたい．

研究組織

研究代表者: 上田 和紀 (早稲田大学理工学術院教授)
研究分担者: 加藤 紀夫 (産業技術総合研究所システム検証研究センター)
(研究協力者: 中島 求 (早稲田大学大学院理工学研究科, 現在日本電信電話 (株)))
(研究協力者: 永田 貴彦 (早稲田大学大学院理工学研究科, 現在 NTT ソフトウェア (株)))
(研究協力者: 矢島 伸吾 (早稲田大学大学院理工学研究科, 現在三菱電機 (株)))
(研究協力者: 原 耕司 (早稲田大学大学院理工学研究科, 現在ソニー (株)))
(研究協力者: 水野 謙 (早稲田大学大学院理工学研究科, 現在日本アイ・ビー・エム (株)))
(研究協力者: 乾 敦行 (早稲田大学大学院理工学研究科, 現在 (株) 日立製作所)))
(研究協力者: 工藤 晋太郎 (早稲田大学大学院理工学研究科, 現在 (株) 日立製作所)))
(研究協力者: 櫻井 健 (早稲田大学大学院理工学研究科, 現在 (株) エー・アンド・デイ)
(研究協力者: 岡部 亮 (早稲田大学大学院理工学研究科, 現在三菱電機株式会社))
(研究協力者: 中野 敦 (早稲田大学大学院理工学研究科, 現在 (株) 野村総合研究所))
(研究協力者: 村山 敬 (早稲田大学大学院理工学研究科, 現在ソニー (株)))

研究経費

(単位千円)

	直接経費	間接経費	合計
平成 16 年度	2,500	0	2,500
平成 17 年度	3,500	0	3,500
平成 18 年度	2,300	0	2,300
総計	8,300	0	8,300

研究発表

学会誌等論文

[1] 中島求, 加藤紀夫, 水野謙, 上田和紀：LMNtal を用いた分散処理の実現. 第 8 回プログラミングおよび応用のシステムに関するワークショップ (SPA 2005), 2005 年 3 月, pp. 101–110.
http://spa.jssst.or.jp/2005/papers/papers/spa2005-05-3.pdf

[2] Ueda, K. and Kato, N., LMNtal: a Language Model with Links and Membranes. In *Proc. Fifth Int. Workshop on Membrane Computing (WMC 2004)*, LNCS

3365, Springer, 2005, pp. 110–125 (invited lecture).

[3] 工藤晋太郎, 加藤紀夫, 上田和紀：LMNtal 処理系におけるグラフ構造の操作機能の設計と実装. 情報科学技術レターズ, 2005, pp. 9–12.

[4] Ueda, K., Constraint-Based Concurrency and Beyond. In *Proc. Workshop on Algebraic Process Calculi, The First Twenty Years (PA'05)*, Aceto, L. and Gordon, A.D. (eds.), BRICS Notes Series NS-05-3, 2005, pp. 227–230.

[5] 上田和紀：プログラムと対称性. 夏のプログラミングシンポジウム「アッと驚くプログラミング」報告集, 情報処理学会, 2005, pp. 69–74.

[6] 乾敦行, 原耕司, 水野謙, 上田和紀：階層グラフ書き換え言語 LMNtal 処理系とその応用例. 第 8 回プログラミングおよびプログラミング言語ワークショップ (PPL2006), 2006 年 3 月, pp. 119–133.

[7] Kazunori Ueda, Logic Programming and Concurrency: a Personal Perspective. *The ALP NewsLetter*, Vol. 19, No. 2, May 2006.

[8] Kazunori Ueda, Norio Kato, Koji Hara and Ken Mizuno, LMNtal as a Unifying Declarative Language. In *Proc. Third workshop on Constraint Handling Rules (CHR 2006)*, 2006, pp. 1–15 (invited talk).

[9] Kazunori Ueda, Constraint-Based Concurrency and Beyond. *Electronic Notes in Theoretical Computer Science*, Vol. 162 (2006), pp. 227–231.

[10] 上田和紀：純粋 λ 計算の階層グラフ書換えへのエンコーディング. 第 9 回プログラミングおよびプログラミング言語ワークショップ (PPL2007), 2007, pp. 221–232.

[11] 乾敦行, 工藤晋太郎, 原耕司, 水野謙, 加藤紀夫, 上田和紀：階層グラフ書換えモデルに基づく統合プログラミング言語 LMNtal. コンピュータソフトウェア, Vol. 25, No. 1 (2008), pp. 124–150.

[12] 村山敬, 工藤晋太郎, 櫻井健, 水野謙, 加藤紀夫, 上田和紀：階層グラフ書換え言語 LMNtal の処理系. コンピュータソフトウェア, Vol. 25, No. 2 (2008), pp. 47–77.

[13] Kazunori Ueda, Encoding Distributed Process Calculi into LMNtal. *Electronic Notes in Theoretical Computer Science*, Vol. 209 (2008), pp.187–200.

口頭発表

[14] 加藤紀夫, 上田和紀. 階層グラフ書換え言語における並行プロセスの型推論. 情報処理学会第 50 回プログラミング研究会 (SWoPP2004), 2004 年 7 月.

[15] 原耕司, 水野謙, 矢島伸吾, 永田貴彦, 中島求, 加藤紀夫, 上田和紀：LMNtal 処理系および他言語インタフェースの設計と実装. 情報処理学会第 50 回プログラミング研究会 (SWoPP2004), 2004 年 7 月.

[16] 中島求, 加藤紀夫, 水野謙, 上田和紀：LMNtal 分散処理系の設計と実装. 日本ソフト

ウェア科学会第 21 回大会論文集, 2004 年 9 月, pp. 149–153.

[17] 加藤紀夫, 水野謙, 上田和紀：言語モデル LMNtal の操作的意味論の設計. 日本ソフトウェア科学会第 21 回大会論文集, 2004 年 9 月, pp. 153–163.

[18] 坏弘明, 加藤紀夫, 上田和紀：階層グラフ書き換えによるプロトコル検証. 第 7 回プログラミングおよびプログラミング言語ワークショップ (PPL2005), 2005 年 3 月, p. 41.

[19] 矢島伸吾, 加藤紀夫, 上田和紀：小規模制御系向け LMNtal 処理系の設計と実装. 第 7 回プログラミングおよびプログラミング言語ワークショップ (PPL2005), 2005 年 3 月, p. 41.

[20] 櫻井健, 加藤紀夫, 水野謙, 上田和紀：LMNtal コンパイラにおける並び替えとグループ化を用いた命令列の最適化, 第 4 回情報科学技術フォーラム (FIT2005), 第一分冊, 2005, pp. 33–36.

[21] 水野謙, 加藤紀夫, 原耕司, 上田和紀：階層グラフ書換え言語 LMNtal 処理系における非同期実行の実現. 日本ソフトウェア科学会第 22 回大会講演論文集, 3A-4, 2005 年 9 月, 9pp.

[22] 工藤晋太郎, 乾敦行, 櫻井健, 上田和紀：OCaml による LMNtal 実行時処理系 OCaMNtal の実装. 第 8 回プログラミングおよびプログラミング言語ワークショップ (PPL2006), 2006 年 3 月 (ポスター発表).

[23] 水野謙, 上田和紀：非決定的 LMNtal とその検証への応用. 第 8 回プログラミングおよびプログラミング言語ワークショップ (PPL2006), 2006 年 3 月 (ポスター発表).

[24] Kazunori Ueda, Norio Kato, Koji Hara and Ken Mizuno, LMNtal as a Unifying Declarative Language: Live Demonstration. In *Proc. 22nd Int. Conf. on Logic Programming (ICLP'06)*, LNCS 4079, Springer, 2006, pp. 457–458 (デモ発表).

[25] 上田和紀：分散プロセス計算の LMNtal へのエンコーディング. 日本ソフトウェア科学会第 23 回大会論文集, 1A-4, 2006 年 9 月, 8pp.

[26] Kazunori Ueda, Hierarchical graph rewriting as a unifying model of concurrency. LIX Colloquium on Emerging Trends in Concurrency Theory, École Polytechnique, France, November 2006 (invited talk).

[27] 中野敦, 上田和紀：階層グラフ可視化ツール "UNYO-UNYO" (うにょうにょ) の設計と実装. 第 9 回プログラミングおよびプログラミング言語ワークショップ (PPL2007), 2007 年 3 月 (ポスター発表).

[28] 石川力, 堀泰祐, 岡部亮, 村山敬, 上田和紀：軽量な LMNtal 実行時処理系 SLIM の設計と実装. 情報処理学会第 70 回全国大会, 6ZJ-8, 2008 年 3 月.

[29] 岡部亮, 上田和紀：階層グラフ書換え言語 LMNtal によるモデル検査. 情報処理学会第 70 回全国大会, 6ZJ-9, 2008 年 3 月.

[30] 岡部亮, 上田和紀：階層グラフ書換え言語 LMNtal によるモデル検査. 第 10 回プロ

グラミングおよびプログラミング言語ワークショップ, 2008 年 3 月 (ポスター発表).
[31] 堀泰祐, 石川力, 岡部亮, 村山敬, 上田和紀：高速 LMNtal 実行時処理系 SLIM の設計と実装. 第 10 回プログラミングおよびプログラミング言語ワークショップ, 2008 年 3 月 (ポスター発表).

*　　*　　*

本報告書の構成を簡単に記す．各章ができるだけ独立に読めるようにするため，基本的な定義等は繰返し掲げている．

第 1 章は，本研究開発の基礎となった言語モデル LMNtal の仕様について述べている．本章の内容は前掲の論文 [2] に基づいているが，その後の研究成果を反映させた改訂を行っている．

第 2 章は，LMNtal に実用プログラミング言語に必要な機能を付加し，その記述力を多くの例題を用いて示した成果を紹介している．本章の内容は論文 [11] に基づいている．

第 3 章は，本研究開発の中心課題である LMNtal 処理系作成の経過と成果を詳述している．本章の内容は論文 [12] に基づいている．

目次

第1章

言語モデル LMNtal

LMNtal (pronounced "*elemental*") is a simple language model based on graph rewriting that uses logical variables to represent links and membranes to represent hierarchies. The two major goals of LMNtal are (i) to unify various computational models based on multiset rewriting and (ii) to serve as the basis of a truly general-purpose language covering various platforms ranging from wide-area to embedded computation. Another important contribution of the model is it greatly facilitates programming with dynamic data structures. This chapter describes an overview of LMNtal, its syntax and semantics, program examples, and related work.

1.1 Introduction

This work is motivated by two "grand challenges" in computational formalisms and programming languages. One is to have a computational model that unifies various paradigms of computation, especially those of concurrent computation and computation based on multiset rewriting. The other is to design and implement a programming language that covers a variety of computational platforms which are now developing towards both wide-area computation and nanoscale computation. As the first step towards these ends, this chapter proposes a language model LMNtal (pronounced "*elemental*") whose design goals are as follows:

1. *simple* — to serve as a computational model as well as the basis of a practical programming language (hence a language *model*).
2. *unifying and scalable* — to unify and reconcile various programming concepts. For instance, LMNtal treats
   (a) processes, messages and data uniformly,
   (b) dynamic process structures and dynamic data structures uniformly, and
   (c) synchronous and asynchronous communication uniformly.
   Also, through such uniformity and resource-consciousness implied by (a) and (b)

above, LMNtal is intended to be *scalable*, that is, be applicable to computational platforms of various physical scales.

3. *easy to understand* — since we often use figures to explain and understand concurrent computation and programming with dynamic data structures, the language is designed so that computation can be viewed as diagram transformation.
4. *fast* — optimizing compilation techniques are an important subject of the project, though this chapter will focus on basic concepts.

We briefly describe the design background of LMNtal. The first author designed Guarded Horn Clauses (GHC) [16] in mid 1980's, a concurrent language that made use of the power of logical variables to feature channel mobility. Various type systems such as mode and linearity systems were later designed for GHC [17]. A lot of implementation efforts and techniques have been accumulated over the past two decades. Concurrent logic programming was generalized to concurrent constraint programming that allowed data domains other than finite trees, and a concurrent constraint language Janus [14] chose multisets (a.k.a. bags) as an important data domain. Another important generalization was Constraint Handling Rules (CHR) [9] that allowed multisets of atomic formulae in clause heads. CHR was designed as a language for defining constraint solvers, but at the same time it is one of the most powerful multiset rewriting languages.

Given these two extensions, a natural question arises as to whether (the multiset aspect of) the two extensions can be unified or embedded into each other. LMNtal was designed partly as a solution to this question. The language design was first published in [18]. It was then reviewed and revised through intensive discussions, receiving feedback from the implementation effort that ran in parallel. This chapter reflects the latest design published in [20], which in turn refines the semantics published in [19].

1.2 Overview of LMNtal and Related Work

The "four elements" of LMNtal are *logical links, multisets, nested nodes, and transformation* — hence the name LMNtal. This section elaborates these four elements, touching on related work.

1. **Logical links** — Structures of communicating processes can be represented as graphs in which nodes represent processes and links represent communication channels. Likewise, dynamic data structures can be represented using nodes and links. LMNtal treats them uniformly, that is, links represent both one-to-one communication channels between logically neighboring processes and logical neighborhood relations between data cells.
   Two major mechanisms in concurrency formalisms are name-based communi-

cation (as in the π-calculus) and constraint-based communication using logical, single-assignment variables (as in concurrent constraint programming [17]). Of these, links of LMNtal are closer to communication using logical variables in that (i) a message sent through a link changes the identity of the link and (ii) links are always private (i.e., third processes cannot access them). The first point is the key difference between LMNtal and the π-calculus. However, LMNtal links are different also from links of concurrent logic/constraint programming and CHR in that LMNtal has no notion of *instantiating* a link variable to a value.

LMNtal links are non-directional like chemical bonds. However, if links are always followed in a fixed direction to reach partners, the direction could be represented and "reconstructed" using appropriate type systems.

2. **Multisets of nested nodes** — There have been many diverse proposals of computational models equipped with the notion of multisets, early examples of which include Petri Nets and Production Systems. Concurrent processes naturally form multisets; Gamma [2] and Chemical Abstract Machines [3] are two typical computational models based on multiset rewriting; languages based on Linear Logic [11] take advantage of the fact that the both sides of a sequent are multisets; Linda's tuple spaces are multisets of tuples.

   However, not all of them feature multisets as first-class citizens; many of the programming languages featuring multisets (e.g., Gamma, Linda, CHR) incorporate them in a way different from other data structures. The advantage of having multisets as first-class citizens is that it gives us greater expressive power such as the nesting and the mobility of multisets.

   LMNtal features multiset hierarchies and encapsulation by allowing a multiset of nodes enclosed by a membrane to be viewed as a single node. Hierarchical multisets can be found in the Ambient Calculus [4], the P-system [13], the bigraphical model [12], the Seal Calculus [5], the Kell Calculus [15], as well as in the fields of knowledge representation [7].

   Hierarchization of multisets plays many important rôles, for instance in (i) logical management of computation (e.g., user processes running under administrative processes), (ii) physical management of computation (e.g., region-based memory management), and (iii) localization of computation (i.e., reaction rules placed at a certain "place" of the hierarchy of membranes can act only on processes at that place).

3. **Transformation** — LMNtal has a rewrite-rule-based syntax. There has been a lot of work on graph grammars and transformation [1], including hierarchical graph transformation [6], but LMNtal's emphasis is on its design from the programming language point of view. The key design issue has been the proper treatment of free links in the presence of membrane structures.

Rewrite rules specify reaction between elements of a multiset, but reaction between interlinked elements can be much more efficient (in finding partners) than reaction between unlinked elements.

LMNtal features both channel mobility and process mobility. In other words, it allows dynamic reconfiguration of process structures as well as the migration of nested computation.

1.3 Syntax of LMNtal

1.3.1 Link Names and Atom Names

First of all, we presuppose two syntactic categories:

- *Link Names* (or *link variables*), denoted by X. In the concrete syntax, link names are denoted by identifiers starting with capital letters.
- *Atom Names*, denoted by p. In the concrete syntax, atom names (including numbers and special symbols) are denoted by identifiers different from link names. The name "=" is the only reserved atom name in LMNtal.

1.3.2 Syntax

The two major syntactic categories of LMNtal are processes and process templates. The former is the subject of the language that evolves with program execution. The latter is used in reaction rules and can express *local contexts* of processes, namely contexts within particular cells.

The syntax of LMNtal is given in Figure 1.1. As usual, parentheses ( ) are used to resolve syntactic ambiguities. Commas for molecules connect tighter than the ":-" for rules. P and T have several syntactic conditions, as will be detailed in this section. The part of a process not included in any rule is called the *non-rule part* of the process. Cells can be arbitrarily nested. The part of a cell $\{P\}$ or $\{T\}$ not contained in nested cells is called the *toplevel* of $\{P\}$ or $\{T\}$, respectively.

We can think of a subset of LMNtal, *Flat LMNtal*, that does not allow cell hierarchies. The syntax of Flat LMNtal does not feature the lines with daggers (†).

The rest of this section explains processes, rules and process templates in more detail.

Processes.

A process P must observe the following *Link Condition*:

$$
\begin{array}{lll}
P ::= \mathbf{0} & & \text{(null)} \\
\quad \mid\ p(X_1, \ldots, X_m) & (m \geq 0) & \text{(atom)} \\
\quad \mid\ P, P & & \text{(molecule)} \\
\quad \mid\ \{P\} & & \text{(cell) †} \\
\quad \mid\ T :\!- T & & \text{(rule)} \\
\\
T ::= \mathbf{0} & & \text{(null)} \\
\quad \mid\ p(X_1, \ldots, X_m) & (m \geq 0) & \text{(atom)} \\
\quad \mid\ T, T & & \text{(molecule)} \\
\quad \mid\ \{T\} & & \text{(cell) †} \\
\quad \mid\ T :\!- T & & \text{(rule)} \\
\quad \mid\ \mathtt{@}p & & \text{(rule context) †} \\
\quad \mid\ \$p[X_1, \ldots, X_m \mid A] & & \text{(process context) †} \\
\quad \mid\ p(*X_1, \ldots, *X_m) & (m > 0) & \text{(aggregate) †} \\
\\
A ::= [] & & \text{(empty) †} \\
\quad \mid\ *X & & \text{(bundle) †}
\end{array}
$$

Fig.1.1 Syntax of LMNtal (Lines with daggers (†) are not in Flat LMNtal)

Link Condition: Each link name in the non-rule part of P can occur *at most twice.*

A link name occurring only once in the non-rule part of P represents a *free link* of P. Each of the other link names occurring in P represents a *local link* of P. A *closed process* is a process containing no free links.

Intuitively, $\mathbf{0}$ is an empty process; $p(X_1, \ldots, X_m)$ is an atom with m ordered links; P, P is parallel composition (or multiset union); $\{P\}$ is a process enclosed with the *membrane* { }; and $T :\!- T$ is a rewrite rule for processes.

An atom $X = Y$, called a *connector*, connects one side of the link X and one side of the link Y.

Note that the Link Condition never prevents us from composing two processes P_1 and P_2. When each of P_1 and P_2 satisfies the Link Condition but the composition P_1, P_2 does not, there must be a link name occurring twice in one and at least once in the other. Since the former represents a local link, we can always α-convert it to a fresh link name (Section 1.4.1) to restore the Link Condition. The link names used in rules are not considered in the Link Condition because they are understood to be local to the rules.

Rules and Process Templates.

Rules have the form $T :- T$, where the T's are called *process templates*. The first and the second T are called the left-hand side (LHS) and the right-hand side (RHS), respectively.

Process templates have three additional constructs, namely *rule contexts*, *process contexts*, and *aggregates*. *Contexts* in LMNtal refer to the rest of the entities in the innermost surrounding membrane. Rule contexts are to represent multisets of rules, while process contexts are to represent multisets of cells and atoms.

A process context consists of a name $\$p$ and an argument $[X_1, \ldots, X_m \mid A]$. The argument of a LHS process context specifies the set of free links that the context must have. X_i denotes a specific link if it occurs elsewhere in the LHS and an arbitrary free link if it does not occur in the LHS. The final component A is called a *residual*. A residual of the form $*V$ receives the bundle of zero or more free links other than $X_1, \ldots, X_m$, and a residual `[]` means that there should be no free links other than $X_1, \ldots, X_m$.

An aggregate represents a multiset of atoms with the same name, whose multiplicity coincides with the number of links represented by the argument bundles.

The precise semantics of all these additional contexts will be given in Section 1.4.

Rules have several syntactic side conditions. Firstly, rules must observe the following:

LHS Conditions:

(L1) Rules cannot occur in the LHS of a rule.
(L2) Aggregates cannot occur in the LHS of a rule.
(L3) Rule contexts and process contexts occurring in the LHS of a rule must occur within a cell.

Note that Condition (L1) disallows the decomposition of rules. Condition (L3) means that rule contexts and process contexts deal only with local contexts delimited by membranes.

Secondly, rules must satisfy the following *Occurrence Conditions* on link names and other syntactic constructs:

Occurrence Conditions:

(O1) A link name and a bundle occurring in a rule must occur exactly twice in the rule.
(O2) Link names occurring in the argument of a process context must be distinct.
(O3) Bundles occurring in the LHS of a rule must be distinct.

(O4) A rule context and a process context occurring in a rule must occur exactly once in the LHS and must not occur in another rule occurring inside the rule.

(O5) The toplevel of each cell occurring in the LHS of a rule may have at most one process context and at most one rule context.

Condition (O1) implies that a rule cannot have free links. Condition (O2) is imposed because the link names specify the *set* of free links to be owned by a process matching the process context. Condition (O3) is because a bundle in the LHS of a rule is to receive, rather than compare, a set of free links of the matching process. The "must occur once" condition in Condition (O4) means that a rule context or a process context must receive a multiset of rules or a process upon application of the rule, and the "exactly once" condition means that they cannot be used to compare two contexts. Note that rule contexts and process contexts may occur more than once in the RHS of a rule. Condition (O5) is to ensure that the values received by rule contexts and process contexts are uniquely determined.

Of the links occurring in a rule L :- R, those occurring only in L are consumed links; those occurring only in R are links generated by the rule, and those occurring once in L and once in R are inherited links.

Finally, we introduce several *Consistency Conditions*:

Consistency Conditions:

(C1) The residuals of the process contexts with the same name in a rule must be either all empty (`[]`) or all bundles.

(C2) The arity m of the process contexts with the same name in a rule must coincide.

(C3) The process contexts having the same bundle must have the same name.

(C4) For each aggregate $p(*X_1, \ldots, *X_m)$ $(m > 0)$ in a rule, there must be a process context name $\$q$ and each $*X_i$ must occur as the residual of a process context with the name $\$q$ in the rule.

Example. The rule

```
{exch,$a[X,Y|[]]} :- {$a[Y,X|[]]}
```

satisfies Conditions (C1) and (C2) (Conditions (C3) and (C4) hold vacuously) and says that when a cell contains an atom `exch` and exactly two free links at its toplevel, the two free links are crossed and the atom `exch` is erased. □

Example. The rule

```
{kill,$a[|*X]} :- killed(*X)
```

satisfies Conditions (C3) and (C4) (the other conditions hold vacuously) and says that

(E1)	$\mathbf{0}, P$	$\equiv$	P	
(E2)	P, Q	$\equiv$	Q, P	
(E3)	$P, (Q, R)$	$\equiv$	$(P, Q), R$	
(E4)	P	$\equiv$	$P[Y/X]$	if X is a local link of P
(E5)	$P \equiv P'$	$\Rightarrow$	$P, Q \equiv P', Q$	
(E6)	$P \equiv P'$	$\Rightarrow$	$\{P\} \equiv \{P'\}$	
(E7)	$X = X$	$\equiv$	$\mathbf{0}$	
(E8)	$X = Y$	$\equiv$	$Y = X$	
(E9)	$X = Y,\ P$	$\equiv$	$P[Y/X]$	if P is an atom and X occurs free in P
(E10)	$\{X = Y,\ P\}$	$\equiv$	$X = Y, \{P\}$	if exactly one of X and Y occures free in P

Fig.1.2 Structural congruence on LMNtal processes

when a cell contains an atom `kill` at its toplevel, the cell is erased and each link crossing the membrane is terminated by a unary atom `killed`. □

The above conditions do not allow dynamic composition of rules, but do allow (i) statically determined rules to be spawned dynamically and (ii) the set of rules inside a cell to be copied and migrated to another cell. Thus LMNtal enables the cell-wise compilation of the set of rules while providing certain higher-order features.

1.4 Operational Semantics

We first define structural congruence ($\equiv$) and then the reduction relation ($\longrightarrow$) on processes.

1.4.1 Structural Congruence

We define the relation $\equiv$ on processes as the minimal equivalence relation satisfying the rules shown in Figure 1.2. Two processes related by $\equiv$ are essentially the same and are convertible to each other in zero steps. Here, $[Y/X]$ is a *link substitution* that replaces X with Y.

(E1)–(E3) are the characterization of molecules as multisets. (E4) allows the renaming (α-conversion) of local names. Note that the link Y cannot occur free in P for the Link Condition on $P[Y/X]$ to hold. (E5)–(E6) are structural rules that make $\equiv$ a congruence. (E7)–(E10) are concerned with connectors. (E7) says that a self-absorbed loop is equivalent to $\mathbf{0}$, while (E8) expresses the symmetry of `=`. (E9) is an absorption law of `=`, which says that a connector can be absorbed by another atom (which can again be a connector). Because of the symmetry of $\equiv$, (E9) says that an atom can emit

$$\text{(R1)}\ \frac{P \longrightarrow P'}{P, Q \longrightarrow P', Q} \qquad \text{(R2)}\ \frac{P \longrightarrow P'}{\{P\} \longrightarrow \{P'\}} \qquad \text{(R3)}\ \frac{Q \equiv P \quad P \longrightarrow P' \quad P' \equiv Q'}{Q \longrightarrow Q'}$$

$$\text{(R4)}\quad \{X = Y, P\} \longrightarrow X = Y, \{P\} \quad (X \text{ and } Y \text{ occur free in } \{X = Y, P\})$$

$$\text{(R5)}\quad X = Y, \{P\} \longrightarrow \{X = Y, P\} \quad (X \text{ and } Y \text{ occur free in } P)$$

$$\text{(R6)}\quad T\theta, (T :- U) \longrightarrow U\theta, (T :- U)$$

Fig.1.3 Reduction relation on LMNtal processes

a connector as well. (E10) says that a connector can be moved across a membrane boundary as long as it does not change the number of free links of the membrane.

1.4.2 Reduction Relation

Computation proceeds by rewriting processes using rules collocated in the same "place" of the nested membrane structure.

We define the reduction relation $\longrightarrow$ on processes as the minimal relation satisfying the rules in Figure 1.3. Note that the right-hand side of $\longrightarrow$ must observe the Link Condition of processes.

Of the six rules, (R1)–(R3) are structural rules. (R1) says that reductions can proceed concurrently based on local reducibility conditions. Fine-grained concurrency of LMNtal originates from this rule. (R2) says that computation within a cell can proceed independently of the exterior of the cell. For a cell to evolve autonomously, it must contain its own set of rules. Computation of a cell containing no rules will be controlled by rules outside the cell. (R3) incorporates structural congruence into the reduction relation.

(R4) and (R5) deal with the interaction between connectors and membranes. (R4) says that, when a connector in a cell connects two links both coming from outside, the cell can expel the connector. (R5) says that, when a connector connects two links both entering the same cell, the connector itself can enter that cell.

(R6) is the key rule of LMNtal. The substitution θ is to represent what process (or multiset of rules) has been received by each process context (or rule context), respectively, and what multiset of atoms each aggregate represents. In Flat LMNtal, θ becomes unnecessary and (R6) is simplified to

$$\text{(R6')}\quad T, (T :- U) \longrightarrow U, (T :- U).$$

(R6') describes the reaction between a process and a rule not separated by membranes.

Matching between a process and the LHS of a rule under (R6') should generally

be done by α-converting the rule using (E4) and (R3). The whole resulting process, namely $U, (T :- U)$ and its surrounding context, should observe the Link Condition, but this can always be achieved by α-converting $T :- U$ before use so that the local links in U won't cause name crashes with the context.

The substitution θ in (R6) is represented as a finite set of *substitution elements* of the form β_i/α_i (meaning that α_i is replaced by β_i), and should satisfy the following three conditions.

1. The domain of θ is the set of all rule contexts, process contexts and aggregates occurring in the LHS T or in the non-rule part of the RHS U.
2. For each rule context $@p$ in T, θ must contain $P/@p$, where P is a sequence of rules.
3. We assume that the occurrences of the process context name $\$p$ in the RHS U are indexed as $\$p_1$, $\$p_2$, ..., and that the function v is a one-to-one mapping from link names and natural numbers to link names.
   For each process context $\$p[X_1, \ldots, X_m \mid A]$ in T, the following (i)–(iii) hold, where P is a process whose free links are $\{X_1, \ldots, X_{m+n}\}$ (if $A = []$ then $n = 0$; otherwise $n \geq 0$), whose local links are $\{Z_1, \ldots, Z_\ell\}$, and which has no rules outside cells.
   (i) If $A = []$ then
      (a) $P/\$p[X_1, \ldots, X_m] \in \theta$
      (b) For $\$p[Y_1, \ldots, Y_m]$ with the number h in the RHS U,
      $$P[v(Z_1, h)/Z_1, \ldots, v(Z_\ell, h)/Z_\ell, Y_1/X_1, \ldots, Y_m/X_m] \\ / \$p[Y_1, \ldots, Y_m] \in \theta$$
   (ii) If $A = *V$ then
      (a) $P/\$p[X_1, \ldots, X_m \mid *V] \in \theta$
      (b) $v(V, i) = X_{m+i}$ for $1 \leq i \leq n$
      (c) For $\$p[Y_1, \ldots, Y_m \mid *W]$ with the number h in the RHS U,
      $$P[v(Z_1, h)/Z_1, \ldots, v(Z_\ell, h)/Z_\ell, Y_1/X_1, \ldots, Y_m/X_m, \\ v(W, 1)/X_{m+1}, \ldots, v(W, n)/X_{m+n}] \\ / \$p[Y_1, \ldots, Y_m \mid *W] \in \theta$$
      (d) For each $q(*V_1, \ldots, *V_k)$ in the non-rule part of U such that some V_i is V,
      $$(q(v(V_1, 1), \ldots, v(V_k, 1)), \ \ldots, q(v(V_1, n), \ldots, v(V_k, n))) \\ / q(*V_1, \ldots, *V_k) \in \theta$$
   (iii) a free link of T occurring in an atom (i.e., not in process contexts) doesn't occur in P.

Suppose the LHS of a rule contains a process context $p[X_1, \ldots, X_m | *V]$. When the RHS contains a process context of the same name, say $\$p[Y_1, \ldots, Y_m | *W]$, a process isomorphic to the process matched by the corresponding process context in the LHS is created. Its free links corresponding to $X_1, \ldots, X_m$ are connected to $Y_1, \ldots, Y_m$, respectively, and the free links corresponding to $*V$ are connected to the links represented by $*W$.

An aggregate $p(*V_1, \ldots, *V_m)$ represents as many copies of the m-ary atom p as the number of links denoted by the bundle $*V_i$. Each $*V_i$ must have the same origin with respect to the process context name (Condition (C4)); in other words, the other occurrences of the $*V_i$'s must all appear in process contexts with the same name. Condition (O4) implies that exactly one of $*V_1, \ldots, *V_m$ occurs in the LHS of a rule.

Example. The rule

```
kill(S), {i(S),$p[|*P]} :- killed(*P)
```

can reduce the process

```
kill(S), {i(S),a(X),b(Y,Z),c(Z,U)},
```

by letting `$p[|*P]` receive `a(X),b(Y,Z),c(Z)` , and the process is reduced to

```
killed(X), killed(Y).
```

In this example, the membrane is used to delimit the process structure to be controlled, and the tag `i( )` is attached to the message channel from outside the cell. The above rule says that, when a `kill` message is sent through the channel `S`, the target cell is deleted and each free link owned by the cell is terminated by an atom `killed`. □

Example. Consider the process

```
cp(S,S1,S2), {i(S),a(X),b(Y,Z),c(Z)}
```

and the rule

```
cp(S,S1,S2), {i(S),$p[|*P]} :-
    {i(S1),$p[|*P1]}, {i(S2),$p[|*P2]}, cp(*P,*P1,*P2) .
```

Then the process is reduced to

```
{i(S1),a(X1),b(Y1,Z1),c(Z1)}, {i(S2),a(X2),b(Y2,Z2),c(Z2)},
cp(X,X1,X2), cp(Y,Y1,Y2) .
```

The `cp` message makes two copies of the target cell and connects the free links of the copied cells (`X1`, `Y1`, `X2`, `X2`) and the original free links (`X`, `Y`) using ternary `cp` atoms (Figure 1.4). □

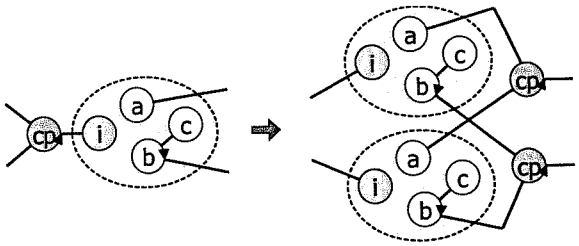


Fig.1.4 Cell copying using process contexts and aggregates

1.5 Program Examples

1.5.1 Concatenating Lists

The skeleton of a linear list can be represented, using element processes c(ons) and a terminal process n(il), as $\mathtt{c}(A_1, X_1, X_0)$, ..., $\mathtt{c}(A_n, X_n, X_{n-1})$, $\mathtt{n}(X_n)$. Here, A_i is the link to the ith element and X_0 is the link to the whole list (from somebody else). This corresponds to a list formed by the constraints $X_0 = \mathtt{c}(A_1; X_1)$, ..., $X_{n-1} = \mathtt{c}(A_n, X_n)$, $X_n = \mathtt{n}$ in (constraint) logic programming languages, except that the LMNtal list is a resource rather than a value. Two lists can be concatenated using the following two rules:

```
append(X0,Y,Z0), c(A,X,X0) :- c(A,Z,Z0), append(X,Y,Z)
    append(X0,Y,Z0), n(X0) :- Y=Z0
```

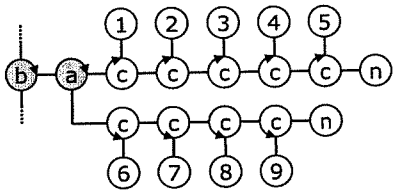
Figure 1.5 shows a graphical representation of the append program and its execution.

The above program has clear correspondence with append in GHC:

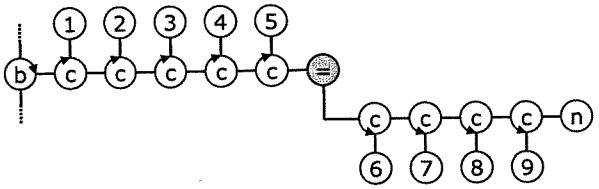
```
append(X0,Y,Z0) :- X0=c(A,X) | Z0=c(A,Z), append(X,Y,Z).
append(X0,Y,Z0) :- X0=n | Y=Z0.
```

but LMNtal has eliminated syntactic distinction between processes and data.

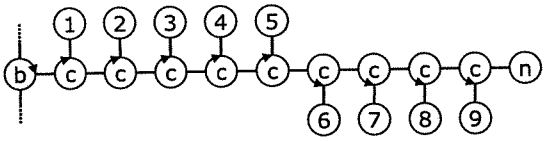
The above program resembles append in Interaction Nets [10]. Indeed, Lafont writes "our rules are clearly reminiscent of clauses in *logic programming*, especially in the use of variables (see the example of difference-lists), and our proposal could be related to PARLOG or GHC" [10]. LMNtal generalizes Interaction Nets by removing the restriction to binary interaction and allowing hierarchical processes.



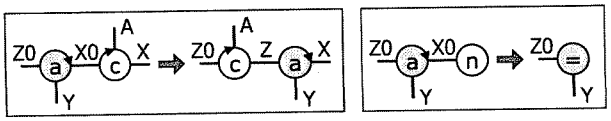
(a) Initial state



(b) Final state 1



(c) Final state 2



(d) Rewrite rules

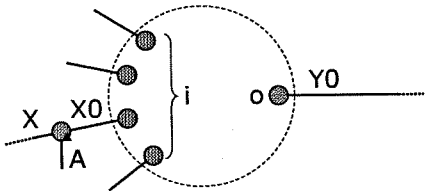
Fig.1.5 List concatenation

1.5.2 Stream Merging

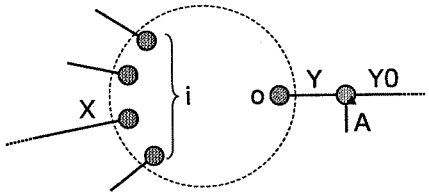
As in logic programming, streams can be represented as lists of messages, and n-to-1 communication by stream merging can be programmed as follows:

```
{i(X0),o(Y0),$p[|*Z]}, c(A,X,X0) :-
                    c(A,Y,Y0), {i(X),o(Y),$p[|*Z]}
```

Here, the membrane `{ }` of the left-hand side records n (≥ 1) input streams with the name `i` and one output stream with the name `o`. The process context `$p[|*Z]` is to match the rest of the input streams and pass them to the RHS. Figure 1.6 shows a redex to which the above rewrite rule is applicable and the result of reduction.

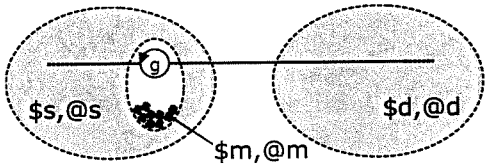


(a)

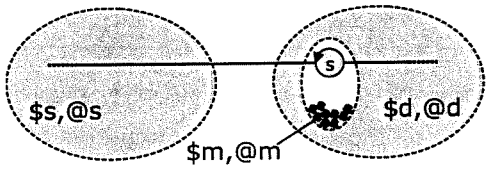


(b)

Fig.1.6 Multiway stream merging



(a) before migration



(b) after migration

Fig.1.7 Process migration

1.5.3 Process Migration

Consider two cells that share a communication link. Suppose they run independently using individual sets of reaction rules most of the time but sometimes migrate processes to each other through the link. The rule for migration is given in an upper layer.

It is the rôle of the upper layer to determine the protocol of process migration, while the cells "hook" processes to be migrated on the communication link according to the protocol. Here we assume that the innermost cell containing $g(S, D)$ is to be migrated by the upper layer, where S and D are the source and the destination sides of the communication link, respectively (Figure 1.7).

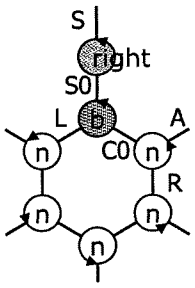


Fig.1.8 Cyclic data structures

```
{$s[S0|*S], @s, {g(S0,D0),$m[|*M],@m}}, {$d[D0|*D], @d} :-
        {$s[S|*S], @s}, {{s(S,D),$m[|*M],@m}, $d[D|*D], @d}
```

When `@m` is non-empty, the rule acts as active process migration; otherwise it acts as data migration. Note that the communication link between the source and the destination processes changes after migration. This is an important characteristic of logical links. The membrane delimiting migrated resources can be removed at the destination site.

1.5.4 Cyclic Data Structures

Most declarative languages handle lists and trees elegantly but cyclic data structures awkwardly. This is not the case with LMNtal. In LMNtal, a bidirectional circular buffer with n elements can be represented as

$$\mathtt{b}(S, X_n, X_0), \mathtt{n}(A_1, X_0, X_1), \ldots, \mathtt{n}(A_n, X_{n-1}, X_n),$$

where `b` is a header process, the A_i's are links to the elements, and S is the link from the client process. Operations on the buffer are sent through S as messages such as `left`, `right` and `put` (Figure 1.8). The reaction rules between messages and the buffer can be defined as follows:

```
 left(S,S0), n(A,L,C0), b(S0,C0,R) :- b(S,L,C), n(A,C,R)
right(S,S0), b(S0,L,C0), n(A,C0,R) :- n(A,L,C), b(S,C,R)
          put(A,S,S0), b(S0,L,R) :- n(A,L,C), b(S,C,R)
...
```

Shape Types [8] are another attempt to facilitate manipulation of dynamic data structures. Interestingly, Shape Types took a dual approach, namely they used variables to represent graph nodes and names to represent links.

1.6 Concluding Remarks

We have presented a concise language model LMNtal, which has logical links, multisets, nested nodes and transformation as its "big four" elements. LMNtal was inspired by communication using logical variables, and its principal goal as a concurrent programming language has been to unify processes, messages, and data. There are many languages and computation models that support multisets and/or graph rewriting, but LMNtal is unique in the design of link handling in the presence of membrane hierarchies.

CHR is another multiset rewriting language that features logical variables. While Flat LMNtal can be thought of as a linear fragment of CHR, LMNtal and CHR have many differences in the use of logical variables, control of reactions, intended applications, and so on. It is a challenging research topic to embed CHR into LMNtal.

Both P-systems and LMNtal feature membrane hierarchies and rewrite rules local to membranes. One apparent difference between P-systems and LMNtal is that LMNtal features logical links as another key construct. We can think of a fragment of LMNtal that allows only nullary atoms (atoms without links). This fragment is somewhat close to P-systems, but one important design criteria of LMNtal has been that computation inside a cell cannot affect its environment, that is, a cell cannot export any process by itself. Instead, a cell communicates with its environment by spawning (within the cell) particular processes that can be recognized and handled by the rules in the environment.

We have released a prototype implementation in Java[*1]. It features

- a construct for detecting inactive cells,
- built-in number types,
- the notion of type constraints for typechecking and comparison of numbers and symbols, and
- foreign language interface,

in addition to most of the constructs described in this chapter.

Many things remain to be done. The most important issue in the language design is to equip it with useful type systems. We believe that many useful properties, for instance shapes formed by processes and links, the directionality of links (i.e., whether links can be implemented as unidirectional pointers), and properties about free links of cells, can be guaranteed statically using type systems. Challenging topics in our implementation project include compact representation of processes and links, optimizing compilation of reaction rules, and parallel and distributed implementation. Since LMN-

[*1] `http://www.ueda.info.waseda.ac.jp/lmntal/`

tal is intended to unify various existing computational models, relating LMNtal to them by embedding them into LMNtal is another important research subject. When the embeddings are simple enough, LMNtal will be able to act as a common implementation language of various models of computation.

Last but not least, we should accumulate applications. Some interesting applications other than ordinary concurrent computation are graph algorithms, multi-agent systems, Web services, and programming by self-organization.

References

[1] Andries, M. *et al.*, Graph Transformation for Specification and Programming. *Sci. Comput. Program.*, Vol. 34, No. 1 (1999), pp. 1–54.

[2] Banâtre, J.-P. and Le Métayer, D., Programming by Multiset Transformation. *Commun. ACM*, Vol. 35, No. 1 (1993), pp. 98–111.

[3] Berry, G. and Boudol, G., The Chemical Abstract Machine. In *Proc. POPL'90*, ACM, 1990, pp. 81–94.

[4] Cardelli, L. and Gordon, A. D. : Mobile Ambients, in *Foundations of Software Science and Computational Structures*, Nivat, M. (ed.), LNCS 1378, Springer-Verlag, 1998, pp. 140–155.

[5] Castagna, G., Vitek, J. and Zappa Nardelli, F., The Seal Calculus. *Information and Computation*, Vol. 201, No. 1 (2005), pp. 1–54.

[6] Drewes, F., Hoffmann, B. and Plump, D., Hierarchical Graph Transformation. *J. Comput. Syst. Sci.*, Vol. 64, No. 2 (2002), pp. 249–283.

[7] Engels, G. and Schürr, A., Encapsulated Hierarchical Graphs, Graph Types, and Meta Types. *Electronic Notes in Theor. Comput. Sci.*, Vol. 1 (1995), pp. 75–84.

[8] Fradet, P. and Le Métayer, D., Shape Types. In *Proc. POPL'97*, ACM, 1997, pp. 27–39.

[9] Frühwirth, T., Theory and Practice of Constraint Handling Rules. *J. Logic Programming*, Vol. 37, No. 1–3 (1998), pp. 95–138.

[10] Lafont, Y., Interaction Nets. In *Proc. POPL'90*, ACM, 1990, pp. 95–108.

[11] Miller, D. : Overview of Linear Logic Programming. In *Linear Logic in Computer Science*, Ehrhard, T., Girard, J.-Y., Ruet, P. and Scott, P. (eds.), Cambridge University Press, 2004, pp. 119–150.

[12] Milner, R., Bigraphical Reactive Systems. In *Proc. CONCUR 2001*, LNCS 2154, Springer, 2001, pp. 16–35.

[13] Păun, Gh., Computing with Membranes. *J. Comput. Syst. Sci.*, Vol. 61, No. 1 (2000), pp. 108–143.

[14] Saraswat, V. A., Kahn, K. and Levy, J., Janus: A Step Towards Distributed Constraint Programming. In *Proc. 1990 North American Conf. on Logic Programming*,

MIT Press, 1990, pp. 431–446.

[15] Schmitt, A. and Stefani, J.-B., The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *Proc. Int. Workshop on Global Computing.*, LNCS 3267, Springer, 2005, pp. 146–178.

[16] Ueda, K., Concurrent Logic/Constraint Programming: The Next 10 Years. In *The Logic Programming Paradigm: A 25-Year Perspective*, Apt, K. R., Marek, V. W., Truszczynski M., and Warren D. S. (eds.), Springer-Verlag, 1999, pp. 53–71.

[17] Ueda, K., Resource-Passing Concurrent Programming. In *Proc. TACS 2001*, LNCS 2215, Springer, 2001, pp. 95–126.

[18] Ueda, K. and Kato, N., Programming with Logical Links: Design of the LMNtal Language. In *Proc. Third Asian Workshop on Programming Languages and Systems (APLAS 2002)*, 2002, pp. 115–126.

[19] Ueda, K. and Kato, N., The Language Model LMNtal. *Computer Software*, Vol. 21, No. 2 (2004), pp. 44–60 (in Japanese).

[20] Ueda, K. and Kato, N., LMNtal: a language model with links and membranes. In *Proc. Fifth Int. Workshop on Membrane Computing (WMC 2004)*, LNCS 3365, Springer, 2005, pp. 110–125.

第 2 章

LMNtal の統合プログラミング言語への展開

LMNtal は階層グラフ書換えに基づく単純な言語モデルであり，接続構造の表現に論理変数を，階層構造の表現に膜を用いることを特徴としている．LMNtal は，多重集合や並行処理やモビリティなどの概念を持つさまざまな計算モデルの統合を目指すと同時に，階層グラフ書換えに基づく実用的なプログラミング言語を提供してその有用性を示すことを重要な目標としている．本章の目的は，プログラミング言語としての LMNtal の諸機能を紹介し，その記述力を多くの例題を用いて示すことである．我々は，算術，ルール適用制御，モジュール，他言語インタフェースなどの重要機能を階層グラフ書換えモデルの中に組み込む方法を設計し実装した．記述力の検証のために λ 計算，π 計算，ambient 計算，CHR などの代表的な関連計算モデルのエンコーディングを行い，それらを実際に処理系上で動作させることに成功した．

2.1 はじめに

LMNtal 言語モデルの開発は，計算理論とプログラミング言語における以下の二つのグランドチャレンジを動機としている．一つはさまざまな計算のパラダイム，特に並行計算と多重集合書換えに基づく計算とを統合する計算モデルを確立することである．もう一つは，広域計算と極小規模計算の両方向に発達しつつあるさまざまな計算プラットフォームに対応できる実際的なプログラミング言語を設計，実装することである．

LMNtal は，制約に基づく並行計算 (並行制約プログラミング)[10] と Constraint Handling Rules (CHR) [3] とを統合する試みの中から着想を得て設計した言語モデルである．LMNtal の基本構成要素はアトムであるが，アトムを膜とリンクの二つの手段で構造化して階層グラフを表現することができるのが特徴である．膜はアトムの多重集合 (要素の個数の概念を持つ集合) を構成するが，階層構造を形成してさらにそれを動的再構成することが可能である．リンクはアトムどうしを一対一で接続する．階層構造と接続構造とは，計算

システムにおいてはもちろんのこと，社会組織から生体に至るまで，あらゆる場面で見ることのできる構造化のしくみであり，しかも両者はしばしば併存する．そこで階層構造と接続構造の両方をサポートする計算や表現の枠組みを構築したいというのが LMNtal の哲学的背景である．

LMNtal の先祖である制約に基づく並行計算と CHR は論理プログラミングパラダイムの中で開花した枠組みである．LMNtal も論理プログラミングと密接な関係を持ち，たとえばリンクは論理変数を非常に限定された形，具体的には線形な (つまりちょうど 2 回出現する) 局所名として用いたものとみなすことができる．しかし，プログラミングの観点からは LMNtal は階層グラフを対象とする書換え系もしくはリンクを備えた多重集合書換え言語とみなすのがもっとも自然であり，本章でもその立場から LMNtal を扱うこととする．

LMNtal は広範な用途をもつにもかかわらず非常に単純な言語であり，予備知識がほとんどなくても使い始めることができる．これは，LMNtal の計算が図的に解釈可能であることと深く関連している．我々は，並行計算や動的なデータ構造などの概念を説明したり理解したりするときに図 (ダイアグラム) を援用するが，LMNtal は対象の図的表現をそのまま扱うことのできる言語と見なすことができる．

LMNtal は，2002 年に基本設計を固めたのち，理論と実践の両面から設計レビューのプロセスを行ってきた．理論面での大きな挑戦は，リンクが形成する接続構造と，(リンクが貫くかもしれない) 膜が形成する階層構造との間の相互作用をきちんと取り扱う操作的意味論を設計することであった．実践面では，言語設計の初期の段階から，設計レビューを重要な目的として本格的な実装研究に取り組んできた．実装を行うことによって，言語機能を構成的に理解したり，言語設計の不備を発見したり，プログラミング経験を蓄えたりすることが可能になるからである．また，理論上は本質的でないが実用上重要な言語機能を洗い出すこともできる．

本章の目的と貢献は大きく分けて次の二つである．一つは，計算モデルとしての LMNtal を提案 [11][12] して以来，上記の検討を通じて確立した「プログラミング言語としての LMNtal」の諸機能を紹介し，その設計について論じることである．階層グラフ書換えモデルをプログラミング言語として実用化する試みは先例がない．LMNtal ではプロセス (手続き) とデータも統一的に扱うが，そのような言語の実用性はこれまで明らかにされてこなかった．そこで我々は，算術，ルール適用制御，モジュール，他言語インタフェースなどの重要機能を設計し実装した．

二つめの目的と貢献は，LMNtal による多様なプログラム例を示すことである．LMNtal の応用分野は多岐にわたるが，本章では，基本的かつ代表的な計算モデルである λ 計算，π 計算，ambient 計算，CHR の LMNtal によるエンコーディングを中心にとりあげる．多様な計算モデルを LMNtal にエンコードすることは，(i) 計算モデル間の関係が明確になる，(ii) LMNtal の記述能力の検証に役立つ，(iii) LMNtal の統合モデルとしての有用性が判断できる，(iv) さまざまな理論計算モデルの実装を提供する，など多くの意義がある．

本章に示すプログラム例はすべて，我々のプロジェクトで開発したLMNtal処理系で動作確認済である．このLMNtal処理系はWeb上で公開されており[*1]，Javaプラットフォームの上で稼働する．

本章の構成は以下の通りである．第2節ではLMNtalの基本概念について実例を用いながら説明する．第3節では我々が開発したLMNtal処理系の機能概要について述べる．第4節ではLMNtalを実用言語として確立するために我々が行った言語拡張について詳しく述べる．第5節では代表的な他の計算モデルのLMNtalへのエンコードと動作例を示し，またそれを通じてLMNtalの記述能力を検討する．第6節はまとめと今後の課題である．

LMNtalの形式的操作的意味論など，言語モデルとしてのLMNtalに関する技術的詳細や関連研究については，文献[11][12]や第1章を参照してほしい．本章では上記文献でカバーしていない実際的側面を扱う．また，本章は処理系の実現方法を論じることは目的とせず，その詳細は第3章に譲ることとする．

2.2 LMNtalの基本的枠組

本節では簡単な例題を紹介しつつ，LMNtalの基本的な枠組について述べる．

2.2.1 LMNtalの基本構成要素

LMNtalの基本用語を手短に定義する．(m 価の)**アトム**はアトム名と m (≥ 0) 個の順序づけられた引数からなる．それぞれの引数は (自分自身または他の) アトムの引数につながるリンクの端点である．**リンク**はリンク名を用いて表記し，同じリンク名をもつアトムの引数どうしが相互接続されていることを表す．LMNtalでは2.2.4節に述べる構文条件 (リンク条件) を設けて相互接続を一対一に制限し，アトムとリンクの両者が形成する構造が無向グラフ構造を表すようにしている．リンクは英大文字から始まる名前，アトムはリンクと区別できる名前で表記する．一重引用符または二重引用符で囲んだ名前もアトム名として利用できる．予約アトム名 = は**コネクタ**と呼ばれ，アトム X=Y はリンク X の一端と Y の一端とを接続する機能をもつ．

アトムの多重集合は**膜** {...} で囲むことができ，囲ったものを**セル**と呼ぶ．膜は入れ子構造をなすことができ，アトム，リンク，膜によって階層グラフが形成される．この階層グラフ構造をLMNtalでは**プロセス**と呼ぶ．図2.1に，LMNtal処理系の可視化機能を利用して表示した階層グラフの例を挙げる．

階層グラフ構造の書換え規則を**ルール**と呼び，次の構文で表記する．

$$Head :- Body$$

Head (ルール左辺) はルールの所属する階層から見た書き換えるべき階層グラフ構造のテン

[*1] http://www.ueda.info.waseda.ac.jp/lmntal/

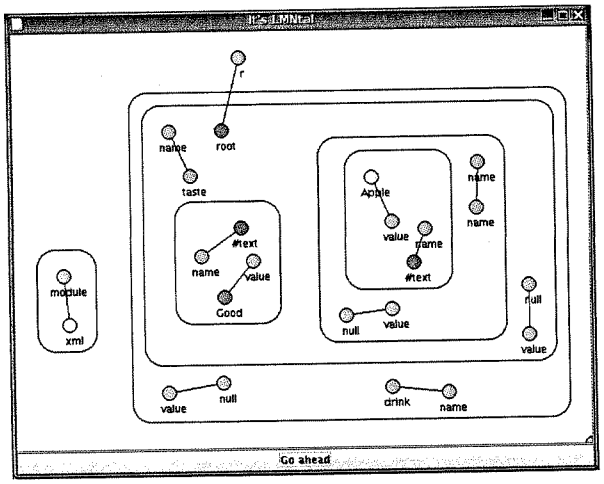


図 2.1　階層グラフ

プレートであり，*Body* (ルール右辺) は書換え後の階層グラフ構造のテンプレートである．

各膜の中にはルールが複数個あってもよく，その多重集合を**ルールセット**と呼ぶ．

2.2.2　例: 階層的多重集合の書換え

最初の実例として，Gamma モデル [1] に代表される多重集合の書換えを LMNtal で記述，実行する例を紹介する．LMNtal システムを対話モードで実行すると次のようになる．

```
$ lmntal
LMNtal version 0.80.20060319
Type :h to see help.
Type :q to quit.

# 1,1,1, {1,1,1,1,1,1, {1,1}, (1,1:-2)}

1,1,1, {2,2,1,{1,1}, @601}
# {out,a,b,c},d,{e,f},({out,$p}:-$p).

d,a,c,b, {f,e}, @603
```

最初の例は，2 個の 1 から 1 個の 2 への多重集合書換えである．膜は階層的多重集合を形成し，ルール `1,1:-2` は，それが所属する膜のトップレベルのアトム (この例では 5 個の 1) に対してのみ適用される．

ルールは，そのルールが属する膜の外側に影響を及ぼすことはできないが，ルールが属する膜の内側のセルを書換えの対象とすることはできる．2 番目の例に現れるルール `{out,$p}:-$p` は，アトム `out` をもつ膜を消去し，`out` 以外の内容物を膜の外に持ち出す

```
dome(L0,L1,L2,L3,L4,L5,L6,L7,L8,L9) :-
    p(T0,T1,T2,T3,T4), p(L0,L1,H0,T0,H4),
    p(L2,L3,H1,T1,H0), p(L4,L5,H2,T2,H1),
    p(L6,L7,H3,T3,H2), p(L8,L9,H4,T4,H3).

p(L0,L1,L2,L3,L4) :-
    c(L0,X1,X0), c(L1,X2,X1), c(L2,X3,X2),
    c(L3,X4,X3), c(L4,X0,X4).

 /* top half */
dome(E0,E1,E2,E3,E4,E5,E6,E7,E8,E9).
 /* bottom half */
dome(E0,E9,E8,E7,E6,E5,E4,E3,E2,E1).
```

図 2.2　C_{60} プログラム

ことを指定している．$p はプロセス文脈と呼び，それが属する膜内の明示されていないプロセスにマッチする．最後の行から推察できるように，計算結果を構成する階層的多重集合の各要素は処理系に依存した順序で出力される．

アットマークで始まるシンボル (@601, @603) はコンパイルされたルールセットを表している．

2.2.3　例: グラフ構造生成

LMNtal の表現力を示す二つめの例題として，密な結合構造をもつグラフ構造の生成をとりあげる．宣言型言語はリストや木構造の扱いを得意としているが，循環構造や密に結合したデータ構造の扱いを積極的にサポートする言語は多くない．いくつかの Prolog 処理系が有理木構造 (有限のグラフ表現をもった木構造) とその単一化をサポートしているのが数少ない例外である．

図 2.2 は，フラーレン構造 (C_{60}) を生成するプログラムであり，2 個のルールと 2 個の初期アトムだけからなる．ルールや初期アトムはこの例のようにピリオド止めにして並べることもできる．

C_{60} 構造は，正二十面体を作った後，その 12 個の頂点 (五面頂点) から五角形を削り出すことで生成できるが，プログラムもそのような構成になっている．

正二十面体は，対極にある二つの頂点が北極と南極になるようにして眺めると，北と南にそれぞれ 5 個の正三角形からなるドームを配置して，それらを赤道上の 10 個の正三角形 (または 10 本のリンク) で接続した構造であると解釈できる．最初のルールはドームを作るためのルールであり，相手方のドームとつなげるための 10 本のリンクを引数にとる．最後

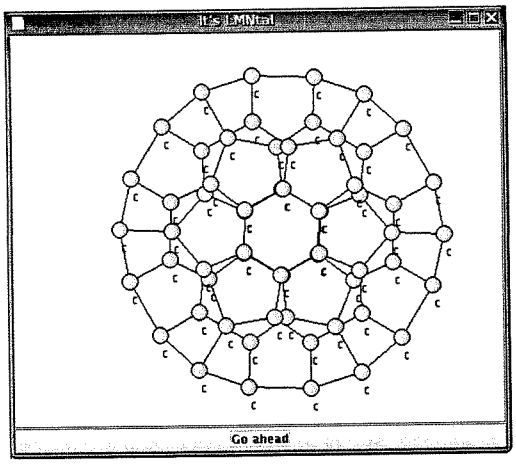


図 2.3　C_{60} 構造

の二つのアトムがドーム生成ルールの呼出しであり，リンク E0〜E9 によって二つのドームを接続している．

二つめのルールは正二十面体の各頂点を削って 5 個ずつの新たな三面頂点を生成することによって，フラーレン構造を生成するルールである．

本プログラムを実行して得られるテキスト出力は複雑であるが，LMNtal システムの可視化オプション (-g) を使ってグラフ構造をレンダリングすると，図 2.3 のような構造を得ることができる．

2.2.4　LMNtal の基本構文

LMNtal の基本構文は，X_i を**リンク名**，p を**アトム名**として図 2.4 のように定義される．P は**プロセス**である．T はプロセスの書換え規則の表現に用いる**プロセステンプレート**であり，局所文脈 (特定のセルの内部での文脈) を扱う機能をもつ．

0 は中身のないプロセス，$p(X_1,\ldots,X_m)$ は m 価アトム，P,P はプロセスの並列合成，$\{P\}$ は**膜** { } によってグループ化されたプロセス，ルール T :- T はプロセスの書換え規則である．

プロセスは，**同じリンク名が 2 回を超えて出現してはならない**という**リンク条件**を満たさなければならない．さらに個々のルールの各リンク名は，そのルール内にちょうど 2 回出現しなければならない．プロセス P に 1 回だけ出現するリンク名は P の**自由リンク**(P の外部につながるリンク) を表し，P に出現するそれ以外のリンク名は P の**局所リンク**を表す．

リンクは 2 つのアトムを一対一で接続する．局所リンク名は局所的接続関係を示すためのもので，その名称自体は重要でない．そのため，局所リンク名の 2 つの出現を同時に別の新しい名前に置き換えることができる (α 変換)．

$$
\begin{array}{lll}
P ::= \mathbf{0} & & \text{(空)} \\
\quad | \ p(X_1, \ldots, X_m)\ (m \geq 0) & & \text{(アトム)} \\
\quad | \ P, P & & \text{(分子)} \\
\quad | \ \{P\} & & \text{(セル)} \\
\quad | \ T :\!- T & & \text{(ルール)} \\
T ::= \mathbf{0} & & \text{(空)} \\
\quad | \ p(X_1, \ldots, X_m)\ (m \geq 0) & & \text{(アトム)} \\
\quad | \ T, T & & \text{(分子)} \\
\quad | \ \{T\} & & \text{(セル)} \\
\quad | \ T :\!- T & & \text{(ルール)} \\
\quad | \ @p & & \text{(ルール文脈)} \\
\quad | \ \$p[X_1, \ldots, X_m \mid A]\ (m \geq 0) & & \text{(プロセス文脈)} \\
\quad | \ p(*X_1, \ldots, *X_m)\ (m > 0) & & \text{(アトム集団)} \\
A ::= [] & & \text{(空)} \\
\quad | \ *X & & \text{(リンク束)}
\end{array}
$$

図 2.4　LMNtal の基本構文

2.2.2 節で例示したように，ルールを膜に入れることができ，膜は計算の局所化のために利用できる．ルールは，そのルールが所属する膜やその子孫膜の内容を書換えの対象とすることができるが，親膜の内容を書き換えることはできない．

ルール文脈は膜の中のすべてのルールの多重集合とマッチし，プロセス文脈は膜の中のルール以外のプロセスのうち，明示的に指定されていないもの全体とマッチする．個々のルール中の文脈の出現はいくつかの構文条件を満たさなければならない [12]．アトム，ルール文脈，プロセス文脈は，同じ名前 p を持っていても互いに無関係である．

プロセス文脈の引数は，自由リンクの出現に関する制約条件を指定するものである．ルール左辺のプロセス文脈 $\$p[X_1, \ldots, X_m \mid A]$ の引数 $X_1, \ldots, X_m$ は，その文脈が持っていなければならない自由リンクを指定しており，これをプロセス文脈の**明示的な自由リンク**と言う．剰余引数 A が $*X$ の形の場合，$*X$ はその文脈が持つ $X_1, \ldots, X_m$ 以外の 0 本以上の自由リンク (**明示的でない自由リンク**と言う) の束を表し，[] の場合は $X_1, \ldots, X_m$ 以外に自由リンクがないことを表す．

2.2.5　省略構文

階層グラフ構造やルールが簡潔に記述できるように，いくつかの省略構文を用意している．

1. 0 価アトムの括弧 () は省略できる．たとえば a() は a と略記できる．
2. 1 価アトム '+'(X) は +X，'-'(X) は -X と略記できる．
3. アトム a の第 k 引数として，(リンク名のかわりに) 最終引数を省略したアトム b を書くと，a の第 k 引数と b の最終引数とがリンク接続されているものとみなす．たとえば x(y(z)) は x(X),y(Z,X),z(Z) と等しい．
   木構造をなすグラフを LMNtal で表現するときは，n 分木のノードを，その参照者からのリンクの名前を最後に追加した $n+1$ 引数のアトムとして表現するのが標準であるが，本略記法は，そのような形式の木構造グラフを通常の項 (term) の形式で記述するためのものである．
4. P をプロセス，A をリンク名とするとき，p(..., {P},...) は p(...,A,...), {+A, P} の略記とみなす．p はセル {+A, P} を参照するアトムとみなすことができ，参照を表すリンク A を終端するのに 1 価アトム名 '+' が標準的に使われる．
5. プロセス文脈のリンク束が空のときは|[] を省略してよい．またそのように略記したプロセス文脈に 3. の略記法を援用して，p(...,A,...), $q[$A$] を p(...,q,...) と略記してよい．
6. ルールに $p[|*X] の形の同一のプロセス文脈が 2 回出現するならば，両方を同時に $p と略記してよい．

上記の略記法の帰結として，f(3) と 3(f) はどちらも 1 価の f および 1 価の 3 からなる同一のプロセス (f(X),3(X)) を表す．さらに LMNtal は，相互接続されたアトムとコネクタに関して

$$p(\ldots,X,\ldots),X=Y \equiv p(\ldots,Y,\ldots)$$

という合同規則 (0 ステップの相互変換規則) をもっている．これを用いると，(f(X),3(X)) は (f(X),3(Y),X=Y) あるいは (f(X),3(Y),Y=X) と展開でき，それぞれに略記法を適用すると f=3 および 3=f が得られる．これらはすべて同じプロセスを表す．このように，= は等号ではなくてアトム間の相互接続を表す記号として多用される．

また，リストの表現には Prolog の記法を利用することができる．たとえば res=[a,b,c] はリスト [a,b,c] と 1 価アトム res とが接続された構造を表し，X=[a,b,c|X] は a, b, c からなる環状リストを表す．

2.3 処理系の概要

公開している LMNtal 処理系は約 49,500 行の Java コードから成り，その中核をなす実行時処理系が 17,300 行，コンパイラが 16,700 行である．Eclipse と CVS を用いてチーム開発されてきた．開発にかかわった人数は 18 名である．現時点でアトム集団機能 (詳細は文献 [12] を参照) を除く全機能と次節に述べる拡張機能を実装しており，アトム集団機能も代替機能を標準ライブラリで提供している (2.4.5 節)．

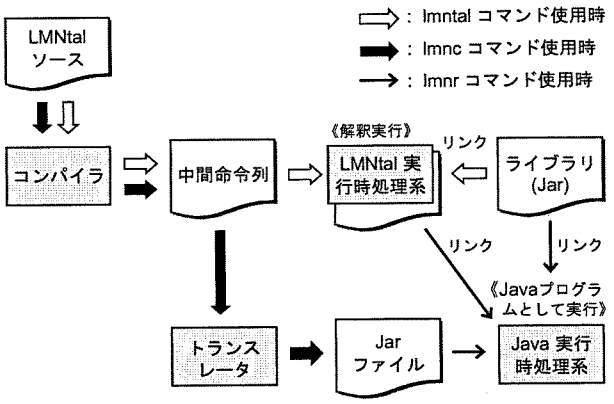


図 2.5　コンパイル・実行の手順

階層グラフ書換えモデルの実装は，逐次実装に限定すれば困難さが大きく減るが，本処理系の特徴は並列分散処理への拡張を考慮した階層グラフの非同期書換えアルゴリズムを設計し実装した点である [14]．膜を貫くリンクの実装などの，実装上のノウハウの多くはプロトタイプ処理系 [15] から受け継いでいる．LMNtal プログラム開発環境として Eclipse 用プラグインの開発も行われている．

2.3.1　処理系の動作

`lmntal` コマンドを入力してシステムを起動すると，LMNtal ソースコードを中間命令までコンパイルし，インタープリタが中間命令を解釈実行する．

また `lmnc` コマンドを入力してコンパイラを起動すると，LMNtal ソースコードから中間命令を経由して Java コードが生成される．生成された jar ファイルを指定して `lmnr` コマンドを実行することにより LMNtal プログラムが実行される．中間言語トランスレータを実装したことによって，インタープリタが中間命令を解釈実行した場合と比べて実行速度が 2 倍から 20 倍 (整列や多重集合生成など数個のプログラムの測定値) 向上した．

LMNtal プログラムを指定せずに `lmntal` コマンドを起動した場合は対話モード (read-eval-print-loop) となり，1 行ごとに中間命令へのコンパイルとその解釈実行が行われる．LMNtal 標準ライブラリ (2.4.5 節) は実行時に動的にリンクされる．これら一連の流れを図 2.5 に示す．

代表的なコマンドラインオプションとして以下が用意されており，角括弧内に示すコマンドで利用可能である．

`-e` *string*　：*string* をプログラムとして実行する [`lmntal`].

`-g`　(グラフィックモード)：グラフ構造を可視化する．ルールが 1 回適用されるたびに実行が中断されグラフ構造が表示される [`lmntal`, `lmnr`].

`-O`n ($0 \leq n \leq 3$)　：最適化レベルの指定 [`lmntal`, `lmnc`].

-s　(シャッフルモード)：膜内のルール選択や，ルールにマッチするアトムや膜の選択をランダムに行う (2.4.7 節) [lmntal, lmnc, lmnr].

-t　(トレースモード)：ルール適用のたびにプログラムの状態を表示する [lmntal, lmnr].

-vn ($0 \leq n \leq 9$)　：出力の詳細度レベルの設定 [lmntal, lmnr].

--debug　(コマンドラインデバッガ)：gdb のようにルールにブレークポイントを設定してステップ実行を行う [lmntal].

--library　：モジュールをコンパイルしてプログラムライブラリを作成する [lmnc].

-p　：実行時間やルール適用回数などのプロファイル情報を出力する [lmntal, lmnr].

--remain　：対話モードにおいて過去に入力した行を蓄積する．対話中に:remain, :noremain コマンドによって両モードを切り替えることも可能である [lmntal].

2.4 実用言語機能

第 2 節で紹介した LMNtal は単純かつ強力な基本計算モデルを提供しているが，実用プログラミング言語とするためにはさまざまな考慮が必要となる．λ 計算と Lisp，あるいは Horn 節論理と Prolog の対比を考えれば，前者から後者を得るのに自明ではない設計努力が必要となることは明らかであろう．本節では，実用言語として必要となり拡張した諸機能について，その背景・設計を交えて解説する．

2.4.1 構文の拡張

プログラムの可読性向上のために，プロセスの区切り記号としてカンマの他にピリオドを導入した．結合の優先順位は高い方からカンマ，:-，ピリオドの順であり，たとえば a. b, c :- d. は a, ((b,c) :- d) と解釈される．上記のように，ピリオドは最後のプロセスの後ろに付加することもできる．

さらに LMNtal の構文を次のように拡張した．

1. ルールは以下の構文のように**ガード**をもつことができる．

   Head :- *Guard* | *Body*

   ガードには，ルールの適用検査における付帯条件を指定する (2.4.3 節).
2. ルール左辺のセルは "/" をつけて {/*T*/} という形をとることができる．これは，当該セルが (自分自身がもつルールセットだけでは) それ以上書換えを進めることができないという条件を指定するもので，子孫膜における計算終了の検出に利用する (2.4.7 節).
3. アトム名の空間は大域的であるが，モジュール名で修飾した *modulename*.*atom* の形のアトム名を利用できる．修飾はたかだか 1 段階までである．*modulename* の部

分はモジュールの参照と解釈される (2.4.5 節).

4. ルールの先頭に *rulename*@@ の形を付加してルールに名前をつけることができる. トレースやデバッグに役立つ.
5. ブロックコメントとして /* */, 一行コメントとして // と % が利用できる.

2.4.2 システムルールセットと算術演算

言語モデルとしての LMNtal は算術演算を規定していないが, 実際にプログラムを書く上で, 基本的な算術演算の機能は必要である. そこで, 数値 (整数と浮動小数点) に対する算術演算の機能を**システムルールセット**として導入した. システムルールセットはすべての膜に暗黙に備わったルールセットであり, 四則演算は処理系組込みのシステムルールセットとしてサポートしている.

算術演算は引数を 2 つ取って計算結果を返すものであるので, LMNtal では二項演算子は 3 価のアトム, 数値は 1 価のアトムとして表現する. つまり引数を持たない 8 や 3.14 は LMNtal の数値ではなく, 8(X) や 3.14(X) が LMNtal における数値である. これらをそれぞれ**整数アトム**, **浮動小数点数アトム**と呼ぶ. 算術演算は中置記法で書くことができる. たとえば

```
n(1), n(2), n(3), n(4), n(5).
n(A), n(B) :- n(A*B).
```

というプログラムを与えると, 四則演算のルールセットによって n(120) という最終結果が得られる.

ユーザ定義のルールをシステムルールセットに追加する機能も備わっている. **ユーザ定義システムルールセット**とは, すべての膜が暗黙のうちに備えるべきルールセットをアトム system_ruleset と一緒に膜で囲んだものであり, これを用いることで, 特定のアプリケーションにとっての大域的な基本演算を簡潔に定義することができる.

2.4.3 型付きプロセス文脈

背景と目的

プロセス文脈は不特定のプロセス (階層グラフ構造) にマッチし, 1 本のルールで複数通りの書換えを記述する仕組みである. 2.2.4 節で述べたように, プロセス文脈にマッチする範囲は膜によって決定されるので, LMNtal における二つの構造化手段のうちの階層構造を扱う機能であると言える.

一方, 実際的なプログラムにおいては, グラフ (特別な場合として単一のアトムを含む) の比較, 複製, 破棄などの操作も重要であり, それぞれの操作を行うにあたっては, 対象のグラフをそれを取り囲む膜によって指定するのではなく, それ自身の接続構造によって限

定する機能も必要である．

上記の動機から，我々は**型付きプロセス文脈**を導入した．型付きプロセス文脈は従来のプロセス文脈の拡張概念であるが，プロセスの内部構造を規定する**プロセス型**を用いてマッチするプロセスを指定する．接続構造を扱う点で，従来のプロセス文脈の相補概念と考えることもできる．各プロセス型の詳細は 2.4.3 節で述べる．

型付きプロセス文脈は所属する膜の指定されたプロセス型の構造をもつプロセスにマッチする．型付きでないプロセス文脈と違い，膜の外に出現することや同じ階層に複数個出現することも許される．マッチする構造を指定する機能は，並行処理におけるデータの待ち合わせにも利用できる．

型付きプロセス文脈は型付きでないプロセス文脈と同様の構文で記述する．明示的な自由リンクを一つだけ持つ型付きプロセス文脈にはアトムと同様な略記法が用意されており，`a(X)`，`$b[X]` は `a($b)` または `a=$b` と略記できる．

プロセス型

我々が導入したプロセス型は `unary`, `int`, `float`, `string`, `class`, `ground` からなる．以下ではまず `unary` 型とその部分型について述べ，最後に `ground` 型について述べる．

`unary` **型のプロセス**は，単一の 1 価アトムからなるプロセスと定義される．たとえば `longlongname(L)`，`15(N)` は `unary` 型プロセスの例である．

`unary` 型のプロセスにのみマッチする型付きプロセス文脈を使ったルールは，たとえば次のように記述する．

```
a(A), $n[A] :- unary($n) | b(B), $n[B].
```

このように，型付きプロセス文脈に対するプロセス型の指定はガード部に記述する．複数個ある場合はカンマで区切る．このルールは，`a` という名前の 1 価のアトムと，その第 1 引数につながった任意の 1 価アトムとからなるプロセスに適用可能である．

この `unary` 型の部分型として，`int` 型，`float` 型，`string` 型を導入した．`int` 型，`float` 型はそれぞれ 2.4.2 節で述べた整数アトム，浮動小数点数アトムの全体である．また，`string` 型は**文字列アトム**の全体である．文字列アトムとは二重引用符で囲んで記述した名前をもつ 1 価アトムである．

また，他言語インタフェースの導入とともに，`unary` 型の部分型として `class` 型を導入した．これについては 2.4.6 節で詳しく述べる．

さらに，複数のアトムからなるグラフ構造を扱うために `ground` **型**というプロセス型を設計した．`ground` 型のプロセスとは，

- 膜を持たず
- リンクで全体が連結しており

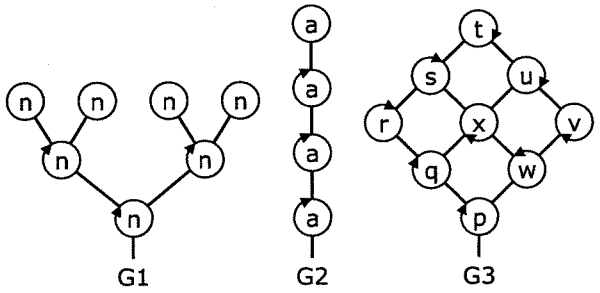


図 2.6　ground 型のプロセスの例

- ちょうど一本の自由リンクを持つ

プロセスのことである．具体例を図 2.6 に示す．図中の矢印は各アトムの第 1 引数の位置を示し，矢印の向きに第 2 引数，第 3 引数，... と並ぶことを意味している．これらのテキスト表現は次のようになる．

```
G1=n(n(n,n),n(n,n)),
G2=a(a(a(a))),
G3=p(q(r(s(t(u(v(w(x(Q,S,U),W)),U)),S)),Q),W)
```

設計上の重要な選択は，ground 型プロセスは膜を貫くリンクを持たないことにした点である．つまり ground 型プロセスは，膜をその構造の内部に持つことも膜をまたがって存在することもない．

膜は多重集合を扱うための概念である．つまり膜を含む構造の比較は，一般に階層的多重集合の比較を伴うため，基本操作として導入するには大きすぎる．そこで比較を目的の一つとする ground 型は膜を含まない仕様とした．

なお，unary 型のプロセスはすべて ground 型プロセスである．つまり現在実装されているすべてのプロセス型は ground 型の部分型であり，型つきプロセス文脈に対する型制約としては ground 型が最も弱い制約となる．

ガード制約

型付きプロセス文脈に関しては，その構造を限定するプロセス型指定以外にも，いくつかの制約をガード部に指定できる．これらのガード部に指定する制約条件を**ガード制約**という．各プロセス型に定義されているガード制約は表 2.1 の通りである．

プロセス型指定以外のガード制約の代表例は '=' である．'=' は ground 型プロセスに対して定義された制約であり，その部分型である unary 型，int 型，float 型，string 型にも使用できる．

'=' を用いることで，二つの型付きプロセス文脈が等しい構造を持つ場合にのみ適用され

表 2.1　プロセス型の制約

制約の種類	ガードに記述可能な制約	本制約が含意する型制約
型制約	ground($c)	
	unary($c)	ground($c)
	int($c)	unary($c)
	float($c)	unary($c)
	string($c)	unary($c)
	class($c, クラス名)	unary($c)
比較制約	$c1=$c2	ground($c1), ground($c2)
	$c1\=$c2	ground($c1), ground($c2)
	$c1=:=$c2, $c1=\= $c2	
	$c1 < $c2, $c1=< $c2	int($c1), int($c2)
	$c1 > $c2, $c1>= $c2	
	$c1=:=.$c2, $c1=\=.$c2	
	$c1 <. $c2, $c1=<. $c2	float($c1), float($c2)
	$c1 >. $c2, $c1>=. $c2	
その他	uniq($c1, ... ,$cn)	ground($c1), ..., ground($cn)

るルールを記述することができる．たとえば

```
person(tommy,male),
person(jenny,female),
person(danny,male),
getout(male).

getout($g), person($n,$s) :-
   unary($g), unary($n), unary($s), $g=$s |
   getout($g).
```

というプログラムを実行すると

```
getout(male), person(jenny,female).
```

という結果が得られる．

int 型と float 型には大小比較演算も用意し，両辺には四則演算も記述できるようにした．たとえば，最大の整数アトムを引数に持つプロセスのみを残しそれ以外を消すプログラムは

```
num($n), num($m) :- $n>$m | num($n).
```

と書くことができる．このように '>' 制約を記述した場合，int 型の型指定を省略できる．これは，'>' 制約が int 型だけに定義された比較演算であるためである．

なお，ガードを用いたルールは，ガードを持たないルールの (無限) 集合を表現した**ルールスキーム**と考えることができる．たとえば

```
a($m,$n) :- $m>0, $m<$n | b($m,$n)
```

というルールは，次の無限個のルールの集合と等価である．

```
a(1,2) :- b(1,2).  a(1,3) :- b(1,3). ...
a(2,3) :- b(2,3).  a(2,4) :- b(2,4). ...
a(3,4) :- b(3,4).  a(3,5) :- b(3,5). ...
...
```

このように，ガードを用いたルールは，各プロセス文脈にガード条件を満たす具体的なプロセスを代入した (ガードをもたない) ルールの無限集合と等価である．図 2.4 の構文ではプロセス内のルールの個数を有限個に限っているが，ルールセットの概念をルールの無限集合に拡張しても基本言語モデルの操作的意味論は変更不要であり，その意味で，ガードを用いたルールは基本言語モデルからの自然な拡張であると言える．

この考え方に立つと，ガードで新たな型付きプロセス文脈を定義することもできる．たとえば

```
plus($x,$y,Ret) :- $z=$x+$y | $z[Ret].
```

は，$5 = 2 + 3$ であるから

```
plus(2,3,Ret) :- 5(Ret).
```

を含む．この例は $z の値がガード部で確定するが，具体値が確定しない型付きプロセス文脈がある場合はコンパイル時エラーとしている．

なお，上記の比較演算はそれぞれ，二つのプロセス型の直積の部分型と考えることもでき，その意味で広義の型指定と見なすことも可能である．

プロセス文脈の複製・破棄

型付きであるなしに関わらず，プロセス文脈はルール右辺での出現が複数回ならば複製を意味し，出現が 1 回ならば移動または存続，出現がなければ破棄を意味する．

```
cp(X),{$p[X]} :- done(Y,Z),{$p[Y]},{$p[Z]}.          // 複製
mv(X,Y),{$p[X]},{$q[Y]} :- done(Z,W),{$p[Z],$q[W]}.  // 移動
cn(X),{$p[X]} :- done(Y),{$p[Y]}.                    // 存続
rm(X),{$p[X]} :- ().                                 // 破棄
```

複製・破棄操作の計算量は，型なしのプロセス文脈も型付きプロセス文脈もマッチするプロセスの大きさに比例する．また，型付きプロセス文脈の比較検査についてもマッチするプロセスの大きさに比例した計算量で実現できている．これは，比較対象のグラフ構造が連結であり，しかも比較開始点 (根) が決まっているためである．

論理プログラムとの関係

型付きプロセス文脈は，階層グラフ書換え言語である LMNtal と論理プログラミング言語との関係づけにも役割を果たしている．

LMNtal ではリンクの出現回数に強い構文的制限があり，数値やリストのようにデータの役割をもつアトムは，プロセス (または手続き) の役割をもつアトムと一対一結合しなければならない．これは，データはプロセスが資源として所有するものであるという見方を反映したものである．

その設計の背景には，過去に記述された論理プログラムの分析がある．論理プログラムの各ルール中の変数の出現回数には特に制限がないが，Prolog のリスト処理プログラムを観察すると，多くの変数がルール中にちょうど 2 回出現することがわかる．また 190 ルールからなる並行論理 (KL1) プログラムの分析結果 [9] によれば，90% の変数がちょうど 2 回出現し，5% の変数が 1 回，残り 5% の変数が 3 回以上出現していた．3 回以上出現する変数も，基底項を共有するための変数であることが静的モード解析によってわかるものであった．つまり，実際の論理プログラムの変数の多くは一対一の情報授受に使われており，それと一対一以外の情報授受とを別の方法で扱う言語を考えるのは意味のあることである．Janus[7] はそのような思想で設計された並行論理型言語であり，変数の出現回数を LMNtal と同じく構文的に 2 回に限定し，一対一以外の情報授受には特別な標記を付した変数を用いることとした．

LMNtal では，一対一以外の情報授受に型付きプロセス文脈を用いることができる．たとえば図 2.7 は LMNtal によるパイプライン作業に基づく素数生成プログラムであるが，論理型言語の素数生成プログラムの中の非線形変数 (ガード以外に 1 回以下または 3 回以上現れる変数) を型付きプロセス文脈に機械的に置き換えて，データの複製や廃棄を明示したものである．実際，型付きプロセス文脈を論理変数に戻し，演算子 '|' をカット '!' に変更し，2 行目の加算のための代入演算子を `is` に変更すれば Prolog 処理系で動作する．

複数の解をもつ論理プログラムの LMNtal での取扱いの例は 2.4.7 節で簡単に紹介する．

```
gen($n,$max,Ns) :- $n > $max | Ns=[].
gen($n,$max,Ns) :- $n =<$max | Ns=[$n|Ns1], N1=$n+1, gen(N1,$max,Ns1).

sift($max,[],Zs) :- int($max) | Zs=[].
sift($max,[$p|Xs],Zs) :- $p*$p<$max  |
    Zs=[$p|Zs1], filter($p,Xs,Ys), sift($max,Ys,Zs1).
sift($max,[$p|Xs],Zs) :- $p*$p>=$max | Zs=[$p|Zs1], sift($max,Xs,Zs1).

filter($p,[],Ys) :- int($p) | Ys=[].
filter($p,[$x|Xs],Ys) :- $x mod $p=:=0 | filter($p,Xs,Ys).
filter($p,[$x|Xs],Ys) :- $x mod $p=\=0 | Ys=[$x|Ys1], filter($p,Xs,Ys1).

primes($max,Ps) :- int($max) | gen(2,$max,Ns), sift($max,Ns,Ps).
```

図 2.7　素数生成プログラム

2.4.4　uniq

LMNtal のルールは適用可能な限り何回でも適用されるが，実際のプログラムでは，あるルールを同等なグラフ構造に対して一度だけ適用させたい場合もある．そこでマッチした構造の履歴を管理して，同等な構造に対しては一度しか適用しないルールを記述するための uniq 機能を設計した．uniq は型に関する制約ではないが，ガード制約の一つと位置づけることとした．

履歴として管理する対象は，管理が比較的容易な ground 型に限定することとした．

uniq 制約は uniq($c_1, \ldots, \$c_n$) $(n \geq 0)$ という形をもち，

1. 個々の $\$c_i$ が ground 型プロセスで，かつ
2. この制約をもつルールが，同じプロセスの組に対して過去に適用されたことがない

場合に成功する．この特別な場合として，引数をもたない uniq は当該ルールがかつて使用されたことがない場合のみ成功する．

たとえば，プログラム

```
n(1), n(2), n(3), n(4).
n($a), n($b)  :-
   $a<$b, uniq($a,$b), $c=$a*$b  |
   n($a), n($b), m($c).
```

を実行すると，次のような結果が得られる．

```
n(1), n(2), n(3), n(4),
m(3), m(4), m(2), m(6), m(8), m(12).
```

`uniq($a,$b)` により，同じ整数アトムの組み合わせに対しては一度だけ反応していることがわかる．

2.5.4 節では，`uniq` 機能の重要な応用例として CHR の LMNtal へのエンコーディングを論じる．

2.4.5 モジュールシステム

信頼性を低下させずに大きなプログラムを作るためには一度に取り扱うコード量は適度に小さくなければならない．そこで，LMNtal プログラムを構成するルール集合をより小さな集合に分割し，関連性の高いルールをモジュールとしてまとめ，機能ごとのプログラム再利用ができるようにするために，モジュールシステムを導入した．

設計

モジュールとは，ルールの集合 (ルールセット) を膜で囲ってグループ化したものである．またモジュールを使うとは，利用側の膜内のプロセスにモジュールのルールを適用することである．しかし，LMNtal の基本計算モデルにおいては，膜で囲んだルールセットが膜の外のプロセスと直接反応することはできず，反応できるようにするにはモジュール内のルールセットを利用側の膜にコピーする必要がある．

このようなルールセットのコピー作業は，LMNtal のルール文脈機能を使って記述することもできるが，モジュール機能の重要性を考えると，プログラミング言語としてはもっと積極的な形でサポートする意義がある．

モジュールシステムの導入にあたって行った言語拡張は以下の通りである．

- モジュール名宣言を導入した．**モジュール名宣言**は膜内で `module(`*modulename*`)` のように記述する．モジュール名宣言とルールセットを含んだ膜を**モジュール膜**と呼ぶ．
- モジュール名つきアトム名 (2.4.1 節) を導入し，アトム名をモジュール名で修飾できるようにした．
- モジュール名つきアトムが生成されるタイミングで，そのモジュール名に対応するモジュール膜の中のルールを呼出し側の膜にコピーするようにした．

なお，LMNtal の名前空間は単一かつ大域的である．モジュール名つきアトム名はモジュール名で修飾した形式で大域的名前空間に属し，必ずその形で使用する．モジュール膜の中のルールの左辺には，そのモジュール名がついたアトム名が必ず含まれていなければならないわけではなく，通常のアトム名だけからなっていてもよい．

```
{ module(list).

  H=list.new :- H=[].
  ...
  H=list.append([], B) :- H=B.
  H=list.append([Value|Next], B) :-
     H=[Value|list.append(Next, B)].
  ...
}.
```

図 2.8　モジュール list (抜粋)

表 2.2　標準ライブラリ

種類	モジュール名
入出力，ソケット通信	io, socket
データ，データ構造	string, integer, float, boolean, list, array, map, queue
制御，システム関係	thread, nd, timer, sys, java, if, nlmem
GUI	graphic, wt

標準ライブラリ

さまざまなアプリケーションを記述するために必要となるモジュールが標準ライブラリとして提供されている．たとえばモジュール list は図 2.8 のように定義されている．次はその使用例である．

```
a = list.append([1,2], [3]).
```

実行すると，2 つのリストが連結されて a=[1,2,3] という結果が得られる．

現在は list の他に表 2.2 に示すモジュールを標準ライブラリとして提供している．モジュール nd (nondetermistic) は，すべての書換え方法を全解探索し，結果を簡約グラフ (reduction graph) の形で生成する．モジュール sys (system) はコマンドライン引数の解析や外部コマンドの実行などを行う．モジュール java は他言語インタフェース (2.4.6 節) における Java 言語のインスタンス作成やメソッド呼び出しを行う．wt は Window Toolkit である．

モジュール integer は多重集合生成に便利な integer.set 機能を備えている．たとえば集合 $\{\mathtt{n}(i) \mid 1 \leq i \leq 100\}$ は，再帰を用いなくても n(integer.set(1,100))

```
{ module(if).
  ...
  H=if(true,  T, F), {$t[T]}, {$f[F]} :- $t[H].
  H=if(false, T, F), {$t[T]}, {$f[F]} :- $f[H].
  ...
}
```

図 2.9　モジュール if (抜粋)

と書くことで生成できる．integer.set は，第 1 引数と第 2 引数で指定した範囲の各整数アトムに，最終引数につながる ground プロセスのコピーを接続する．この仕様の帰結として，直積集合 $\{n(i,j) \mid 1 \leq i \leq 100, 1 \leq j \leq 50\}$ は n(integer.set(1,100), integer.set(1,50)) で生成できる．二つの integer.set の展開順序は非決定的であるが，かりに後者の展開が先行して，たとえば n(integer.set(1,100), 3) すなわち integer.set(1,100,X), n(X,3) が生成されると，ground プロセス n(X,3) のコピーが 100 個作られて，各コピーの自由リンクが 1 価アトム i $(1 \leq i \leq 100)$ に接続される．

モジュール integer とモジュール float は結果を true または false アトムで返す比較演算子を備えている．これを利用して条件分岐を行う機構をモジュール if が提供しており (図 2.9)，たとえば最大公約数を計算する以下のプログラム

```
H=gcd($m,$n) :- $m>$n | H=gcd($m-$n,$n).
H=gcd($m,$n) :- $m<$n | H=gcd($m,$n-$m).
H=gcd($m,$n) :- $m=$n | H=$m.
```

は，次のように書くことができる．

```
if.use. integer.use.
H=gcd($m, $n) :- int($m), int($n) |
   H=if($m>$n, T, F),
      {T=gcd($m-$n, $n)},
      {F=if($m<$n, T2, F2),
         {T2=gcd($m,$n-$m)},
         {F2=$m}
      }.
```

if アトムの第 1 引数が条件節，第 2 引数が then 節，第 3 引数が else 節である．then 節，else 節は膜によって保護されており，条件節の真偽が求まるまで実行されない．モジュール if を使うことで，同じ形のルール左辺を複数回書く必要がなくなる．

モジュール nlmem (nonlinear membrane) には，次のルールに相当するルールが他言語インタフェースを用いて記述されている．

```
R=nlmem.copy(X), {$p[X|*Z]} :-
   R=copied(X1,X2),
   {$p[X1|*Z1]}, {$p[X2|*Z2]},
   copied(*Z1,*Z2,*Z).
nlmem.kill(X), {$p[X|*Z]} :- killed(*Z).
```

これは明示的でない自由リンクをもつプロセス文脈の複製・破棄を行うルールである．明示的でない自由リンクをもつプロセス文脈の複製・破棄を行うには，それらの自由リンクの接続方法を指定するためにアトム集団 (上の例では copied(*Z1,*Z2,*Z) と killed(*Z)) を用いる必要がある．未実装の言語機能をカバーするのに他言語インタフェースを活用した例である．

2.4.6 他言語インタフェース

新たなプログラミング言語の有用性や利便性を高めるためには，既存のプログラミング言語との円滑なインタフェースを提供することが重要である．他言語インタフェースの主な目的は次の通りである．

- 外界との通信
  入出力，グラフィカルユーザインタフェース，ネットワーキングなど，ベース言語やその API が提供する機能に LMNtal プログラムから自由にアクセスできるようにする．
- 処理の高速化
  オペレーティングシステム機能へのアクセスを伴わない計算は LMNtal 言語で表現できるが，ベース言語で実装すると実行効率が大幅に改善するアルゴリズムやデータ構造も存在する．そのような場合，他言語で書いた効率の良いコードを利用することが可能になる．

設計

前述の動機や目的を達成するための設計を論ずる．決めるべき事柄は次の 3 つである．

1. どの言語を書けるようにするか
2. 他言語コードの記述場所
3. 他言語コードの実行タイミング

1. については，処理系の実装言語である Java を選択した．Java の豊富なライブラリ機能が LMNtal から使えるようになることも大きな理由である．

2. については，以下に説明するインラインアトム方式を採用した．インラインコードを LMNtal のプロセスとして容易に扱うことができ，また LMNtal と Java 間の情報の受け渡しがスムーズにできるからである．

以下ではインラインアトムと 3. に関する詳しい説明を与える．

インラインアトム

他言語とのインタフェースは言語モデルには存在しない．インライン機能を実現するために構文に特殊な変更を加えるのは最低限にしたいと考えた．

そこで，既存の言語要素であるアトムを利用する方法を採用した．具体的には，「アトム名がある文字列で始まるとき，そのアトム名そのものをインラインコードとして扱う」という方法である．そのようなアトムを**インラインアトム**と呼ぶことにする．

インラインアトムは，構文的にはアトムの一種にすぎないが，処理系から見るとインラインコードが含まれるアトムとして区別される．こうすることで，言語モデルに対する変更は一切なしに他言語インタフェースを実現できる．

インラインアトムはさらにインライン実行アトムとインライン宣言アトムの 2 種類に分類される．

- インライン実行アトム
  `/*inline*/` で始まる名前を持つアトムを，**インライン実行アトム**と呼ぶ．
  インライン実行アトムに書かれた Java コードは，LMNtal ソースのコンパイル時に Java 処理系を通してコンパイルされ，ルールセットごとにインライン実行アトム用クラスが生成される．実行時には，インライン実行アトムを生成するルールに対応する中間命令 (トランスレータの場合は対応する Java コード) の実行が終わってから対応するインライン実行アトム用クラスがロードされ，対応するインラインコードが実行される．ルールによるデータ構造操作がすべて終わってからインラインコードを実行しないと，インラインコードがアクセスしたいデータ構造がまだ生成されていない可能性があるからである．
  通常のアトムの意味が，当該アトムを左辺に含むルールによって定まるのに対し，インライン実行アトムの意味はそのインラインコードによって定まると考えることができる．
- インライン宣言アトム
  `/*inline_define*/` で始まる名前を持つアトムを，**インライン宣言アトム**と呼ぶ．
  インライン宣言アトムには Java のクラス定義を書くことができる．インライン宣言アトムがソースコード中のどこかに存在すると，対応するインラインコードは Java コードファイルのグローバルな領域にそのまま展開される．したがって，独自のク

```
io.use :-
  io.stdin=[:/*inline*/
    Atom stdin = mem.newAtom(new ObjectFunctor(
      new java.io.BufferedReader(new java.io.InputStreamReader(System.in))));
    mem.relink(stdin, 0, me, 0);
    me.remove();
    :],
  io.stdout=[:/*inline*/
    Atom stdout = mem.newAtom(new ObjectFunctor(
      new java.io.PrintWriter(System.out, true)));
    mem.relink(stdout, 0, me, 0);
    me.remove();
    :].
```

図 2.10　io.use の実装

ラス定義をインライン宣言アトム名として書き，そのクラスをインライン実行アトムの生成時に参照することができる．
インライン宣言アトムを見つけて Java ソースに展開する処理は，コンパイル時に静的に行われる．

このインライン方式を採用するにあたり，他言語コードがそのまま書ける構文的クォータ [: :] を導入した．[: :] 内にはスペースやタブや改行を記述でき，そのままアトム名となる．

ここで，図 2.10 および図 2.11 を例にして LMNtal における Java コードの使用例を説明する．

インライン実行アトムのコード中では，当該インライン実行アトムに対応する Java オブジェクト me およびそれが属する膜を表す Java オブジェクト mem が使用できる．これによって，インライン実行アトムとリンクでつながっているアトムの名前を強制的に書き換えたり，膜を操作したりすることが可能である．たとえば i 番目の引数はインラインコードで me.nth(i) のようにアクセスすることができる．

詳細は省くが，インラインコードの中でスレッドを使用することもできる．別スレッドの中から LMNtal の実行時データ構造に安全にアクセスするためのロック機構も提供している．

他言語インタフェースの利用に際し，Java インスタンスを生成してそれを保持しておきたい場合，インライン実行アトム内で Java インスタンスを生成し，**Java インスタンスアトム**として保持することができる．そして，プロセス型として新たに導入した class 型で制約した型付きプロセス文脈 (2.4.3 節) を用いることで，こうして生成された Java インス

```
H=io.print(String), io.stdout(STDOUT) :-
  class(STDOUT, "java.io.PrintWriter") |
  H=io.print(STDOUT, String), io.stdout(STDOUT).

H=io.print(Object, String) :-
  class(Object, "java.io.PrintWriter"), string(String) |
  H=[:/*inline*/
    try {
      java.io.PrintWriter pw =
        (java.io.PrintWriter)
          ((ObjectFunctor)me.nthAtom(0).getFunctor()).getObject();
      Atom done = mem.newAtom(new Functor("done", 1));
      if(pw!=null) {
        pw.print(me.nth(1));
        pw.flush();
      }
      mem.relink(done, 0, me, 2);
      me.nthAtom(1).remove();
      me.remove();
    } catch(Exception e) {e.printStackTrace();}
  :](Object, String).
```

図 2.11　io.print の実装

タンスアトムにだけ反応するルールを記述できる．図 2.11 の例が示すように，ガード内の class は，第 1 引数で指定された型付きプロセス文脈が，第 2 引数で指定されたクラスに属することを検査するものである．

モジュール io を利用する側はまずアトム io.use を用いて初期化を行う必要がある．io.use の中身は図 2.10 のようになっており，このルールによって標準入力，標準出力を表すアトム io.stdin，io.stdout が生成される．これらは io.print，io.readline などが呼び出されるときにモジュール内部で使われる．

2.4.7　ルール適用の制御機能

LMNtal は，書換え可能なプロセスや適用可能なルールが複数存在する場合の適用順序を規定しておらず，処理系が自由な順序で適用することができる．

シャッフルモード

シャッフルモード (2.3.1 節) を利用すると，膜内のルールの選択や，ルールにマッチするアトムや膜の選択を乱数を用いて行うことができる．

例として以下のような自然数の分割プログラムを挙げる.

```
add(X, Y,Z)     :- X=0, Y=Z.
add(X0,Y,s(Z)) :- X0=s(X), add(X,Y,Z).

add(x,y,s(s(s(0)))).
```

標準の実行モードにおいては常に同じ実行結果が得られるが，シャッフルモードで実行した場合

- (1 回目) `y(s(s(s(0)))), x(0)`
- (2 回目) `y(s(0)), x(s(s(0)))`
- (3 回目) `y(s(s(0))), x(s(0))`
- ...

のように実行結果はランダムに決まる．なお，nd モジュール (2.4.5 節) を使ってすべての書換え経路を探索することもできる.

ルール適用の逐次化

多くの実際的な計算はいくつかのフェーズからなり，個々のフェーズで異なるルールセットを使用することも多い．LMNtal の基本言語モデルでは，ルール文脈の機能を使うと膜の中のルールセットを動的に入れ替えることができるが，ルールを入れ替えるタイミング，すなわちあるフェーズの計算の終了を検出することは一般に困難である.

そこで，ある子膜の計算が終了していることを検出するための機能を追加した.

ルール左辺のセルに "/" をつけて {T}/ という形 (2.4.1 節) で指定すると，{T} の形であってかつ (自分自身がもつルールセットだけでは) それ以上書換えを進めることができないセルにのみマッチする.

図 2.12 はこの機能を利用した素因数分解プログラムである．`{n($prime),$p,@p}/` によって，素数を生成する膜の実行が終了するまで待合せを行っている．このプログラムを実行すると,

```
p(3,1), p(2,2)
```

という結果が得られる．これは $12 = 3^1 \times 2^2$ であることを表している.

2.5 各種計算モデルとの関連付け

LMNtal は多重集合や並行処理などの概念を持つ計算モデルの統合を一つの目標としている．多重集合はグラフの特殊な場合であるので，Gamma モデルのような多重集合書換えモデルが LMNtal に自然にエンコードできるのはほぼ明らかであるが，我々はそれに加

```
/* generate primes in a membrane */
{ n(integer.set(2,12)).
  n($m), n($n) :- $m mod $n =:= 0 | n($n).
}.

/* extract prime numbers out of the membrane */
{n($prime),$p,@p}/ :- int($prime) | {$p,@p}, t($prime,12,0).
{@p} :- .

/* factorize */
t($p,$a,$k) :- $a mod $p =:= 0, int($k) | t($p, $a/$p, $k+1).
t($p,$a,$k) :- $a mod $p =\= 0, int($k) | p($p, $k).
p($p,0) :- int($p) | .
```

図 2.12　素因数分解

えて，(i) λ 計算，(ii) π 計算，(iii) ambient 計算，(iv) CHR などの重要な計算モデル (またはその中心的機能) を LMNtal にエンコードし，処理系上で動作確認を行った．

2.5.1　λ 計算

λ 計算がどのようにエンコードできるかを確認することは，計算モデルの表現力を考察する上で大変有用でかつ興味深い．LMNtal はグラフ書換え言語であり，一方 λ 式における変数の束縛関係はグラフとして自然に表現できるので，LMNtal でグラフ簡約に基づく λ 計算が実現できることを確認することは重要である．

LMNtal に関連の深い言語として Interaction Nets[4] がある．Interaction Nets は強い構文的制限をもつ階層のないグラフ書換え言語であるが，LMNtal は Interaction Nets の構文的制限を緩和して拡張した言語と見なせる．λ 計算の Interaction Nets へのエンコードはさまざまな提案があるが，中でも Sinot の方法 [8] が簡明である．そこで我々はこれを出発点に独自の工夫を行って LMNtal へのエンコードを行った (図 2.13)．

文献 [8] の方法は，(i) 値呼びと名前呼びを，λ 式を表すグラフ構造の中を動き回る 1 個の制御トークンを用いて実現した点と，(ii) λ 式のコピーを 2 種類のコピートークンと他のグラフノードとの反応によって実現した点を特徴とする．しかしこの方法によって得られるものは弱頭部標準形 (weak head normal form, $xM_0 \ldots M_n$ $(n \geq 0)$ もしくは $\lambda x.M$ の形の式，M および M_i は標準形でなくてもよい) であり，かつその計算は制御トークンによって逐次化している．これに対して図 2.13 のプログラムは，λ 計算の基本的な簡約意味論の LMNtal へのエンコードであり，あらゆる β 基を非決定的に簡約する．

λ 式のエンコードにはアトム `lambda`，`apply`，`rm`，`cp`，`fv` を用いている．λ 式の束縛変

```
beta@@ H=apply(lambda(A, B), C) :- H=B, A=C.

l_c@@  lambda(A,B)=cp(C,D,L), {+L,$q} :-
         C=lambda(E,F), D=lambda(G,H), A=cp(E,G,L1), B=cp(F,H,L2),
         {{+L1},+L2,sub(S)}, {super(S),$q}.
a_c@@  apply(A,B)=cp(C,D,L), {+L,$q} :-
         C= apply(E,F), D= apply(G,H), A=cp(E,G,L1), B=cp(F,H,L2),
         {+L1,+L2,$q}.

c_c1@@ cp(A,B,L1)=cp(C,D,L2), {{+L1,$p},+L2,$q} :- A=C, B=D, {{$p},$q}.
c_c2@@ cp(A,B,L1)=cp(C,D,L2), {{+L1,$p},$q}, {+L2,top,$r}
          :- C=cp(E,F,L3), D=cp(G,H,L4), {{+L3,+L4,$p},$q},
             A=cp(E,G,L5), B=cp(F,H,L6), {+L5,+L6,top,$r}.
u_c@@  fv($u)=cp(A,B,L), {+L,$q} :- unary($u) | A=fv($u), B=fv($u), {$q}.
l_r@@  lambda(A,B)=rm :- A=rm, B=rm.
a_r@@  apply(A,B)=rm :- A=rm, B=rm.
c_r1@@ cp(A,B,L)=rm, {+L,$q} :- A=rm, B=rm, {$q}.
c_r2@@ cp(A,B,L)=rm, {{+L,$p},$q} :- A=rm, B=rm, {{$p},$q}.
c_r3@@ A=cp(B,rm,L), {+L,$p} :- A=B, {$p}.
c_r4@@ A=cp(rm,B,L), {+L,$p} :- A=B, {$p}.
r_r@@  rm=rm :- .
u_r@@  fv($u)=rm :- unary($u) | .

promote@@ {{},$p,sub(S)}, {$q,super(S)} :- {$p,$q}.
c2c@@  A=cp(B,C) :- A=cp(B,C,L), {+L,top}.
gc@@   {top} :- .
```

図 2.13　非決定的 λ 計算

数は LMNtal のリンクにエンコードされるが，前者は出現回数に制限がないので，ちょうど 2 回出現する変数以外のエンコードにはリンクの終端や分岐が必要となる．ここでは 1 価アトム rm (remove) を導入後一回も使わない変数を終端するために，3 価アトム cp (copy) を複数回使う変数のためにリンクを分岐させるのに用いている．簡約の過程において 3 価の cp を 4 価の cp に変換するがこれについては後述する．

たとえば自然数 n の Church によるエンコーディング (Church 数) は $\lambda fx.f^n x$ であるが，LMNtal での表現は

```
0:  lambda(rm,lambda(X,X),Result)
1:  lambda(F,lambda(X,apply(F,X)),Result)
2:  lambda(cp(F0,F1),lambda(X,apply(F0,apply(F1,X))),Result)
    ...
```

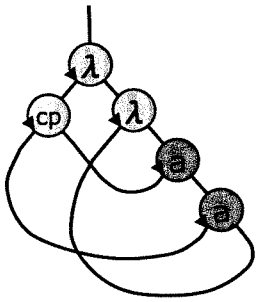


図 2.14　Church 数 2 のグラフ表現

となる (図 2.14. a は apply を表す).

λ 式中の自由変数は，自由変数の存在を示す 2 価アトム fv を介して，その自由変数の名前をもつ 1 価アトムを接続することで表現する．たとえば λ 式 xy の表現は apply(fv(x),fv(y),Result) となる.

図 2.13 の最初のルール beta は，λ 計算の中核をなす β 簡約を表す．右辺の A=C は引数の授受を，H=B は λ 式本体の計算結果を全体の計算結果として返すことを表す．それに続く 13 本のルールは，アトム rm と cp が他のアトムと出合った場合の処理ルールであり，グラフ構造をインクリメンタルに廃棄または複写している．最後の 3 本は，以下に述べるアトム cp の色管理のためのものである.

λ 式の中の 3 価アトム cp はまずルール c2c によって 4 価アトム cp に変換される．追加の引数は，入れ子になった λ 式の複写作業を並行して行う際に，出自の異なる cp どうしを識別するためのものである．この識別情報を cp アトムの**色**と呼ぶことにする．色は実際にはセルを用いて表現していて，同一セルに入射する色リンクは同一の色を表現するものと解釈する．さらに色は階層木構造を形成し，親子関係にある色セルは，super アトムと sub アトムを両端にもつリンクで相互接続している．最上位の色セルは，sub アトムのかわりに 0 価の top アトムを保有する．個々の 4 価の cp は，初期状態では最上位の色をもつ．実装を簡単にするために，ルール c2c は cp アトムごとに最上位色のセルを個別に生成しているが，最上位色のセルどうしを統合するか否かは，本エンコーディングの動作には影響しない.

グラフの複写作業が始まるのは，β 簡約によって仮引数と実引数とが相互接続されて，仮引数側の cp アトムが実引数側の lambda や apply や自由変数と正対したときである．cp が apply に出合うと相手を複写して自分も分裂し，apply の二つの引数の複写に行くが，その際には色は変化しない (ルール a_c)．これに対して，cp が lambda に出合うと，相手を複写して自分も分裂する点は同じであるが，その時点で新たな下位の色を作成して，分裂した cp はその色に変化する (ルール l_c)．また，λ 式の仮引数側から反時計回り (λ 式を図 2.14 のように書いた場合) に動く cp と本体側から時計回りに動く cp は，色は同一だが周回方向が異なるので，前者は色セルの内部に別の膜を設けてその内部で色リンクを終端

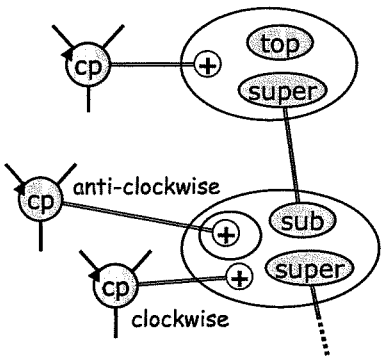


図 2.15　cp アトムの色表現

することで後者と区別する．反時計回りの cp は通常の cp と逆の極性 (負極性) をもつと考えるとわかりやすい．図 2.15 に，最上位色の cp アトムおよびその下位色の cp アトム 2 個 (正極性および負極性) のグラフ表現を示す．

Church 数の例からもわかるように，λ 抽象の仮引数のリンクは，rm で終端されるか，または 0 個以上の cp で枝分かれして，本体を表すグラフのどこかとつながる．したがって反時計回りの cp はいずれ同じ色で時計回りの cp と正対し，ルール c_c1 によって複写が完了する．反時計回りの cp が最上位色の cp と正対したときは，ルール c_c2 を用いて相手を複写した上で，自分も 2 個に分裂して前進する．

一方，λ 抽象の内部を時計回りに動く cp は，同じ色で反時計回りの cp といずれ出合うとは限らない．その λ 抽象が非局所変数をもつ場合，その変数を表すリンクを伝って上位レベルに脱出するからである．脱出した cp は元の上位の色に戻す必要がある．この作業を行うのがルール promote で，同じ色で負極性の cp の個数が 0 になったとき，残存する正極性の cp の色を上位色と統合している．

図 2.16 に，cp アトムに関する反応規則を図示する．図示の都合上，異なる色はリンクでなくて添字で区別している．添字 0 は最上位色を表す．

残りのルールのうち，u_c は大域自由変数の複写ルールである．l_r, a_r, c_r1, c_r2, r_r, u_r は，rm アトムと出合った相手を消去するためのルールである．c_r3 と c_r4 は，cp による分岐先の片方が rm で終端された場合の簡約ルールである．gc は，どこからも参照されなくなった最上位色のセルを消去するルールである．

Sinot の方法では，グラフ構造の複写を行うのに 2 種類の複写トークンを使い分けていた．2 色で済んだのは λ 抽象の内部を評価しないためである．我々の方法では λ 抽象の内部を非同期的に評価するために，セルを用いて多色を表現することとした．

λ 計算のエンコードの動作確認のための重要な例題は Church 数のべき乗である．Church 数のべき乗 m^n は $\lambda mn.nm$ とエンコードされるが，この式は単純であるにもかかわらず解の大きさに相当する (すなわち指数的な) 量のグラフ複写作業をともなう．実際に図 2.17 のプログラムを実行すると，res=fv(9) へと評価される．この例のように，

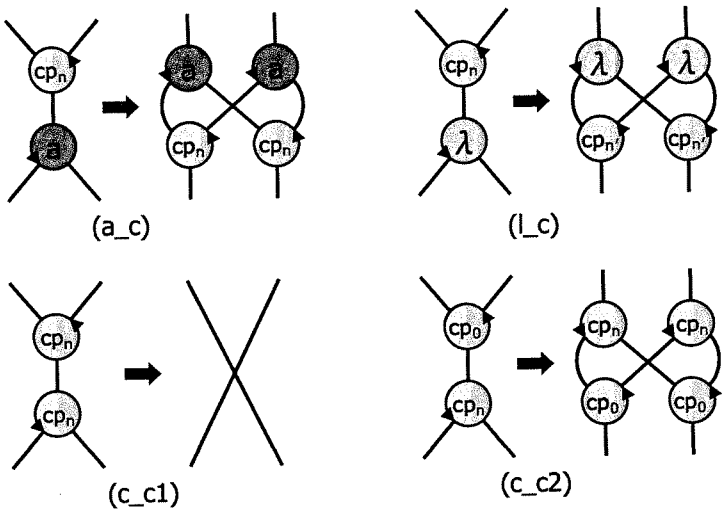


図 2.16　cp アトムの反応規則

```
N=n(2) :- N=lambda(cp(F0,F1), lambda(X, apply(F0,apply(F1,X)))).
N=n(3) :-
   N=lambda(cp(F0,cp(F1,F2)), lambda(X, apply(F0,apply(F1,apply(F2,X))))).
res=apply(apply(apply(n(2), n(3)), fv(s)), fv(0)).
H=apply(fv(s), fv($i)) :- int($i) | H=fv($i+1).
```

図 2.17　Church 数とその冪乗計算

LMNtal へのエンコーディングでは，純粋な λ 計算の枠組みに s や 0 のような自由名に対する書換えルール (図 2.17 の最後のルール)，すなわち δ 簡約ルールを追加することができ，これによって λ 計算から LMNtal へのインタフェースをスムーズにとることができる．なお，図 2.17 の実行結果は一意的であるものの，実行順序には多くの非決定性があり，結果を得るのに必要な簡約回数も一意には定まらないことが，-s (シャッフル) および -t (トレース) オプションを用いて観察できる．

我々は上記の非同期的エンコードに加えて，文献 [8] の方法に忠実な名前呼び λ 計算もエンコードし，Church や Turing の不動点演算子を用いて表現した再帰関数の動作確認を行った．

2.5.2 π 計算

並行計算の代表的なモデルである π 計算と LMNtal との関係づけも重要である．π 計算の定式化には多数のバリエーションがあるが，ここでは下記のものを考えることとする．

$$
\begin{aligned}
p &::= x(y) \mid \overline{x}\langle y\rangle \\
R &::= \Sigma_{j=1}^{n} p_j.P_j \quad (n \geq 0) \\
P &::= (\mathsf{new}\ a)P \mid P|P \mid R
\end{aligned}
$$

通信は下記の簡約規則にしたがって行われる．

$$
(x(y).P + M \mid \overline{x}\langle z\rangle.Q + N) \rightarrow P[z/y] \mid Q \tag{2.1}
$$

上記の定式化では文献 [5] に従い，プレフィクス (p) つきのプロセスに対してだけ和 (sum) をとることを認めることで選択 (choice) を通信とを一体化し，自律的な選択 (internal choice) を排除しているが，これはさまざまな問題のエンコードの観点からも現実的な制限と言える．

π 計算では名前の扱いが非常に重要な役割を果たすが，我々のエンコーディングでは，名前を LMNtal のアトム名で表現するのではなくセル (名前セルと呼ぶことにする) によって表現する．そして名前の各出現を，その名前セルに入射するリンク ('+' アトムによって終端する) によって表現する．ただし受信形式 $x(y).P$ の y は仮引数の役割をもつため，対応する本体 P における y の出現と直接リンクで接続するものとする．P に y が出現しない場合や複数回出現する場合は，λ 計算のエンコーディングと同様，`rm` や `cp` アトムを用いてリンクの終端や分岐を行うものとする．

π 計算の名前の表現にアトムでなくてセルを用いるのは，次の動機と目的からである．

1. LMNtal のグラフ書換えの記述力を確認する．
2. π 計算の名前は通信チャネルであるのでそのトポロジを明示化する．
3. new によって導入される局所名を扱う．
4. LMNtal の名前は π 計算の固定的構成要素のエンコーディング用に限定する．

名前セルはアトム `name()` (大域名) または `name` (局所名) をもつものとするが，これらは人間が計算結果を理解するためであり，計算の過程では利用されない．

送受信形式は 3 価のアトム `snd`, `get` を用いて表現する．三つの引数は通信チャネル，通信内容，および通信成立後に実行するプロセスである．

π 計算の構文カテゴリ R は送受信形式 $p_j.P_j$ の多重集合を膜で囲ったものとして表現する．P は R の多重集合であるが，送受信形式の本体の P_j はその範囲を明示するために膜で囲むこととする．LMNtal へのエンコーディングでは，π 計算の名前の参照関係を名前セルとそこへの入射リンクで表現するため，局所名導入のための (new a) は無名の名前セル

```
comm@@ {$x,+C1,+C2}, {get(C1,Y,{$p[Y|*V1]}), $m}, {snd(C2,Z,{$q}),  $n} :-
          {$x}, $p[Z|*V1], $q,
          nlmem.kill({$m},rm), nlmem.kill({$n},rm).
 rm@@ {+C,$c}, rm(C) :- {$c}.
 cp@@ {+C,$c}, cp(C1,C2,C) :- {+C1,+C2,$c}.
gc1@@ {name($n)} :- unary($n) | .
gc2@@ {name} :- .
```

図 2.18　π 計算

にエンコードされる．一例として，プロセス

$$(\mathsf{new}\ z)(\overline{x}\langle y\rangle + z(w).\overline{w}\langle y\rangle \mid x(u).\overline{u}\langle v\rangle \mid \overline{x}\langle z\rangle)$$

は

```
{snd(X0,Y0,{}),get(Z0,W,{{snd(W,Y1,{})}})},
{get(X1,U,{{snd(U,V,{})}})},
{snd(X2,Z1,{})},
{name(x),+X0,+X1,+X2}, {name(y),+Y0,+Y1},
{name,+Z0,+Z1},
{name(v),+V}
```

と表現できる．

図 2.18 が π 計算のエンコーディングである．最初のルール comm は同じ名前を共有する送受信形式間の同期通信を表現している．プロセス文脈 $m, $n は通信の際に使われなかった選択肢の多重集合であり，簡約規則 (2.1) からもわかるように廃棄しなければならない．しかし $m, $n は名前への参照リンクを含むかもしれず，$m, $n の廃棄時にはそれらのリンクの一端を何らかのアトムに接続しなければならない．ライブラリ呼出し nlmem.kill({$m},rm) と nlmem.kill({$n},rm) は，それらの自由リンクを一価のアトム rm で終端するためのものである．

簡約規則 (2.1) の P と Q はプロセス文脈を用いて{$p[Y|*V1]} および {$q} とエンコードしている．プロセス文脈を囲む膜は，P および Q に属するプロセスを明示するだけでなく，膜の内容物をルールから隔離して計算が進まないようにする役割も果たしている．comm ルールの適用後はそれらの膜が除去されて実行が始まる．

ルール rm と cp は名前出現の消去と複製のためのものであり，gc1, gc2 は参照されなくなった名前の廃棄ルールである．

上記のプロセス

$$(\mathsf{new}\ z)(\overline{x}\langle y\rangle + z(w).\overline{w}\langle y\rangle \mid x(u).\overline{u}\langle v\rangle \mid \overline{x}\langle z\rangle)$$

の実行結果は非決定的であるが，-s オプションを与えて実行すると，結果

```
{y(name), +L7},
{v(name), +L6},
{snd(L6,L7,L8), {+L8}},
```

および

```
{name(x), +L91},
{y(name), +L55},
{snd(L55,L56,L57), {+L57}},
{name, +L92},
{snd(L91,L92,L93), {+L93}},
{v(name), +L56}
```

のいずれかがランダムに得られる．前者は $\overline{v}\langle y\rangle$ を，後者は $(\text{new } z)(\overline{y}\langle v\rangle \mid \overline{x}\langle z\rangle)$ を表す．LMNtal の略記法では `name(x)` と `x(name)` とは同一物であり，とちらも `x` と `name` がつながったプロセスの表現であることに注意してほしい．

独立にエンコードされた二つのプロセスを後から並列合成する場合は，それぞれのプロセスがもつ同一の大域名を同一視する必要があるが，そのためには，同一大域名を表す二つのセルを一つに併合するための下記のようなルールを用意すればよい．

```
{name($a), $m}, {name($b), $n} :-
    unary($a), unary($b), $a=$b |
    {name($a), $m, $n}
```

本節では，π 計算の '!' (replication) 機能は扱わなかった．$!P$ は P の任意個の複製を許す機能であるが，計算の発散を避けるためには複製をオンデマンドで行う必要がある．Replication 機能の主目的は他の言語の手続きに相当するものを表現することであるので，実用的観点からは，$\Sigma_{j=1}^{n} x_j(y_j).P_j$ の形の式に付加することができればほぼ十分である．この限定された '!' 機能は `nlmem` モジュールを利用して実装済である．

2.5.3 ambient 計算

Ambient 計算 [2] は，ambient と呼ばれる階層化可能な計算主体が ambient の階層構造の中を移動することを特徴とする計算モデルである．Ambient 計算は移動と通信の両側面を持つが，ambient 計算の核心部は移動機能にあるのでここではそのエンコード法について論じる．

Ambient 計算の式は，n を名前を表す構文カテゴリとして以下の構文で定義される．

$$\begin{aligned}(\text{processes})\ P ::= {} & (\nu n)P \mid \mathbf{0} \mid P|Q \\ & \mid\ !P \mid n[P] \mid M.P \\ (\text{capabilities})\ M ::= {} & \mathsf{in}\ n \mid \mathsf{out}\ n \mid \mathsf{open}\ n\end{aligned}$$

$n[P]$ が名前 n をもつ ambient を表す．in, out, open に関する簡約規則は

$$n[\mathsf{in}\ m.P \mid Q] \mid m[R] \to m[n[P \mid Q] \mid R]$$
$$m[n[\mathsf{out}\ m.P \mid Q] \mid R] \to n[P \mid Q] \mid [R]$$
$$\mathsf{open}\ n.P \mid n[Q] \to P \mid Q$$

である．

Ambient はプロセスを (名前つきの) 膜で囲ったものであるので LMNtal のセルで表現するのがもっとも自然である．また，in, out, open のエンコード法は π 計算の送受信のエンコードと同様の手法が使える．名前のエンコードにセルを用いるのも π 計算の場合と同じである．

Ambient 計算のエンコードが π 計算のエンコードと異なるのは，ambient を構成する膜の存在である．まず，ambient の内部で計算を局所的に進めるため，簡約規則は ambient 膜の中でも有効でなければならない．一方，in, out, open の本体部を表す膜の中のプロセスの計算は，移動が成功するまで進行させてはならない．この要請を満たすには，個々の ambient 膜の中に ambient 計算のルールセットを展開しなければならない．

次に，ambient 計算における名前は ambient のさまざまな階層から参照されるが，それらの名前の同一性は，大域的に管理すると同時に，計算を局所的に進めるためには局所的に検出することもできなければならない．この要請を満たすには，名前は大域的な名前セルとそこへの入射リンクという形で一元的に管理するのでなく，ambient 膜の各階層で，その内部で使われている各名前に対応するプロキシセルを保持する必要がある．個々のプロキシセルは

$$\{\mathtt{id}, \mathtt{-}L_0, \mathtt{+}L_1, \ldots, \mathtt{+}L_n\}$$

(L_0 は一つ外側のプロキシまたは名前セルへの参照リンク，$L_1, \ldots, L_n$ はセル内部からの参照リンク) という形を持ち，それぞれの名前は，プロキシセルの木構造の形で表現することになる．さらに，ambient が移動すると膜の階層構造が変化するが，各名前の参照を表現したプロキシセルの木構造もそれに伴って更新しなければならない．

これらのことを考慮した ambient 計算の簡約規則のエンコーディングは図 2.19 のようになる (π 計算の場合と同様，'!' に関するルールは省く)．我々は文献 [2] の多数の例題のエンコードを行ったが，その中でファイアウォールアクセスの例題をエンコードしたものを図 2.20 に示す．

この例は，非公開の ambient 名 `w` をもつ Firewall ambient の中に，鍵 `k`, `kk`, `kkk` を保有する Agent が外から入るためのプロトコルを表現している．`w` の中の ambient `k` が一旦外に出てエージェントを招き入れるというのが基本アイデアである．`pp`, `qq` は入室後実行すべきプロセス P および Q をそれぞれ表す．このプログラムの実行結果は

```
{pp, qq, amb(L629),
  {id, -L653, +L629}, @601},
```

```
{ module(a).

/* n[in m.P | Q] | m[R] --> m[n[P|Q] | R] */
in@@
{amb(N0), {id,+N0,-N1,$n}, {id,+M0,-M1,$m0}, in(M0,{$p}), $q,@q},
{amb(M2), {id,+M2,-M3,$m1}, $r,@r},
{id,+M1,+M3,$m2} :-
   {amb(M4), {id,+M4,+M5,-M9,$m1}, {id,+N3,-N1},
      {amb(N2), {id,+N2,-N3,$n}, {id,$m0,-M5},$p,$q,@q},
   $r,@r},
   {id,+M9,$m2}.

/* m[n[out m.P | Q] | R] --> n[P|Q] | m[R] */
out@@
{amb(M0), {id,+M0,+M2,-M3,$m1} {id,+N1,-N2,$n2},
  {amb(N0), {id,+M1,-M2,$m0}, {id,+N0,-N1,$n}, out(M1,{$p}), $q,@q},
  $r,@r} :-
    {amb(N), {id,-M2,$m0}, {id,+N,-N1,$n}, $p,$q,@q},
    {amb(M4), {id,+M4,-M5,$m1}, {id,-N3,$n2}, $r,@r},
    {id,+M2,+M5,-M3}, {id, +N1,+N3,-N2}.

/* open m.P | m[Q] --> P|Q */
open@@
open(M,{$p}), {amb(M1), {id,+M1,-M2,$mm}, $q,@q}, {id,+M,+M2,$m} :-
   $p, $q, {id,$m,$mm}.

proxy_enter@@
{$p[M0,M1|*P],@p}, {id,+M0,+M1,$m} :-
   {$p[M0,M1|*P],@p, {id,+M0,+M1,-M}}, {id,+M,$m}.

proxy_merge@@
{id,-M,$m0} {id,+M,$m1} :- {id,$m0,$m1}.

proxy_insert_middle@@
{{{id,-M,$m},$p,@p},$q,@q} :- {{id,+M0,-M}, {{id,-M0,$m},$p,@p},$q,@q}.

proxy_insert_outer@@
{{id,+M0,-M,$m0},$p,@p} :- {{id,-M1,$m0},$p,@p}, {id,+M1,+M0,-M}.

local_name_out@@
{{id,+M0},{$p[M0|*M],@p},$q,@q} :-
    {{id,+M0,-M},{$p[M0|*M],@p},$q,@q}, {id,+M}.

global_name_out@@
{{id,name($n),+M0},{$p[M0|*M],@p},$q,@q} :- unary($n) |
    {{id,+M0,-M},{$p[M0|*M],@p},$q,@q}, {id,name($n),+M}.

name_resolution@@
{id,name($n0),$m0}, {id,name($n1), $m1} :- unary($n0), unary($n1), $n0=$n1 |
  {id,name($n0),$m0,$m1}.

gc1@@ {id} :- .
gc2@@ {id,name($n)} :- unary($n) | .
gc3@@ {id,+X,$m}, {{id,-X}, $p,@p} :- {id,$m}, {$p,@p}.
gc4@@ a.use :- .

}.
```

図 2.19　ambient 計算

```
// Firewall Access ////////////////////////////////////////////////////
//   Firewall =def (new w) w[k[out w.in kk.in w] | open kk.open kkk.P]
//   Agent    =def kk[open k.kkk[Q]]
//////////////////////////////////////////////////////////////////////////

{id,name(k),+K9,+K3}, {id,name(kk),+L3,+L9}, {id,name(kkk),+M9,+M3}, {id,+W9},
{a.use. amb(W0),
  {id,+W0,+W8,-W9}, {id,+K8,-K9}, {id,+L1,+L8,-L9}, {id,+M0,-M9},
  {a.use. amb(K0), {id,+K0,-K8}, {id,+W1,+W2,-W8}, {id,+L0,-L8},
     out(W1,{in(L0,{in(W2,{})})})},
  open(L1,{open(M0,{pp})})
},
{a.use. amb(L2), {id,+L2,-L3}, {id,+K2,-K3}, {id,+M2,-M3},
  open(K2,{{amb(M1), {id,+M1,-M2}, qq}})
}.
```

図 2.20　ambient 計算の記述例

```
{id, +L653}, @601
```

となる．非公開名をもつ ambient の中に pp が入ることができたことを示している．

各 ambient は，構成要素のプロセスのほかに，ambient 名を表すプロセス amb(n) および簡約規則をロードするためのアトム a.use をもつ．さらに各 ambient のトップレベルには，上述のようにその ambient が使用しているすべての名前のプロキシが置かれる．

Ambient 計算の in, out, open に対応する LMNtal のルールはそれぞれ 1 本で書ける．proxy_ から始まる名前をもつルールは，ambient の階層構造の変化に伴って各 ambient の参照する名前の集合が変化した場合にプロキシの木構造を修正するためのものである．たとえば proxy_enter は，同一 ambient 内からの同一名の参照を，その ambient 内部のプロキシで認識するようにするためのものであり，proxy_merge は，同一名に対するプロキシが ambient 内に複数生成された場合に両者を併合するためのものである．これらの修正作業は ambient 計算本来の計算と並行して行われるが，修正が未完了の名前は in, out, open ルールの左辺とマッチできないことがある．逆にプロキシが正しく挿入されていない名前をこれらの 3 ルールが誤って認識して計算を進めることはない．

local_name_out, global_name_out は，名前プロキシの木構造を正規化するためのもので，ambient の階層構造の最外部に向かって木構造の根を伸張させる．ambient 計算においては ambient の移動に伴って局所名の有効範囲が動的に変化するので，局所名に対応する木構造も，ambient 階層の最外部が根となるように管理することによりルールの簡素化を図った．

gc から始まるルールは使われなくなった名前の整理と，モジュール読込みのためのアト

ムの消去のためのものである．モジュール a の内容を膜の中に読み込むのには，その膜の中に a.use と書いてモジュール a に言及するのが標準的な技法である．この a.use を計算に利用しない場合は，a の定義の中でそれを消去する．つまり a.use は，そこにモジュール a のルールセットが書いてあるものとみなすことができる．

2.5.4 CHR

CHR [3] は制約プログラミングのための宣言型言語として利用が広まっている．CHR と LMNtal とを比較すると，CHR は

1. 変数値が具体化するという概念を持つ点，
2. 変数の出現回数に制限がない点，
3. 階層構造をもたない点，
4. propagation ルールを持つ点

などの相違はあるものの，多重集合書換えに基づいている点をはじめとして LMNtal と多くの類似点も持つ．

CHR の構文と意味は次の通りである．

$$\text{(Simplification)}\quad \mathit{rulename}@H \Leftrightarrow G \mid B$$
$$\text{(Propagation)}\quad \mathit{rulename}@H \Rightarrow G \mid B$$

ここで H, G, B はそれぞれ原子論理式の多重集合である．Simplification ルールは，原子論理式の集合 H' が H にマッチ (すなわち $H' = H\theta$ となる代入 θ が存在) し，かつガード $G\theta$ が成り立ったら H' を消去して $B\theta$ を追加する．これは，変数への代入 θ を扱う点を除けば LMNtal のルールに近い．一方，propagation rule は H' を消さずに残したまま $B\theta$ を追加する．

ルール $H \Rightarrow G \mid B$ はルール $H \Leftrightarrow G \mid H, B$ の単なる略記法ではない．後者は同じ原子論理式の多重集合に対して 1 回でも適用可能ならば無限回適用可能であるが，前者は，ルール左辺とマッチした構造 H' に対して 1 回しか適用されない．

CHR の propagation ルールを LMNtal にエンコードする場合，

- 論理変数の扱い
- propagation ルールの適用履歴の管理

が課題となる．我々は，前者については，var(*name*) と表記した論理変数を内部ではセルで表現する手法を採用した．後者については，uniq ガード制約 (2.4.4 節) を利用して解決した．

uniq ガード制約を用いて不等式制約の伝播の例題を LMNtal で記述したものを図 2.21 に示す．この例題は CHR の

```
Reflexivity @@
   leq(v(X0),v(X1)), {+X0,+X1,$e} :- {$e}.
Antisymmetry @@
   leq(v(X0),v(Y0)), leq(v(Y1),v(X1)),
   {+X0,+X1,$e0}, {+Y0,+Y1,$e1} :-
      {$e0, $e1}.
Transitivity @@
   leq(v({name($x),$e0}),v(Y0)), leq(v(Y1),v({name($z),$e1})),
   {name($y),+Y0,+Y1,$e2} :- uniq($x,$y,$z) |
      leq(v({name($x),+NX,$e0}),v(Y0)), leq(v(Y1),v({name($z),+NZ,$e1})),
      {name($y),+Y0,+Y1,$e2}, leq(v(NX),v(NZ)).

VarDeclare @@
   H=var($name) :- unary($name) | H=v({name($name)}).
VarUnify @@
   {name($name0),$e0}, {name($name1),$e1} :- $name0=$name1 |
      {name($name0),$e0,$e1}.
```

図 2.21　不等式制約の伝播

```
 Reflexivity @ X leq X <=> true.
Antisymmetry @ X leq Y , Y leq X <=> X=Y.
Transitivity @ X leq Y , Y leq Z
                    ==> X leq Z.
```

というルール，すなわち伝播規則

$$X \leq X$$
$$X \leq Y \wedge Y \leq X \Leftrightarrow X = Y$$
$$X \leq Y \wedge Y \leq Z \Rightarrow X \leq Z$$

をエンコードしたものである．

不等式制約 $a \leq b \wedge b \leq c \wedge c \leq a$ からは $a = b = c$ を導き出せるが，我々のプログラムでは

```
leq(var(a),var(b)),
leq(var(b),var(c)),
leq(var(c),var(a))
```

を入力すると，処理系のルール適用戦略によるが，たとえば推移律 3 回，反対称律 2 回，反射律 2 回の適用ののち

```
{name(c), name(b), name(a)}
```

が最終結果として得られる．同一セルは同一変数を表現しており，name(a)，name(b)，name(c) が同じ膜にあるということは $a = b = c$ が推論できたことを表している．

2.6 まとめと今後の課題

本章では，階層グラフ書換えモデルとして提案した LMNtal の展開研究として確立を目指してきたプログラミング言語としての LMNtal について，言語機能および記述力の両側面から紹介と議論を行った．

Lisp や Prolog の例を見てもわかるように，実用となった宣言型言語の多くはベースとなった計算モデルから多くの拡張を行い，ベースモデルでは説明できない機能も多数導入している．LMNtal の場合もさまざまな拡張を行ったが，基本言語モデルとしての LMNtal の設計を保ちつつ実用言語機能を提供できたと考える．

LMNtal は多様な宣言型計算モデルや言語の統合を目標としているが，λ 計算，π 計算，ambient 計算，CHR という，それぞれ性格の異なる重要なモデルが簡潔にエンコードでき，かつその動作を多くの例題で確認できたことで，統合プログラミング言語としての LMNtal の能力を示すことができたと考える．

階層グラフ書換え言語の特徴は，膜による階層構造とリンクによる接続構造の両方を同時に扱えることである．膜は単にプロセスのグループ化や多重集合の表現だけでなく，ルールや計算の局所化，プロセスのルールからの保護など多くの役割を果たすが，本章の記述例でも，我々の設計した膜機能やそれに付随する文脈機能が，これらの目的に有効に利用できることがわかった．

グラフ書換えの理論研究では圏論などが駆使されてきたが，LMNtal は，扱う階層グラフやその書換えのクラスを限定することによって，計算を数学的予備知識なしに初等的かつ図形的に把握することを可能にした．

本章では基本的な記述能力の立証のための例を中心に紹介したが，それ以外にも，並列分散処理，宣言型グラフィクス，およびビジュアルプログラミング環境の整備などが進行中であり，理論計算機科学から実践分野までの広範囲をカバーするプログラミング言語システムの確立を目指している．

今後の課題であるが，理論面では型体系の設計・実装および処理系との統合が重要課題である．その意義は論をまたないであろう．たとえば，これまでに蓄積した記述例から膜やリンクの役割を分析することにより，プログラミングと実装の両方に役立つ型体系が得られるものと確信している．LMNtal の膜は多くの用途をもつが，多様な用途を単一の言語要素に負担させることは必ずしも好都合でないという評価もある．型体系はこの問題点に対する解決を与えるものと期待される．

我々が開発した処理系の詳細は第 3 章に譲るが，その開発においては，宣言型言語の既存

実装技術から容易に発想できる技法の適用はあえて後回しにして，LMNtal 特有の実装上の問題の同定と解決に努めてきた．LMNtal の表現力を考慮すれば，現在得られている性能は高性能計算用途以外では実用の域に達していると言えるが，今後はプログラム解析とコンパイラ最適化にも力を入れてゆく予定である．

記述例の蓄積では，Bigraphical Reactive System (BRS) [6] や線形論理型言語の LMNtal への埋め込みが特に興味深い課題である．前者は LMNtal と同様に階層構造と接続構造の両立を設計動機としている点で LMNtal と深く関連しており，エンコーディングができれば BRS のプログラミング言語への展開にも寄与できる．一方線形論理型言語は資源の扱いに関して強力な表現力をもつが，LMNtal もデータを資源として扱う立場の言語であり，両者を関連付ける意義は大きい．

参考文献

[1] Banâtre, J.-P. and Le Métayer, D. : Programming by Multiset Transformation, *Commun. ACM*, Vol. 35, No. 1 (1993), pp. 98–111.

[2] Cardelli, L. and Gordon, A. D. : Mobile Ambients, in *Foundations of Software Science and Computational Structures*, Nivat, M. (ed.), LNCS 1378, Springer-Verlag, 1998, pp. 140–155.

[3] Frühwirth, T. : Theory and Practice of Constraint Handling Rules, *J. Logic Programming*, Vol. 37, No. 1–3 (1998), pp. 95–138.

[4] Lafont, Y. : Interaction Nets, in *Proc. POPL'90*, ACM, pp. 95–108.

[5] Milner, R. : The Polyadic pi-Calculus: A Tutorial, in Bauer, F. L., Brauer, W. and Schwichtenberg, H. (eds.), *Logic and Algebra of Specification*, Springer-Verlag, 1993, pp. 203–246.

[6] Milner, R., Bigraphical Reactive Systems. In *Proc. 12th Int. Conf. on Concurrency Theory (CONCUR 2001)*, LNCS 2154, Springer-Verlag, 2001, pp. 16–35.

[7] Saraswat, V. A., Kahn, K. and Levy, J. : Janus: A Step Towards Distributed Constraint Programming, in *Proc. 1990 North American Conf. on Logic Programming*, MIT Press, 1990, pp. 431–446.

[8] Sinot, F.-R. : Call-by-Name and Call-by-Value as Token-Passing Interaction Nets, in *Proc. 7th Int. Conf. on Typed Lambda Calculi and Applications (TLCA 2005)*, LNCS 3461, Springer-Verlag, 2005, pp. 386–400,

[9] Ueda, K. : Experiences with Strong Moding in Concurrent Logic/Constraint Programming, in *Proc. Int. Workshop on Parallel Symbolic Languages and Systems (PSLS'95)*, LNCS 1068, Springer-Verlag, 1996, pp. 134–153.

[10] Ueda, K. : Resource-Passing Concurrent Programming, in *Proc. 4th Int. Symp. on Theoretical Aspects of Computer Software (TACS 2001)*, LNCS 2215, Springer-

Verlag, 2001, pp. 95–126.

[11] Ueda, K. and Kato, N. : LMNtal: A Language Model with Links and Membranes, in *Proc. Fifth Int. Workshop on Membrane Computing (WMC 2004)*, LNCS 3365, Springer-Verlag, 2005, pp. 110–125.

[12] 上田和紀，加藤紀夫 : 言語モデル LMNtal，コンピュータソフトウェア，Vol. 21, No. 2 (2004)，pp. 44–60.

[13] 工藤晋太郎，加藤紀夫，上田和紀 : LMNtal 処理系におけるグラフ構造の操作機能の設計と実装，情報科学技術レターズ, Vol. 4 (2005)，pp. 9–12.

[14] 水野謙，加藤紀夫，原耕司，上田和紀 : 階層グラフ書換え言語 LMNtal 処理系における非同期実行の実現，日本ソフトウェア科学会第 22 回大会講演論文集，3A-4，2005，pp. 213–221.

[15] 矢島伸吾，永田貴彦，加藤紀夫，上田和紀 : LMNtal プロトタイプ処理系の設計と実装，日本ソフトウェア科学会第 20 回記念大会論文集，1A-5，2003，pp. 21–25.

第 3 章

LMNtal 処理系の設計と実装

LMNtal は階層グラフ書換えに基づく言語モデルであり，リンク構造による接続構造と膜による階層構造の表現・操作機能によって，動的データ構造や多重集合書換えを扱うプログラムを簡潔に記述することができる．LMNtal は書換え規則の適用を単位とする細粒度の並行性をもっており，正しく効率的な実装方式は自明でない．そこで言語処理系を Java を用いて開発し，効率をできるだけ犠牲にせずに正しく動作する実装方式を確立した．処理系は中間命令列へのコンパイラ，その解釈実行系及び Java ソースへのトランスレータからなり，他言語インタフェースをはじめとするさまざまな有用な機能を備えている．複数の膜を貫くリンク構造を正しくつなぎかえるための処理や，複数の膜にある書換え規則を正しく非同期実行させるための工夫も行っている．本章では，処理系開発において主要な技術的課題となった階層グラフ構造の保持方法，中間命令体系，安全な非同期実行方式等を中心として，公開中の LMNtal 処理系の設計と実装について論じる．

3.1 はじめに

3.1.1 LMNtal 言語モデル

LMNtal [12] は階層グラフ書換えに基づく言語モデルであり，計算モデルとしての簡潔性とプログラミング言語としての実用性の両立を目指している．LMNtal の研究開発は，多様な理論計算モデル，特に並行計算と多重集合書換えに基づく計算とを統合する計算モデルを確立するとともに，並行性を持つ多様なコンピューティング環境のためのプログラミング言語の基礎概念とその実装を提示することを目標として 2002 年度から推進してきた．

我々が上記の要求を満たす表現手段として着目して LMNtal の基本データ構造に採用した階層グラフは，アトムを基本構成要素として，それを膜とリンクの二つの手段で構造化したものである．膜はアトムの多重集合 (要素の個数の概念を持つ集合) を構成し，多重集合は入れ子にできる．またリンクはアトム同士を一対一で接続する．どちらの構造も動的再構成が可能である．階層構造と接続構造は，計算システム，社会組織，生体などさまざまな

場面に遍在する構造化の仕組みであり，しかも両者はしばしば併存する．我々が表現手段として階層グラフ構造に着目したのもこのことが動機となっている．階層グラフの枠組は，グラフ書換えや知識表現分野でとりあげられたことはあるが [1][2]，階層グラフ書換えに基づく具体的なプログラミング言語やその本格的実装は存在しなかった．

LMNtal の特徴の一つとして，参入障壁の低さが挙げられる．情報系の大学 1 年生に 1 時間強の説明をしただけで，多くの学生が，格子構造グラフの生成や簡単な分散アルゴリズムなど，他の言語では容易には解けない課題をこなすことができた．これは，LMNtal の計算が図的に解釈可能であることとも関連している．我々は，並行計算や動的データ構造などの概念を説明したり理解したりするときに図を援用するが，LMNtal は対象の図的表現 (に直接対応するプログラム表現) をそのまま操作対象にできる言語である．

3.1.2 LMNtal 処理系とその開発目的

LMNtal プロジェクトは 2002 年度に始まった．年度前半の週 1 回の検討会で基本設計を固めたのち，年度後半からは理論面の検討と並行して処理系の開発も行ってきた．同年度末までに第 1 次処理系の試作 [15] を行った後，その経験をもとに 2003 年度初頭には Ruby で第 2 次試作処理系を作成した．本章で扱う処理系は現在のプロジェクトの中核をなす第 3 次処理系であり，2003 年度前半から概念設計，詳細設計，実装を進めてきた．本処理系は逐次計算機上で動作する形で公開されているが，将来の並列分散処理への展開のベースとなるべく，非同期実行機能 [9] の設計と実装を行っていることを重要な特徴としている．

LMNtal は単純な言語であるが，細粒度の並行性をもっている点に加えて，多重集合が膜によって階層化できる点，膜が局所的な書換えルールを持てる点，およびルールセットが移送可能である点などから，正しくかつ効率的な実装方式は自明でない．また階層グラフ書換えに基づくプログラミング言語の実装には先行研究がないため，実装方式については入念に検討を重ねてきた．逐次処理に限定した実装を目指すならば，言語の操作的意味論を記号処理言語を用いて忠実に表現すれば一応動作する処理系が得られるが，それでは並列・分散処理系への拡張が困難である．したがって複数タスクの非同期実行を許す実装方式 (第 3.5 節) が，第 3 次処理系の研究開発における重要な研究課題となり，多くのバージョンを重ねてきた．

本章で紹介する処理系は Java を用いて開発している．Java を採用した理由は，開発メンバの使用経験がもっとも豊富な言語であること，開発効率が高いこと，およびすぐれた統合開発環境 (Eclipse) が存在することである．一方，Java の採用に伴い，バイト単位のメモリ使用量の削減やネイティブ環境に応じた最適化などは当面の目標とはしないこととなった．

本章では，LMNtal の基本概念について簡単な解説を行った後，処理系の中核機能である実行時処理系と中間命令体系についてそれぞれ解説を行う．主要な技術的課題となった階層グラフ構造の操作法，中間命令列の最適化法，安全な非同期実行方式については，検討過

程も含めて詳しく解説する．基本言語モデルに含まれない拡張言語仕様やプログラミング環境については，その実装法を簡単に紹介する．

本章に示すプログラム例はすべて，我々のプロジェクトで開発したLMNtal処理系で動作確認済である．このLMNtal処理系はWeb上で公開されており[*1]，Javaプラットフォームの上で稼働する．

3.1.3 本章の構成

以下，第2節ではLMNtalの基本概念について説明する．第3節では開発したLMNtal処理系の機能概要と設計指針について述べる．第4節では実行時処理系のデータ構造と制御構造について説明する．第5節では非同期実行機能の詳細について述べる．第6節では中間命令列の実行を担当する抽象機械の基本概念を説明したあと，中間命令列の動作を具体例に即して解説する．第7節では中間命令列を書換えを含むいくつかの最適化技法について述べる．第8節では拡張言語仕様の実装を，第9節ではプログラム開発支援機能の実装を説明する．第10節では関連研究について述べ，第11節ではまとめと今後の課題について述べる．

LMNtalの形式的操作的意味論など，言語モデルとしてのLMNtalに関する技術的詳細や関連研究については，文献[12][13]および第1章を参照してほしい．また，LMNtal処理系において実現された実用言語としての諸機能の解説は文献[5]および第2章を参照してほしい．

3.2 LMNtal言語モデル

3.2.1 LMNtalの基本構成要素

LMNtalの基本用語を手短に定義する．(m価の)**アトム**はアトム名とm個($m \geq 0$)の順序づけられた引数からなる．それぞれの引数は(自分自身または他の)アトムの引数につながるリンクの端点である．**リンク**はリンク名を用いて表記し，同じリンク名をもつアトムの引数どうしが相互接続されていることを表す．LMNtalでは後述の構文条件(リンク条件)を設けて，アトムとリンクの両者が形成する構造が(ハイパーグラフではない)無向グラフ構造を表すようにしている．リンク名には英大文字から始まる名前，アトム名にはリンクと区別できる名前を用いる．数値や一部の特殊記号(+, -, =等)，一重引用符または二重引用符で囲んだ名前もアトム名として利用できる．予約アトム名=は**コネクタ**と呼ばれ，アトムX=YはリンクXの一端とYの一端とを接続する機能をもつ．

アトムの多重集合は**膜**{...}で囲むことができ，囲ったものを**セル**と呼ぶ．膜は入れ子構造をなすことができ，アトム，リンク，膜によって階層グラフが形成される．この階層グ

[*1] http://www.ueda.info.waseda.ac.jp/lmntal/

$$
\begin{array}{lll}
P ::= \mathbf{0} & & \text{(空)} \\
\quad \mid\ p(X_1, \ldots, X_m)\ (m \geq 0) & & \text{(アトム)} \\
\quad \mid\ P, P & & \text{(分子)} \\
\quad \mid\ \{P\} & & \text{(セル)} \\
\quad \mid\ T :\!\!- T & & \text{(ルール)} \\
T ::= \mathbf{0} & & \text{(空)} \\
\quad \mid\ p(X_1, \ldots, X_m)\ (m \geq 0) & & \text{(アトム)} \\
\quad \mid\ T, T & & \text{(分子)} \\
\quad \mid\ \{T\} & & \text{(セル)} \\
\quad \mid\ T :\!\!- T & & \text{(ルール)} \\
\quad \mid\ @p & & \text{(ルール文脈)} \\
\quad \mid\ \$p[X_1, \ldots, X_m \mid A]\ (m \geq 0) & & \text{(プロセス文脈)} \\
\quad \mid\ p(*X_1, \ldots, *X_m)\ (m > 0) & & \text{(アトム集団)} \\
A ::= [] & & \text{(空)} \\
\quad \mid\ *X & & \text{(リンク束)}
\end{array}
$$

図 3.1　LMNtal の構文

ラフ構造を LMNtal では**プロセス**と呼ぶ.

階層グラフ構造の書換え規則を**ルール**と呼ぶ. ルール $Head$:- $Body$ の左辺 ($Head$) は書き換えるべき階層グラフ構造のテンプレートであり, 右辺 ($Body$) は書換え後の階層グラフ構造のテンプレートである. 各膜はルールを 0 個以上保持することができ, その多重集合を**ルールセット**と呼ぶ.

3.2.2　LMNtal の構文

LMNtal の構文は, X_i を**リンク名**, p を**アトム名**として図 3.1 のように定義される. P は**プロセス**である. T はプロセスの書換え規則の表現に用いる**プロセステンプレート**であり, 局所文脈 (特定のセルの内部での文脈) を扱う機能をもつ.

$\mathbf{0}$ は中身のないプロセス, $p(X_1, \ldots, X_m)$ は m 価アトム, P, P はプロセスの並列合成, $\{P\}$ は膜 { } によってグループ化されたプロセス, T :- T はプロセスの書換え規則である.

LMNtal におけるプロセスは, **同じリンク名が 2 回を超えて出現してはならない**という**リンク条件**を満たさなければならない. あるリンク名の各出現はそのリンクの端点を表し, それらの集合がリンクを表す. プロセス P に 1 回だけ出現するリンク名は P の**自由リンク** (P の外部につながるリンク) を表し, P に出現するそれ以外のリンク名は P の**局所リ**

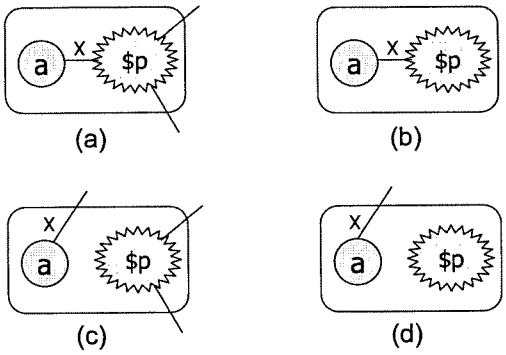


図 3.2　プロセス文脈の自由リンク

ンクを表す.

さらに個々のルールの各リンク名は，そのルール内にちょうど 2 回出現しなければならない．ルール左辺または右辺に 2 回出現するリンク名はそれぞれリンクの存在検査と生成を表し，左辺と右辺に 1 回ずつ出現するリンク名はリンクの引継ぎを表す.

ルールを膜に入れることができ，膜は計算の局所化のために利用できる．ルールは，そのルールが所属する膜やその子孫膜の内容を書き換えることができるが，親膜の内容を書き換えることはできない.

ルール文脈は膜の中のすべてのルールの多重集合とマッチし，**プロセス文脈**は膜の中のルール以外のプロセスのうち，明示的に指定されていないもの全体とマッチする．個々のルール中の文脈の出現はいくつかの構文条件を満たさなければならない [12]．アトム，ルール文脈，プロセス文脈は，同じ名前 p を持っていても互いに無関係である.

プロセス文脈の引数は，自由リンクの出現に関する制約条件を指定するものである．ルール左辺のプロセス文脈 $\$p[X_1,\ldots,X_m|A]$ の引数 $X_1,\ldots,X_m$ は，その文脈が持っていなければならない自由リンクを指定しており，これをプロセス文脈の**明示的な自由リンク**と言う．剰余引数 A が $*X$ の形の場合，$*X$ はその文脈が持つ $X_1,\ldots,X_m$ 以外の 0 本以上の自由リンク (**明示的でない自由リンク**と言う) の束を表し，[] の場合は $X_1,\ldots,X_m$ 以外に自由リンクがないことを表す.

例 3.2.1　プロセステンプレートとプロセスのマッチングの可否を例で示す．図 3.2 に示すプロセス (a)〜(d) (`$p` は図示するような自由リンクをもつ階層グラフ構造) があるとき，

- `{a(X), $p[X|*Y]}` は (a) と (b) にマッチし，
- `{a(X), $p[X|[]]}` は (b) にマッチし，
- `{a(X), $p[|*Y]}` は (c) と (d) にマッチし，
- `{a(X), $p[|[]]}` は (d) にマッチする.　□

アトム集団は指定された種類のアトムを一度に複数個生成するための構文で，ルール右

辺にのみ置くことができる．アトム集団は，明示的でない自由リンクを持つプロセス文脈を複製，廃棄するときに必要となる．

膜の階層構造に関する議論をするときには，対象となるプロセス全体を含む仮想的な膜を考え，その膜を**世界的ルート膜**と呼ぶ．

3.2.3 基本型と拡張構文

実用プログラミング言語では，数値型をはじめとする基本的な型とその基本演算を提供することは不可欠である．

LMNtal では，個々の数値を，8(X) や 3.14(Y) のように，その数値をアトム名とする 1 価アトムで表現する．数値アトムの引数はその数値を参照するプロセスにつながる．LMNtal では数値もプロセスであり，数値型をはじめとする基本型 (組込みの型) はプロセス型 (プロセスに対する型)[5] の一種である．

アトムが基本型に属するかどうかの検査や基本型に対する演算を指定するために，LMNtal では，**型付きプロセス文脈**という拡張構文を導入している．通常のプロセス文脈のマッチする相手が膜 (階層構造) によって決定されるのに対し，型付きプロセス文脈のマッチする相手はグラフ構造 (接続構造) とグラフ中のアトム名によって決定される．マッチする相手を指定するために，拡張構文として**ガード**つきルールを用意して，ルールの適用のための付帯条件を指定できるようにしている．

$$Head \texttt{ :- } Guard \texttt{ | } Body$$

例 3.2.2 ガードを持つルール

```
a(X), $n[X] :- int($n), $n>0 | a(Y), $n[Y], a(Z), $n[Z]
```

は，1 価アトム a が正整数アトムにつながっている場合，その構造の複製を作ることを表している．ここでガード条件 int($n) は，$n[X] が整数アトムを表現する型付きプロセス文脈であることを求めている．また $n>0 はその整数値が正であることを求めている．$n>0 は同時に $n[X] が整数アトムであることも求めるので，上の例は次のように書いてもよい．

```
a(X), $n[X] :- $n>0 | a(Y), $n[Y], a(Z), $n[Z]
```

□

現在の処理系でガード条件として指定できるプロセス型は，unary, int, float, string, class, ground である．これらはそれぞれ 1 価アトム，整数，浮動小数点数，文字列，他言語インタフェースを用いて作成した Java オブジェクトを保持するアトム，および自由リンクを一本だけ持つ無階層グラフを表す．

型付きプロセス文脈はこのように，指定されたプロセス型の構造をもつプロセスにマッチするが，型付きでないプロセス文脈と違い，膜の外に出現することや同じ階層に複数個出

現することも許される.

その他の拡張構文として，膜に対するプラグマ付加機能があるが，これは第 3.5 節で紹介する.

3.2.4 省略構文

階層グラフ構造やルールが簡潔に記述できるように，いくつかの省略構文を用意している.

1. $a(s_1,\ldots,s_m), b(t_1,\ldots,t_n)$ は，s_i $(1 \leq i \leq m)$ と t_n が同じリンクならば，$a(s_1,\ldots,s_{i-1},b(t_1,\ldots,t_{n-1}),s_{i+1},\ldots,s_m)$ と略記できる．例えば `x(X)`, `y(Z,X),z(Z)` は `x(y(z))` と等しい.
   木構造を LMNtal で表現するときは，n 分木のノードを，その参照者からのリンクを最後に追加した $n+1$ 引数のアトムとして表現するのが標準であるが，本略記法は，そのような形式の木構造グラフを通常の項 (term) の形式で記述するためのものである.
2. P をプロセス，A をリンクとすると $p(\ldots,A,\ldots)$, {+A, P} は $p(\ldots,$ {P},$\ldots)$ と略記してよい．p はセル {+A, P} を参照するアトムとみなすことができ，参照を表すリンク A を終端するのに 1 価アトム名 + が標準的に使われる.
3. プロセス文脈のリンク束が空のときは `|[]` を省略してよい．またそのように略記したプロセス文脈に 1. の略記法を援用して，$p(\ldots,A,\ldots)$, `$q[`A`]` を $p(\ldots,$ `$q`$,\ldots)$ と略記してよい.
4. ルールに `$`p`[|*`X`]` の形の同一のプロセス文脈が 2 回出現するならば，両方を同時に `$`p と略記してよい.

このほか，**0** は何も書かないことによって表すこと (ただし分子 (並列合成) の中に現れることはできない)，0 価アトムの括弧 () が省略できること，2 価の =, > 等が中置演算子として使えること，1 価の +, - が前置演算子として使えること，などはプログラム例から明らかであろう.

上記の略記法の帰結として，`f(3)` と `3(f)` はどちらも 1 価の `f` および 1 価の 3 からなる同一のプロセス (`f(X),3(X)`) を表す．さらに LMNtal は，相互接続されたアトムとコネクタに関して

$$p(\ldots,X,\ldots),X=Y \equiv p(\ldots,Y,\ldots)$$

という構造合同規則 (0 ステップの相互変換規則，3.2.5 節) をもっている．これを用いると，(`f(X),3(X)`) は (`f(X),3(Y),X=Y`) あるいは (`f(X),3(Y),Y=X`) と展開でき，それぞれに略記法を適用すると `f=3` および `3=f` が得られる．これらはすべて同じプロセスを表す．このように，コネクタ = は等号ではなくてアトム間の相互接続を表す記号として多用される.

また，リストの表記にはPrologの記法が利用できる．

例 3.2.3　2本のリストを連結するLMNtalプログラムは，標準記法ではリスト構成子を表すアトム名'.'と，空リストを表すアトム名'[]'を用いて次の2つのルールで表される．

```
append(X0,Y,Z0), '[]'(X0) :- Y=Z0.
append(X0,Y,Z0), '.'(A,X,X0) :- '.'(A,Z,Z0), append(X,Y,Z).
```

略記法を用いると，上のプログラムは論理型言語風に

```
append([],Y,Z) :- Y=Z.
append([A|X],Y,Z0) :- Z0=[A|Z], append(X,Y,Z)
```

と書くことも，項書換え系のように

```
Z=append([],Y) :- Z=Y.
Z=append([A|X],Y) :- Z=[A|append(X,Y)].
```

と書くことも可能である．□

ルールや初期プロセスは，上の例のようにピリオド止めにして並べることも許されている．またデバッグやプロファイリングのために，ルールの先頭に *rulename*@@ の形でルール名を書くこともできる．

3.2.5　操作的意味論

LMNtalの操作的意味論 (図3.3) は 構造合同関係 $\equiv$ を定義する規則 (E1)–(E10) と，簡約関係 $\longrightarrow$ を定義する規則 (R1)–(R6) からなる．(E1)–(E3) は分子が多重集合であることの特徴づけである．(E4) は α 変換を表し，(E5)–(E6) は $\equiv$ を合同関係 (congruence) にするための構造規則である．(E7)–(E10) はコネクタ = に関する規則である．(E7) は自己完結的ループと **0** との等価性を，(E8) は = の対称性を表す．(E9)–(E10) はアトムとセルによる = の吸収放出規則をそれぞれ表す．

簡約関係の中の (R1)–(R3) は標準的な構造規則，(R4)–(R5) は = の移動規則である．図3.3の中心的規則は (R6) で，階層グラフ構造の中の同じ「場所」に存在するグラフ構造とルールとの反応を表現している．代入 θ はプロセス文脈やルール文脈やアトム集団を具体的なプロセス，ルール，アトムに対応づけるものである．

操作的意味論の設計における重要な研究課題は，リンクが形作る接続構造と膜が形作る階層構造の相互作用，つまり膜を貫くリンクを適切に取り扱うことであった．接続構造と階層構造の相互作用の扱いは，意味論だけでなく後述の実装においても重要な研究課題の

(E1) $\mathbf{0},P \equiv P$ (E2) $P,Q \equiv Q,P$ (E3) $P,(Q,R) \equiv (P,Q),R$

(E4) $P \equiv P[Y/X]$ ただし X は P の局所リンク名

(E5) $P \equiv P' \Rightarrow P,Q \equiv P',Q$ (E6) $P \equiv P' \Rightarrow \{P\} \equiv \{P'\}$

(E7) $X=X \equiv \mathbf{0}$ (E8) $X=Y \equiv Y=X$

(E9) $X=Y,\ P \equiv P[Y/X]$ ただし P はアトムで X は P の自由リンク名

(E10) $\{X=Y,\ P\} \equiv X=Y,\ \{P\}$ ただし X と Y のちょうど一方が P の自由リンク名

(R1) $\dfrac{P \longrightarrow P'}{P,Q \longrightarrow P',Q}$ (R2) $\dfrac{P \longrightarrow P'}{\{P\} \longrightarrow \{P'\}}$

(R3) $\dfrac{Q \equiv P \quad P \longrightarrow P' \quad P' \equiv Q'}{Q \longrightarrow Q'}$

(R4) $\{X=Y,P\} \longrightarrow X=Y,\{P\}$ ただし X と Y は $\{X=Y,P\}$ の自由リンク名

(R5) $X=Y,\{P\} \longrightarrow \{X=Y,P\}$ ただし X と Y は P の自由リンク名

(R6) $T\theta,(T:-U) \longrightarrow U\theta,(T:-U)$

図 3.3 LMNtal の操作的意味論

一つとなった.

3.3 処理系の概要と設計方針

公開している LMNtal 処理系は約 49,500 行の Java コードから成る．その中核をなす実行時処理系は 17,300 行，コンパイラは 16,700 行である．Eclipse と CVS を用いてチーム開発を行ってきた．開発にかかわった人数は 18 名である．

3.1.2 節で紹介した現処理系の開発目的と指針は以下のようにまとめることができる.

- LMNtal プログラム実行環境を早期に提供する.
- 実装研究から言語設計へのフィードバックを行う.
- 階層グラフ書換えの基本実行方式を確立する.
- 階層グラフ書換えの非同期実行方式を確立する.
- 最適化研究の土台を提供する.
- 拡張機能の研究開発の土台を提供する.

これらのことから，現処理系は言語モデルの構文が定めるプロセスの構造や操作的意味論が定める簡約過程に忠実に設計することとして，構文や操作的意味論との直接的対応関係を保存しない最適化は行わないこととした.

一方，個々の操作の実装にあたっては，望ましい計算量 (computational complexity) の実現を重視し，計算量を改善しない最適化は必要に応じて適宜進めることとした.

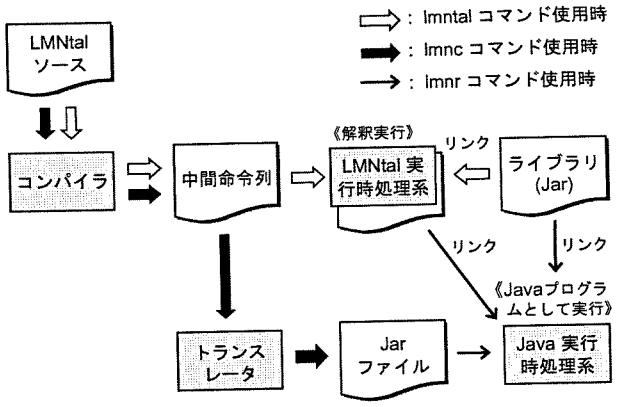


図 3.4　コンパイル・実行の手順

また基本言語モデルにない機能拡張 [5] やプログラミング支援機能等も多々設計，実装してきた．膜を貫くリンクの実装などの実装上のノウハウの多くは，プロトタイプ処理系 [15] から受け継いでいる．

3.3.1　処理系の全体像

処理系は大きく分けて，コンパイラと実行時処理系からなる．

コンパイラは字句解析器，構文解析器，ルールコンパイラ，最適化器等からなっており，LMNtal ソースプログラムを独自に設計した中間命令列へと変換する．実行時処理系は大きく分けて，実行時データ構造の操作機能，および中間命令列で表現したルールの実行制御機能を提供する．

コンパイラが生成した中間命令列の実行のしかたは次の 2 通りの方式を提供している．

1. `lmntal` コマンドを入力してシステムを起動すると，LMNtal ソースコードを中間命令列にコンパイルした後，LMNtal 抽象機械が中間命令列を解釈実行する．
2. `lmnc` コマンドを入力してコンパイラを起動すると，中間命令列はトランスレータにより Java コードに変換される．生成された jar ファイルを指定して `lmnr` コマンドを入力すると LMNtal プログラムが実行される．

これら一連の流れを図 3.4 に示す．

2 つの方式は，どちらも同じ実行時データ構造と実行方式に基づいている．本節ではまず第 3.4 節で実行時データ構造の操作機能と実行制御機能について述べ，中間命令列とその解釈実行については第 3.6 節で詳述する．中間命令列の最適化とトランスレータについては第 3.7 節で解説する．

なお，LMNtal プログラムを指定せずに `lmntal` コマンドを起動した場合は対話モード (read-eval-print-loop) となり，1 行ごとに中間命令列へのコンパイルとその解釈実行が行

われる．この場合，LMNtal 標準ライブラリ (3.8.1 節) は実行時に動的にリンクされる．

現処理系は基本言語モデルのうちアトム集団を除く全機能を実装し，アトム集団機能については代替機能をライブラリで提供している (3.8.3 節)．さらに拡張言語機能として，モジュールシステム，他言語インタフェース等を設計・実装しており (第 3.8 節)，プログラム開発支援のための諸機能も備えている (第 3.9 節)．

3.3.2 単一スレッドによる実行方式の概要

LMNtal プログラムの実行は，ルールを用いた階層グラフ書換えの繰返しである．階層グラフの初期構造はルールとともにプログラムによって与えられ，ルールの適用が不可能になったら実行を終了する．LMNtal の実行には，適用するルールの選択とそのルールが適用するプロセスの選択の双方に非決定性があり，ルール適用の戦略は処理系に委ねられている．

本節では上記の作業を単一スレッドで行う方法の基本を説明する．以下では，あるルールが現在の階層グラフのいずれかの部分に対して適用可能かどうかの検査を**テスト**と呼び，あるルールがどの部分構造に適用できるかどうかの探索を**マッチング**と呼ぶ．

テストの方法として，我々は**アトム主導テスト**および**膜主導テスト**の 2 種類を設計，実装した．

アトム主導テストは，書き換えられる可能性のあるアトムを**アトムスタック**によって管理し，取り出したアトムを出発点としてテストを行う．一方，膜主導テストは，適用できる可能性があるルールを持つ膜を**実行膜スタック**によって管理し，取り出した膜の各ルールについてテストを行う．中間命令列もそれぞれのテストを用意している．実行方式を 2 種類用意したのは，両者が相補的な関係にあるためである．アトム主導テストは高速なマッチングを実現するが，単体では完全性 (可能な書換えが存在する限り実行が停止することはないという性質) を欠く．これに対して膜主導テストは完全性は保証されるが，ルールの実行には時間がかかる．

いずれの実行方式でも，複数個の膜がある場合には「次にどの膜のルールの適用をテストするか」と「どの膜のルールが適用できる可能性があるか」を知る必要がある．実行膜スタックはこの処理を O(1) 時間で行うことを可能にする．実行膜スタックは，ある膜 M が積まれているならば M の親膜は必ず M よりも底方向に積まれているという不変条件 (invariant) を満たすように管理される．実行膜スタック先頭の膜 (**本膜**と呼ぶ) に対してルール適用を繰り返し，適用できなくなったら実行膜スタックから取り除く．実行膜スタックに積まれていない膜の中のプロセスを書き換えた際は，その膜を実行膜スタックに積む．この処理を膜の**活性化**と呼ぶ (3.5.4 節)．

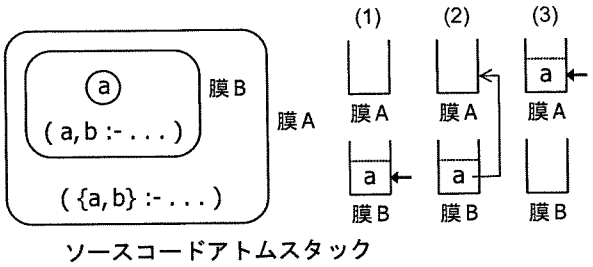


図 3.5　アトムのアトムスタック間の動き

アトム主導テスト

アトム主導テストは，「新たに生成されたアトムが，ルールによる書換えの対象となるかどうか」をテストするものであり，テスト対象となるアトムの管理のためにアトムスタックを使用する．テスト対象となるアトムを，**主導するアトム**と呼ぶ．

アトムスタックは膜ごとにちょうど 1 つある．膜内のルールは，入れ子になった膜の内部のアトムを書き換えることもできるので，子孫膜の内部に作られたアトムもアトムスタックに積まれる．処理系はスタックから取り出したアトムに対し，膜の持つルールの適用を試みる．ルールが適用できたときは，ルール右辺の各アトムをそのアトムが所属する膜のアトムスタックに積み，適用失敗時は取り出したアトムをルールの親膜のアトムスタックに積む (親膜がないときは何もしない)．すべての膜のアトムスタックが空になったら，アトム主導テストの実行は終了である．

例 3.3.1　図 3.5 にアトムのアトムスタック間の動きを示す．この図は，(1) a にルール (a,b :- ...) を適用しようとして失敗し，(2) a を親膜のアトムスタックに積み，(3) a にルール ({a,b} :- ...) を適用しようとして失敗した状況を表している．□

実行開始時のデータ構造生成は ":- 初期データ構造" というルールの適用とみなす (3.4.2 節) ため，生成直後にはすべてのアトムがスタックに積まれる．また，スタックであるので直前のルール適用で生成されたアトムがスタック先頭にくる．これにより，末尾再帰的プログラムを効率よく処理することができる．

例 3.3.2　プログラム

{a}, {b}, ({a} :- {c})

をアトム主導テストで実行したときのアトムスタック間の動きと，実行膜スタックの動きを図 3.6 に示す．

(1) 膜 C (本膜) のアトムスタックから取り出したアトム b に対するルール適用が失敗し (膜 C にはルールがないため)，b を膜 A (親膜) のアトムスタックに積む．膜 C のア

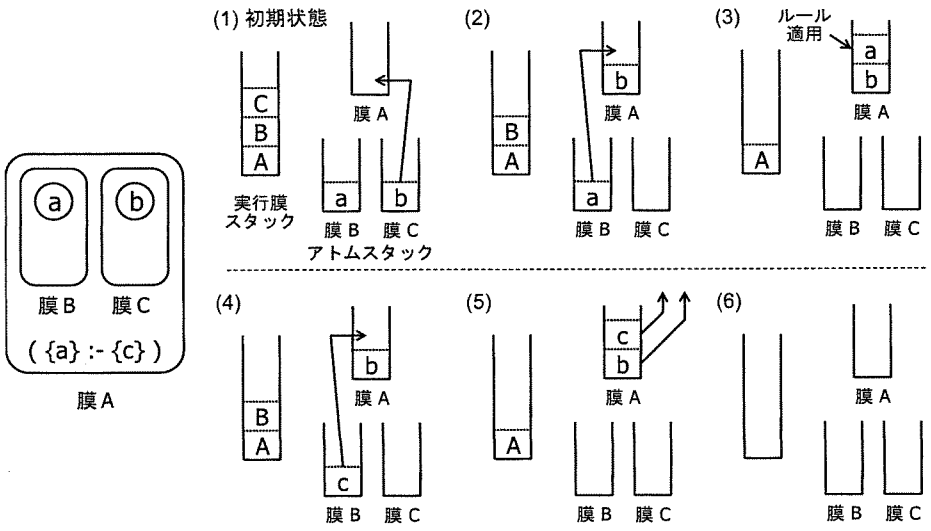


図 3.6　実行膜スタックとアトムスタックの動作

トムスタックが空なので膜 C を実行膜スタックから取り出す.

(2) (1) と同様に a を移動し膜 B を取り出す.

(3) a に膜 A のルール ({a}:-{c}) を適用. ルール右辺の c を膜 B のアトムスタックに追加し, 膜 B を活性化する.

(4) (1) と同様に c を移動して, 膜 B を取り出す.

(5) (1) と同様に b, c を取り出し, 膜 A も取り出すが, A の親膜はないので b, c はどこにも積まない.

(6) 実行膜スタックが空なので終了する. □

膜主導テスト

リスト構成子のようなアトムは個数も適用可能なルールも多いため, アトム主導テストを行うと効率が悪い. そこで, append のように, 関係するルール数も実行中に生成される個数も限定されるアトムのみをアトムスタックに積むことにする. このような「特定のルールに関連付けられている」とみなすことができるアトムを**アクティブアトム**と呼ぶ. アクティブアトムは他言語における関数名や手続き名にほぼ対応するもので, 現処理系では, 英数字や特殊記号からなるアトムを原則としてアクティブアトムと見なしてアトムスタックに積む対象としている. これに対して, 整数, 浮動小数点数, 文字列およびリスト構成子を表すアトムはアトムスタックには積まないこととしている.

これにより, 非決定的バブルソートプログラム

```
L=[$x,$y|L2] :- $x>$y | L=[$y,$x|L2]
```

```
  while (not 本膜が変更) {
    if (not 本膜のアトムスタックが空) {
本膜のアトムスタックからアトムを pop
ルール適用を試みる /* アトム主導テスト */
if (ルール適用できた) {
   ルール右辺のアクティブアトムを所属膜の
   アトムスタックに push，右辺の膜を活性化する
} else {
   pop したアトムを親膜のアトムスタックに push
}
     } else {
本膜の膜主導テストを行う
if (ルール適用できた) {
   ルール右辺のアクティブアトムを所属膜の
   アトムスタックに push，右辺の膜を活性化する
} else {
   本膜を実行膜スタックから pop
}
     }
  }
```

図 3.7　ルール適用処理の流れ

のように，アトム主導テストのみではルールが適用されなくなる場合が発生する．また，

```
{{ }} :- { }
```

のように，アトム主導テストが本質的に適用できないルールもある．そこで，アトムスタックが空になったときには膜の持つすべてのルールの適用を試みる．これを**膜主導テスト**と呼ぶ．膜主導テストが失敗したら，適用できるルールがないとして本膜を実行膜スタックから取り除く．

本節で述べたルールの適用順序をアルゴリズムとして表すと，図 3.7 のようになる．

なお，アトム主導テストと膜主導テストは，実用上や実際の処理順序の観点からは後者が前者のバックアップ手段であるが，完全性をもつ後者を基準と考えて前者を高速化手段と見なすこともできる．この場合アクティブアトムはアトムスタックを用いた高速化を実現するために導入した概念と考えることができる．

3.3.3 複数スレッドによる実行

LMNtal では，ルールは膜によって局所化されている．単一スレッド実行の場合は前節で述べたように実行膜スタックを使って膜単位のスケジューリングを行うが，複数スレッドで非同期実行を行う場合も，上記の理由から膜を実行管理の単位とすることとした．

LMNtal 処理系において実行管理をつかさどる主体を**タスク**と呼ぶ．個々のタスクは，ある膜 (そのタスクの**ルート膜**と呼ぶ) とその子孫膜を管理し，単一スレッドで逐次実行される．ただし，新たに生成する膜にプラグマ (処理系のための付加情報，3.5.2 節) を付加すると別タスクの管理下とすることができ，プラグマを付加した膜 (ルート膜) およびそれより下の階層にある膜は別のスレッドが別のタスクとして実行する．各タスクは膜単位の**ロック**を取得することで，書換え対象のプロセスに排他的にアクセスする．

これらのタスク群は**ランタイム**と呼ばれる LMNtal プログラムの管理主体によって管理される．

3.4 実行時処理系

本節では，第 3.3 節で概説した処理を担当する実行時処理系の詳細を述べる．実行時処理系は大きく分けて，次に挙げる 3 種類の機能を提供している．

- 実行時データ構造の操作機能 (3.4.1 節)
- 単一スレッド実行の制御機能 (3.4.2 節)
- 複数スレッドによる非同期実行の制御機能

このうち非同期実行は，**実行可能プロセス** (書換えができる可能性のあるプロセス) の管理とタスクやロックの管理にまたがり，多くの技術課題をもつ．そこで本節ではタスクとロックの管理について 3.4.3 節で簡単に紹介し，非同期実行機能の全容は第 3.5 節で詳しく論じる．

3.4.1 実行時データ構造

LMNtal のプロセスはアトム，膜，リンク，ルールを四大構成要素とするが，本処理系ではそれぞれ `Atom`, `Membrane`, `Link`, `Rule` のクラスのオブジェクトとして表現している．図 3.8 のように `Membrane` オブジェクトは，`Atom` および `Rule` オブジェクトを多重集合管理用の `AtomSet`, `Ruleset` を用いて管理するとともに，自身が保有する膜 (**子膜**) を表す `Membrane` オブジェクトを Java API の `Set` を用いて管理する．膜は，タスクを表現する `Task` オブジェクトが管理する．全タスクを管理するランタイムは `LMNtalRuntime` オブジェクトとして実行時にただ 1 つ生成される．

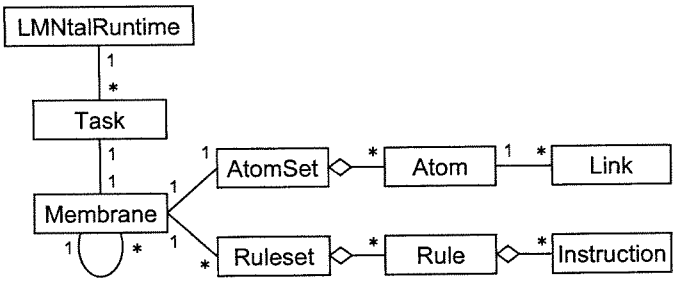


図 3.8　実行時データ構造のクラス図

プロセスのデータ表現

プロセスを構成するアトム，膜，リンク，ルールの表現は，書換え処理の時間計算量を小さくすること，膜間のルール移送が可能なこと，シャッフルモード (ルール適用のランダム化) に対応することなどのさまざまな要求に応じるべく設計した．これらの要求を実現するために各オブジェクトが持つフィールド数は必ずしも少なくないが，メモリ使用量はプロセスの大きさに比例する量に抑えている．

■ **アトム**　Atom オブジェクトは，アトムの各引数に対応する Link オブジェクト以外に，アトムが所属する膜に対応する Membrane オブジェクトへの参照を持つ．所属する膜の情報を持つのは，たとえばリンク先のアトムを参照する際に，そのアトムがどの膜に所属しているかを知る必要があるためである．アトム主導テストにおいても，アトムスタックから取り出した主導するアトムがその膜のトップレベルに所属しているのか子孫膜に所属しているかの情報が必要となる．

■ **リンク**　各リンクは，片方向の参照を持つ Link オブジェクト 2 個によって表現している．1 個の Link オブジェクトが双方向の参照を持つようにすることも検討したが，そうすると，どちらの参照が相手方のアトムにつながるものかを高速に判定するためには何らかの付加情報が必要になるとの結論に達した．リンクをオブジェクトとせずに上記の情報を Atom オブジェクトに埋め込む方法も考えられるが，中間命令列の最適化や分散処理におけるプロセスの送受信を簡潔にするために，オブジェクトとして表現することとした．

■ **膜**　個々の膜を表す Membrane オブジェクトは，膜内のプロセスの管理だけでなく，プロセスの排他制御や実行可能プロセスの管理をつかさどる．

Membrane オブジェクトは，以下の 4 種類のプロセス参照を持っている．

- 自分の**親膜**の Membrane オブジェクト
- 保有する膜 (**子膜**) を束ねる Set オブジェクト
- 保有するアトムを束ねる AtomSet オブジェクト

- 保有するルールを束ねる Ruleset オブジェクトのリスト

また，排他制御のためにロックを，実行終了判定のために stable フラグおよび perpetual フラグをもつが，これらは非同期実行と深く関連するので第 3.5 節で述べる．

親膜への参照は，(i) この膜がルート膜であったとき，ルール適用の終了時に親膜の実行を再開させるため (3.5.4 節)，(ii) 明示的でない自由リンクをもつプロセス文脈にマッチしたプロセスを移動する際のリンクのつなぎかえのため (3.4.1 節)，および (iii) アトム主導テスト (3.3.2 節) で用いられなかったアトムを親膜に知らせるために使っている．

子膜は Set クラスを用いて管理している．計算モデル上は子膜の多重集合であるが，実装上は個々の子膜は別オブジェクトとなるため，集合として扱ってよい．この参照には，Set を継承する HashSet もしくは RandomSet クラスのインスタンスを代入する．HashSet を使うと膜の挿入・削除を O(1) 時間で実行でき，通常はこれを用いる．実行過程をランダマイズするシャッフルモード (3.9.1 節) では RandomSet を使用して，子膜をランダムに選択する．

LMNtal ではルールが膜間で移送可能であるが，これは以下のように実現している．各ルールセットは移動，複製，消去されることはあるものの，その内容は作成時に固定され，動的に分解されることはない．同一膜内のルールセットを他の膜に移動・複製する際には，移動・複製先でルールセットを 1 つに統合せず，同一膜内の複数のルールセットを List で管理している．シャッフルモードにおいては，List の要素をランダムに取得する RandomIterator クラスを用いてルールセットを選択している．

■ ルール 個々のルールを表現する Rule オブジェクトは，中間命令を表現する Instruction オブジェクトのリストを保持する．中間命令列は**ヘッド命令列**，**ガード命令列**，**ボディ命令列**に分けてそれぞれ別のリストで保持する．

各ルールは，実行中に作成されたり構造が変化したりすることはない．ルールの右辺にルールが書けるので実行可能なルールが動的に追加されることはあるが，そのようなルールも静的にコンパイルできる．

ルールは Rule オブジェクトの ArrayList として Ruleset オブジェクトが管理し，通常は格納順序にしたがって適用検査される．ただし，シャッフルモードを用いてランダムに検査することもできる．

アトム多重集合の管理

ルール左辺に対応するグラフ構造を探索するマッチング処理の効率は重要であるが，本処理系では，特定のアトムの探索処理に，アトム名と価数 (arity) の情報を用いている．アトム名と価数の対を**ファンクタ**と呼ぶ．ファンクタは Functor オブジェクトで表現し，同じファンクタをもつアトムが Functor オブジェクトを共有することでメモリ量の削減を図っている．

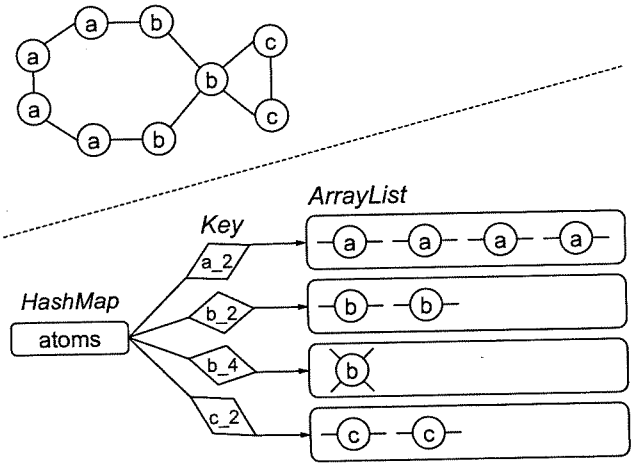


図 3.9　サンプルデータ (上段) および Atom オブジェクトの AtomSet 内での管理方法 (下段)

個々の膜の AtomSet 内では，膜内のアトムを，以下の 2 つの方法のいずれかで保持している．まず，数値アトムおよび文字列アトム以外のアトム (**シンボルアトム**) は，名前の Functor をキーとしてその名前をもつアトムの ArrayList を返す HashMap を用いて管理している (図 3.9)．Atom の挿入・削除時はこの ArrayList を操作する．Atom に ArrayList の index 値を持たせ，削除時には削除した場所に ArrayList の最後尾の要素を挿入することで，指定した Functor をもつ各アトムを O(1) 時間で次々取得できるだけではなく，削除操作も O(1) 時間で実行できるようにしている．

一方，数値や文字列を表すアトム (**データアトム**) は，ほとんどのルールにおいてはシンボルアトムからリンクをたどってアクセスできれば十分で，膜内のデータアトムを高速に検索する必要性は少ない．つまり HashMap に登録することによる計算量上の恩恵よりも，多様なデータアトムを HashMap に登録することによる空間効率上の不利益の方がはるかに大きい．そのためデータアトムはすべて単一の ArrayList にまとめている．

自由リンク管理アトム

非同期に実行されるプロセス間にリンクが存在するとき，個々のプロセスの動作をできるだけ局所化する必要がある．

例 3.4.1　次のプログラムを考える．

```
{(a(X):-c(X)), a(A)}, {b(A), d, (d:-e)}
```

アトム a と b が実行時データ構造においてリンク先を直接参照してしまうと，局所的な書換えである自由リンクのつなぎかえが兄弟膜の実行に影響してしまうため，並行実行できなくなってしまう．□

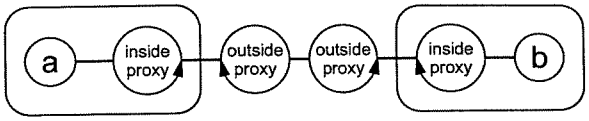


図 3.10　自由リンク管理アトム

この問題を解決するため，膜を貫くリンクに対しては，内部的に**自由リンク管理アトム (proxy)** という特別なアトムを膜の内側と外側に 1 つずつ挿入している (図 3.10)．なお，図中の矢印はアトムの第一引数の位置と引数の並ぶ方向 (右回りまたは左回り) を示す．膜の内側に挿入するものを **inside proxy**，外側に挿入するものを **outside proxy** と呼び，それぞれ特別な `Functor` で表現している．複数の膜を貫く場合は膜ごとに一組ずつ挿入する．自由リンク管理アトムは中間命令列へのコンパイル段階で挿入され，プログラム実行中も，リンクが膜を貫く箇所ごとに自由リンク管理アトムがちょうど一組ずつ挿入した状態を保つ．その具体的な仕組みについては 3.6.4 節で述べる．

3.4.2　単一スレッドによるルール実行の制御

ルール実行の制御において重要なことは，(1) ルールの実行の完全性を保証しつつ，(2) 適用できる可能性がないルールはなるべく実行しないようにすることである．本節では，3.3.2 節で述べた実行方式の実装上の留意点と本処理系における実装について述べる．

ルール実行方式間の優先度

3.3.2 節で述べたように，単一スレッド実行においては，アトムスタックと実行膜スタックの併用で効率とルール実行の完全性の両立を図っている．

実行中の膜 (**本膜**) はアトム主導テストもしくは膜主導テスト (3.3.2 節) のいずれかで実行されるが，アトム主導テストは膜主導テストよりも一般に各テストの成功確率が高いので高速であり，大部分のルールはアトム主導テストによって適用可能である．そのため本処理系では，アトム主導テストに高い優先度を与え，アトムスタックが空であるときのみ膜主導テストを行うようにしている．

アトム主導と膜主導の両方のテストを行うことになった場合，多くのルールに関しては重複して適用検査を行うことになる．ただしこの重複は，アトムスタックが空でない状態から空に変化するときだけしか発生しないため，大きなオーバーヘッドにはならない．

膜間の実行優先度

各タスクの実行膜スタックには，そのタスクが管理する膜のうち，適用可能なルールが存在する可能性があるもの (実行可能プロセス) が積まれるが，その積みかたも効率に影響する．

本処理系では親膜にあるルールよりも子膜にあるルールを優先して実行することとした．LMNtal ではルールが子孫膜を書き換えることを許しているため，ある膜の内容が変化すると，その先祖膜にあるルールの適用が可能になる可能性がある．そのため，子膜側のルール適用を優先的に検査することで，親膜側の適用検査のやり直しを最小限に抑えることができる．

このように多くの場合は子膜を優先した方がルール実行回数が少なくすむが，親膜のルールを定期的に実行した方が効率的になるケースもある．その場合は，定期的に子膜のルール適用を中断させ，親膜ルールの適用後に子膜ルールを再開させるようにプログラムを記述すれば解決できる．上記の考察から，子膜の実行を優先させ，不変条件として，親膜は子膜よりも実行膜スタックの底側に積まれていなければならないことにした．非同期実行下での不変条件の維持については 3.5.4 節で述べる．

コンパイルされたルールの実行

プログラムの実行を始めるには，ソースプログラムに記述されたアトム，リンク，膜，ルールからなる初期プロセスを生成する必要がある．コンパイラは，初期プロセスを作成するための中間命令列をもつ Rule オブジェクトを生成する．

例 3.4.2　ソースプログラムに

```
{a(X)}, {b(X)}.
{a(X)} :- {b(X)}.
```

と記述すると，ルール `{a(X)} :- {b(X)}` の中間命令列をもつ Rule オブジェクトに加えて，

```
() :- {a(X)}, {b(X)}, ({a(X)}:-{b(X)}).
```

(左辺は空プロセス) というルールの中間命令列を持つ Rule オブジェクトを生成する．実行時処理系は，一番最初に後者の中間命令列を実行して，セル `{a(X)}`，`{b(X)}`，およびルール `{a(X)}:-{b(X)}` の中間命令列を生成する．□

このような初期データ構造作成を行うルールは 1 回しか実行しないようになっている．そうでないと右辺のプロセスが無限個生成されてしまうからである．

ルールはルールセットを単位として管理するが，ルールの右辺にルールが出現する場合はその集合が単独のルールセットを形成する．たとえば，上記のソースプログラムをコンパイルすると合計 2 個のルールセットが生成される．

中間命令列形式にコンパイルされたルールを解釈実行する際は，Ruleset のサブクラスである InterpretedRuleset クラスのオブジェクトを作成し，1 回のルール適用を行うメソッドを呼び出す．メソッドの引数には書換え対象の膜を指定する．アトム主導テストの

場合にはそれに加えて主導するアトムも指定する.

中間命令列はヘッド命令列，ガード命令列，ボディ命令列に分かれており，実行はヘッド命令列から始まる．ヘッド命令列ではルール左辺とプロセスとのマッチングを行うが、これにはさらに，アトム主導テスト用の命令列と膜主導テスト用の命令列の 2 種類があり，実行時にいずれかが指定される．マッチングに成功するとガード命令列がガードに記述された制約を満たすかどうかを確認する．制約を満たさない場合はヘッド命令列へのバックトラックが起きる．ガード命令列に成功するとボディ命令列が左辺プロセスの除去と右辺プロセスの生成を行う.

中間命令の詳細や用例については，第 3.6 節にて解説を行う.

3.4.3 タスクとロックの管理

複数スレッドによる非同期実行の制御には，タスク管理とロック管理が必要となる.

タスク管理

3.3.3 節で述べたように，本処理系の実行主体はタスクであり，各タスクは別スレッドで実行される．タスクは**ランタイム** (処理系上では `LMNtalRuntime` クラス) によって管理され，ランタイムはタスクの追加や，全体の実行終了を各タスクに通知する処理を行う.

新たに生成された膜は，別タスクでの実行を指定されない限り，その親膜と同じタスクで実行される．膜階層の最上位で作られた膜は，世界的ルート膜をルート膜とする**ルートタスク**で実行される．世界的ルート膜は実行時データ構造の基準となり，初期データ構造もこの膜を基準に生成する．そのためコンパイラが生成した初期プロセス作成ルールは，ランタイムから世界的ルート膜の参照を受け取って実行される.

ロック管理

あるルート膜とその子孫のルート膜を管理する 2 つのタスクは，同一のプロセスにアクセスすることがあるので，処理系は排他制御を行う必要がある．本処理系では膜をプロセスの排他制御の単位とし，膜内のプロセスにアクセスする際はその膜のロックを取得するようにした．アトム単位の排他制御は，時間および空間オーバーヘッドが大きくなりすぎるためである．具体的なロック取得方法およびロックの種類については 3.5.3 節で述べる．他タスクの膜のロックを取得した際は，ロックを解除する際にも注意深い処理が必要となる．(3.5.4 節).

3.5 非同期実行機能

LMNtal 処理系は，以下の機能の実現を目的として，複数タスクの非同期実行機能を実装している.

- 並列実行 — 共有メモリ型の並列計算機における膜単位の並列処理の実現.
- 他言語インターフェース — インラインコードで記述した Java スレッドから LMNtal におけるプロセスを非同期的に書き換える機能は，特に GUI ライブラリを作成する際に必須である.
- 分散処理系への拡張 — ルール適用と非同期に動作する通信部分を実装すれば分散処理系の基盤が得られる.

非同期実行をサポートするために，本処理系ではロックを用いた排他制御を行っている. 処理系が正しく動作するためには，複数タスクでの実行に起因するデッドロックを起こさず，かつ完全なルール適用を行う必要がある. 本節では，これらの性質を効率良く実現するために採用した方法について説明する.

3.5.1 非同期実行の設計方針

3.4.3 節で論じたように，本処理系では，膜を非同期実行の単位としている. すなわち，異なる膜に所属するルールの適用処理は同時に実行できるが，同一の膜に所属するルールの適用処理を同時に実行することはできない.

非同期実行機能の目的の一つは，LMNtal でタスク並列処理を実現することである. LMNtal では膜に計算の局所化機能があるので，膜をタスクの最小単位として利用することで，タスク並列のために必要な粒度の並列性を，最低限のオーバーヘッドで実現できる.

また，排他制御の単位としても膜を利用することにした (3.4.3 節). つまり，複数のスレッドが同時に同一の膜を操作することはできない. ここで言う膜の操作とは，その膜に所属するアトムの作成と除去，リンクのつなぎかえ，および子膜の作成や除去である. 子膜の操作は含まない. 膜の内容を他の膜に移動する場合，その子孫膜のロックを取得する必要はない.

例 3.5.1 次のプログラムを考える.

```
{                                  %M1
  { a, (a :- calc_something) },  %M2
  doing_something_here_too
},  ({$q, go_up} :- $q)
```

外側の膜を M_1 とし，M_1 の子膜を M_2 とする. M_1 の外側にあるルールによってプロセスの移動を行う場合，M_1 はロックする必要があるが，M_2 はロックしなくてよい. したがって，このプロセス移動ルールと M_2 内部にあるルールとは同時に実行できる. □

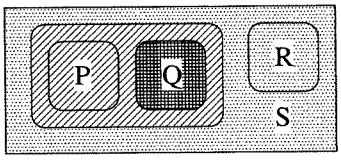


図 3.11　タスク階層

3.5.2　非同期実行に関する概念と記法

ルールの適用処理を行う各タスクは異なるスレッドで実行されるが，これらのスレッドを**ルールスレッド**という．これに対し，プロセスを操作するスレッドのうちルールスレッド以外のものを**非ルールスレッド**と呼ぶ．インラインコードを用いて記述したスレッドがこれに該当する．

プログラマは，複数タスクの利用と各タスクが管理する膜の範囲の指定のために，プログラム中でルート膜 (3.3.3 節) の指定を行うことができる．膜の後に `@"localhost"` というプラグマを記述するとその膜がルート膜になり，ルート膜とその子孫膜のうち他のルート膜とその子孫膜以外のものは，ルート膜の親膜を管理するタスクとは異なるタスクによって管理されるようになる．この記法は分散処理への拡張を想定したものである．分散処理系では，`@` の後にマシン名や IP アドレスを記述することで，その膜を異なるマシン上で実行する．

例 3.5.2　次のプログラムを考える．

```
{ { P },
  { Q }@"localhost"
}@"localhost",
{ R }, S
```

このプログラムには 4 つの膜があり，3 つのタスクを用いて実行される．このプログラムの階層構造をタスク毎に塗り分けて図示すると図 3.11 のようになる．　□

3.5.3　ロックによる排他制御

本処理系は，ロックを用いて膜単位の排他制御を行うが，ロックの導入に起因するデッドロックを防ぐために，ロックに関して次の規則を設けた．

1. 膜のロックを取得するスレッドは，どの膜のロックも取得していないか，またはその親膜のロックを取得していなければならない．
2. ルールスレッドが最初にロックを取得する膜は，自タスクが管理する膜でなければ

ならない.

3. 非ルールスレッドが最初にロックを取得する膜は，ルート膜でなければならない.

1. により，複数のスレッドが互いに他方の保持するロックを取得しようとしてデッドロックすることがなくなる.

この3つの規則から導かれる性質として，ある膜が，その膜を管理するタスクを実行するルールスレッド以外のスレッドによりロックされる場合は，その膜からその膜を管理するタスクのルート膜までの膜は全て同一のスレッドによってロックされる.

なお，非ルールスレッドがこの規則を満たすようにすることは，プログラマの責任である．実際には，ルート膜から指定された膜までの間を親膜から順にロックするメソッドを用意して，プログラマの負担を軽減している.

ロックには他タスクからのアクセスの中断を要求する**強制的なロック**と，他タスクがロックを取得していないときのみロックする**非強制的なロック**の2種類を用意している．LMNtal はルールやプロセスの実行順序を規定していないため，あるプロセスのロックがすでに取得されていても，その間は他のプロセスの書換え作業を進めることができる．そのため通常のルール適用の際は非強制的なロックが利用される.

強制的なロックは，他言語で記述したスレッドから膜にアクセスする際などに用いる．強制的なロックを実現するために，他タスクからのロックリクエストの受付処理を用意している．定期的にロックリクエストを監視し，リクエストがあったときはタスクのスレッドを停止することでリクエスト先の処理を許している．ただしスレッドを停止するタイミングは，一回のルール適用が終了した時点であり，プロセスの書換え途中に処理を受け渡すことはない．ロックを解除する際には他のスレッドを中断させてしまっているので，そのスレッドの実行を再開させる.

3.5.4 タスク内の実行制御方式

本節では，タスクのルール適用の完全性を保証するためのルール適用処理の方法について説明する.

仮の実行膜スタック

タスク T_1 が，別のタスク T_2 が管理する膜 M の内容を変更した場合，M を T_2 の実行膜スタックに積む必要がある．しかし，M の子孫膜がすでに実行膜スタックに積まれている可能性があるため，単純に実行膜スタックの先頭に積むことはできない．そこで，他タスクによって変更された場合は実行膜スタックの底に挿入することにした．ロックに関する規則より，T_2 のルート膜から M までの膜はすべて T_1 がロックしているので，これらをすべて実行膜スタックの底に挿入することで，3.4.2 節の不変条件を満たしつつ M を実行膜スタックに追加することができる.

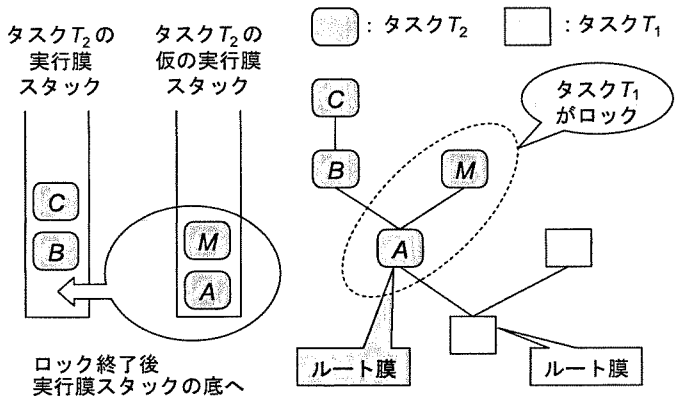


図 3.12　仮の実行膜スタックの操作例

この処理を効率良く行うために，各タスクは**仮の実行膜スタック**を持っている．仮の実行膜スタックの操作例を図 3.12 に示す．他タスク管理の膜の内容を変更した場合には，その膜をまず仮の実行膜スタックに積む．そして，ルート膜のロックを解放するときに，仮の実行膜スタックの中身を実行膜スタックの底に挿入することで，親膜は子膜より底側に積まれるという不変条件を守る．

実行膜スタックに関する不変条件と実行膜間優先度

親膜にあるルールより子膜にあるルールを優先して処理するために，以下の不変条件を設けた．

1. ある膜 M が実行膜スタックに積まれており，M がルート膜でない場合，M の親膜は次のいずれかの状態にある．
   (a) 実行膜スタックの，M よりも底の方に積まれている．
   (b) 仮の実行膜スタックに積まれている．
   (c) 他タスクからロックされている．
2. ある膜 M が仮の実行膜スタックに積まれており，M がルート膜でない場合，M の親膜は仮の実行膜スタックの M よりも底の方に積まれている．

親膜がロックされているか，仮の実行膜スタックに積まれているならば，いずれ実行膜スタックの M よりも底側に積まれることになる．また，膜 M の親膜が M と同じスタックの底側に積まれていれば，M のルールが先に実行されることになる．このことから，親膜 M が上記のいずれかの状態にあれば，親膜にあるルールよりも子膜にあるルールが優先して処理されることになる．

膜の活性化

膜の内容を変更した場合など，ある膜にあるルールが適用できる可能性が生じた場合，3.5.4 節の条件を満たすようにその膜を実行膜スタックに積む．この操作を膜の**活性化**といい，タスクのルール適用の完全性を保証するのが目的である．膜 M の活性化とは，具体的には以下の操作である．

- M が次のいずれかの状態にあるときはすでに活性状態にあるため何もしない．
  - ロックされている．
  - 実行膜スタックに積まれている．
  - 仮の実行膜スタックに積まれている．
- それ以外の場合，子膜優先の不変条件に従い以下の操作を行う．
  - M がルート膜の場合，M を仮の実行膜スタックに積む．
  - M がルート膜でない場合，まず親膜を再帰的に活性化し，次に M を親膜と同じスタックに積む．

例えば，M がルート膜でなく，かつ M の親膜がすでに実行膜スタックに積まれている場合は，M は実行膜スタックに積むことになる．

活性化は，以下のいずれかに該当する場合に，膜 M に対して行う．

1. ルール適用によって本膜以外の膜 M の内容が変化したとき，
2. ルール適用によって新たに膜 M を生成したとき，
3. 非ルールスレッドが膜 M のロックを解放するとき．

膜を活性化する際には，複数の膜が同時に活性化されて実行膜スタックの条件が崩れることを防ぐため，その膜のロックを取得していなければならない．

ルート膜の実行終了後は，そのルート膜の親膜でもルール適用ができる可能性が出てくる．つまり，親膜のタスクの本膜でいったん適用検査が終わったルールも再検査する必要がある．これは単純な活性化処理では解決できないため，強制的なロックを取得して親膜から一つ上位のルート膜までを実行膜スタックに積みなおしている．

3.5.5 実行終了の判定

LMNtal 処理系は，プログラムの実行終了，すなわち適用できるルールがなくなったことを検出する必要がある．そのために，膜は `stable` フラグを持つ．

`stable` フラグがオンの場合，その膜とその子孫膜には適用可能なルールは存在しないことを表す．世界的ルート膜の `stable` フラグがオンになったら，処理系は実行を終了する．

新たな膜が生成されるときは，その膜の `stable` フラグはオフである．本膜に適用できるルールがなかったとき，子膜の `stable` フラグが全てオンならば，本膜の `stable` フラグを

オンにする．また，ある膜の内容 (子孫膜の内容を含む) を変更した場合，その膜の stable フラグをオフにする．

本膜に適用できるルールがなかったとき，子膜に stable フラグがオフのものがあった場合，この本膜は stable フラグがオフのまま実行膜スタックから除去される．しかし，その場合は子孫膜の中に別タスクの実行膜スタックに積まれている膜が存在するので，この本膜はいずれ活性化され，再度ルールの適用検査が行われる．このことから，適用できるルールが存在する状態のまま実行が終了しないことが保証される．

なお，この方法だけでは非ルールスレッドがプロセスを変更する可能性を検出することができない．そこで，膜に perpetual フラグを用意した．非ルールスレッドが変更する可能性がある膜の perpetual フラグをオンにすると，その膜の stable フラグがオンになることはなくなり，非ルールスレッドがプロセスを変更する前に処理系が終了することを防ぐことができる．この機能は，GUI を備えたプログラムが入力待ち状態になってもプログラムの実行が終了しないようにするために用いることができる．

3.6 LMNtal 抽象機械

本 LMNtal 処理系は抽象機械ベースのコンパイラ処理系であり，ソースプログラムは LMNtal 抽象機械の命令列 (中間命令列) にコンパイルされる [8]．中間命令列は，コンパイラと実行時処理系とのインタフェースという重要な役割を果たしている．

中間命令にコンパイルする手法は記号処理言語において広く定着しており，本処理系の中間言語は，Prolog のための Warren Abstract Machine (WAM) [14] の中間言語と類似した命令形式や粒度に設定した．しかし LMNtal は，マッチング処理の非決定性，膜の存在など多くの点で Prolog と異なり，また非同期並行処理の実現を目的の一つとしているため，命令体系は独自に開発した．我々の LMNtal 抽象機械は WAM と異なり，スタックやヒープ上のデータ表現を具体的に定めることは目的とせず，命令体系と制御構造を定めることを主目的としている．

3.6.1 抽象機械の概要

LMNtal 抽象機械の制御構造は，個々のルールの適用のレベルと，個々のルール適用よりも上位のレベルとに分けて考えることができる．コンパイルされた中間命令列を逐次実行するのは個々のルールの適用のレベルであり，本節の目的はこのレベルを詳述することである．LMNtal 抽象機械の上位レベルの制御構造は，単一スレッド実行においては図 3.7 に示した基本アルゴリズムの通りであり，複数スレッド実行においては図 3.7 を実行する複数のタスクが 3.3.3 節，3.4.3 節，3.5 節に述べた方法で協調動作する．

抽象機械の保持情報

LMNtal 抽象機械が，個々の中間命令列の実行のために保持する主な情報は以下の通りである．

- データ関係: 一連の番号で参照される変数からなる変数ベクタ
- 制御関係:
  - プログラムカウンタ
  - 命令の成功・失敗に関する情報
  - 実行中の繰返し命令の状態

番号で参照される変数は，中間命令が参照するアトム，リンク，膜，ファンクタなどを保持する．抽象機械の仮想レジスタと考えてもよい．本抽象機械では，仮想レジスタの個数つまり変数ベクタの大きさは動的に増減する．実行中のルールが属する膜 (本膜) の変数番号が 0 に固定されているため，ルール中で取得するアトム等の変数番号は 1 から順番に定義される．中間命令列は SSA (静的単一代入) 形式で構成され，変数の定義は各変数ごとに 1 回のみである．SSA 形式にした理由は，(i) 命令間の依存関係が明確になり最適化をしやすい，(ii) 実行の流れを解析しやすい，(iii) バックトラックに備えて破壊的代入の履歴を記録する必要がない，など多くの利点があるためである．

命令列の実行制御

抽象機械は，中間命令列からプログラムカウンタが指す命令を読み込み，その種類と引数に応じた処理を行い，成功したらプログラムカウンタの新たな値が指す命令の実行に移る．命令の中には実行が失敗する可能性をもつものもあり，失敗した場合は**バックトラック**が起きてプログラムカウンタが巻き戻されてゆく．

中間命令は大きく**制御命令**と**非制御命令**の 2 つに分けられる．制御命令は中間命令列の実行の準備，制御の移動，終了処理などをつかさどるものであり，それ自体はプロセスに影響を与えない．制御命令以外の命令を非制御命令と呼び，役割はアトムや膜の取得・生成・除去，リンクのつなぎ直し，型検査，四則演算など多岐にわたる．その中にはバックトラック時に実行のやり直しを試み，異なる実行結果を生成して成功する可能性のある命令が存在し，それらを**繰返し命令**と呼ぶ．中間命令列の実行において，ある繰返し命令を最初に実行した際には，繰返しの途中状態を記録するための繰返し子 (iterator) を生成する．

各命令の機能はヘッド命令列，ガード命令列，ボディ命令列のどこに出現するかで分類できる．

ヘッド命令列の命令： アトムやリンクや膜を取得したり，その構造を検査したりする命令．プロセスの書換えは行わず，マッチングのみを目的としている．前述の繰返し命令もヘッド命令列の命令である．条件を満たすデータの取得や構造の検査に失敗

した場合は直前の繰返し命令へバックトラックする.

ガード命令列の命令： ヘッド部にマッチしたプロセスが，ガード部に記述された制約条件を満たすかどうかを検査する．満たさなかった場合は失敗となり，直前の繰返し命令へバックトラックする.

ボディ命令列の命令： ヘッド命令列とガード命令列によってルール適用可能と判断されたプロセスを実際に書き換えるための命令．左辺プロセスの除去と右辺プロセスの生成が主な仕事であり，アトムと膜の除去や生成，リンクのつなぎかえなどを行う．マッチングの成功が確定しているので，ボディ命令列では失敗は起こらない.

中間命令の形式

各中間命令は，命令の種類を表す命令番号 (本論文ではニーモニックで表現) と，命令の引数のリストからなる．命令の引数として現れるものは，変数番号，変数番号のリスト，引数位置，ファンクタ，命令列などである．命令列は，その命令列の開始点を指すラベルで表現することもある．ファンクタはアトム名と価数を下線でつないだ形，たとえば 2 価アトム a の場合は a_2 という形で記述する.

次節以降では，簡単なルールを用いて中間命令列の動作を説明しつつ，代表的な中間命令の紹介を行う．ここで取り上げる以外の代表的な中間命令は付録を参照してほしい.

3.6.2 中間命令列：アトムとリンクのみの例

サイクルを持つ簡単なグラフの書換えルール

```
a(X,Y), b(X,Y) :- a(X,Y), c(Y,X)
```

の中間命令列を図 3.13 をもとに説明する．このルールにはガードがないので，ガード命令列では実質的な作業は行わない.

まず命令列全体の流れについて解説する．01 行目，21 行目，29 行目，32 行目はそれぞれヘッド命令列のアトム主導テスト部と膜主導テスト部，ガード命令列，ボディ命令列の始まりを表す．マッチングをアトム主導テストと膜主導テストのどちらで実行するかは，アトムスタックの状態や最適化に関する実行時オプションなどに基づいて処理系が選択する.

各命令列の先頭では必ず spec 命令が呼ばれる.

---

spec [*formals, locals*]

[制御命令] 仮引数の数 (本命令列に渡されるアトムや膜の数) を *formals*，本命令列で保持すべきアトムや膜の総数を *locals* から受け取って，後者を保持できる大きさの変数ベクタを確保する.

膜主導テストでは本膜が変数 0 に渡されるため，22 行目のように仮引数の個数は 1 とな

```
01 --atommatch:
02    spec          [2, 8]
03    branch        [[
04      spec        [2, 4],
05      func        [1, a_2],
06      testmem     [0, 1],
07      deref       [2, 1, 0, 0],
08      func        [2, b_2],
09      deref       [3, 1, 1, 1],
10      eqatom      [3, 2],
11      jump        [L120, [0], [1, 2], []] ]]
12    branch        [[
13      spec        [2, 4],
14      func        [1, b_2],
15      testmem     [0, 1],
16      deref       [2, 1, 0, 0],
17      func        [2, a_2],
18      deref       [3, 1, 1, 1],
19      eqatom      [3, 2],
20      jump        [L120, [0], [2, 1], []] ]]
21 --memmatch:
22    spec          [1, 4]
23    findatom      [1, 0, a_2]
24    deref         [2, 1, 0, 0]
25    func          [2, b_2]
26    deref         [3, 1, 1, 1]
27    eqatom        [3, 2]
28    jump          [L120, [0], [1, 2], []]
29 --guard:L120:
30    spec          [3, 3]
31    jump          [L103, [0], [1, 2], []]
32 --body:L103:
33    spec          [3, 5]
34    commit        [null, 0]
35    dequeueatom   [1]
36    dequeueatom   [2]
37    removeatom    [1, 0, a_2]
38    removeatom    [2, 0, b_2]
39    newatom       [3, 0, a_2]
40    newatom       [4, 0, c_2]
41    newlink       [3, 0, 4, 1, 0]
42    newlink       [3, 1, 4, 0, 0]
43    enqueueatom   [4]
44    enqueueatom   [3]
45    freeatom      [1]
46    freeatom      [2]
47    proceed       []
```

図 3.13 a(X,Y),b(X,Y) :- a(X,Y),c(Y,X) の中間命令列

る．アトム主導テストでは本膜のほかに主導するアトムが変数 1 に渡されるため，02 行目のように仮引数の個数は 2 となる．ガード命令列，ボディ命令列には以下に述べる jump 命令によりプロセスの構成要素が引き渡されるが，その個数が仮引数の値となる．

ボディ命令列以外の各命令列の最後では，次の命令列の実行に移るために jump 命令が呼ばれる．

---

jump [*instructions, mems, atoms, vars*]

[制御命令] これまでに取得した膜やアトム，およびそれ以外のデータのリストをそれぞれ *mems*, *atoms*, *vars* とし，それらのデータを引き継いで命令列 *instructions* を実行する．データの引継ぎの際，変数番号の再割当てを行う．

変数番号の再割当ては，アトム主導テストと膜主導テストでは同じマッチングを行う場合でも同じアトムに異なる変数番号が割り当てられることがあるので，jump 実行時に分岐元に依存しないように変数番号を振り直すのが目的である．

アトム主導ヘッド命令列

本命令列は，アトム主導テストによりスタックから取り出したアトムが 2 価アトム a か 2 価アトム b のいずれかの場合にのみ実行される．本命令列はそのどちらに主導されている可能性もあるため，branch 命令により場合分けを行う．

---

branch [*instructions*]

[制御命令] 引数の命令列 *instructions* を実行し，成功したらマッチング成功となる．*instructions* の実行に失敗した場合，次の命令があればその実行に移る．なければマッチング失敗となる．

複数の branch を続けて記述することで，命令列の枝分かれを表現できる．

以後は a_2 が選択されたときについて解説を行う．まず 04 行目の spec で局所変数領域の設定を行い，続いて 05 行目の func 命令により主導するアトムが a_2 であることを確認する．そして，testmem 命令によりアトムが属する膜が本膜であるかどうかを検査する．アトムスタックには子膜のアトムも積まれる可能性があるため，主導するアトムが特定の膜に所属していることを確認するために testmem を用いる．

次に deref 命令によりアトム a_2 の最初の引数に接続されたアトムを取得する．

---

deref [*dstatom, srcatom, srcpos, dstpos*]

アトム *srcatom* の第 *srcpos* 引数のリンク先が，あるアトムの第 *dstpos* 引数に接続していたら成功であり，接続先のアトムへの参照を *dstatom* にロードする．第 *dstpos* 引数以外の引数に接続していたら失敗である．

本命令は，すでに取得したアトムに接続しているアトムを取得するために用いる．

次に func で接続先のアトムのファンクタが b_2 であることを確認したあと，2 つめの deref によりアトム a_2 の他方の引数のリンク先アトムを取得し，そのアトムと最初に取得したアトムとが同一かどうかを eqatom 命令を使って検査する．

ここまで成功するとルール左辺のプロセスにマッチングできたことになるため，jump でガード命令列へ移行する．ガード命令列は空なので直ちにボディ命令列に移行する．この 2 回の移行は無駄であるが，各命令列の末尾の jump は最適化器によって除去することができる．

膜主導ヘッド命令列

膜主導テストは主導するアトムを持たないため，まずアトム a_2 か b_2 を探さなくてはならない．この例では findatom 命令により a_2 を最初に探す．

---

findatom [*dstatom, srcmem, funcref*]

[繰返し命令] 膜 *srcmem* の内部のアトムのうち，ファンクタが *funcref* のものを取得して，そのアトムへの参照を *dstatom* にロードする．
膜内に *funcref* をファンクタに持つアトムが複数個ある場合は，後続の命令列が失敗すると本命令までバックトラックが起き，同じファンクタを持つ別のアトムを取得して，後続の命令列を再度実行する．

この例では，変数 0 が指す膜 (本膜) 内でファンクタ a_2 を持つアトムのうちの一つを取得し，アトムへの参照を変数 1 にロードする．その後 deref と func により，アトム a_2 がアトム b_2 と接続しているかどうか検査する．検査に失敗した場合は本命令が別のアトム a_2 への参照を変数 1 にロードし直して，以降のマッチングを再度試みる．変数 0 の膜内にアトム a_2 がなくなるまで繰返し実行してもマッチングに成功しなかった場合，この findatom は失敗となる．

リンクでつながった 2 つアトムのマッチングは，findatom を 2 つ使わず，1 つの findatom で取得したアトムからリンクをたどって相手方アトムを取得し検査することで効率化を図っている．

アトム主導テストと同様，両引数のリンク先アトムが同一であること確認したらボディ命令列に移行する．

ボディ命令列

spec 命令によりガード命令列からアトムや膜を引き継いだあと commit 命令が呼ばれるが，通常のルール実行には直接関係しないためここでは説明を省く (付録参照)．ボディ命令は大きくはルール左辺のプロセスの除去と右辺のプロセスの生成からなるため，最初にプロセス除去関連の命令が呼ばれる．ただしアトム a_2 と b_2 を削除する前に，それぞれ

のアトムに dequeueatom 命令を適用する.

---

dequeueatom [*srcatom*]

アトム *srcatom* が所属膜のアトムスタックに積まれていたら，スタックから削除する.

以後は新たにスタックに積まれない限り，指定されたアトムはアトム主導テストの検査対象から除外される．この例でアトム a_2 から主導された場合，アトム a_2 はすでにスタックに存在しないが，アトム b_2 はスタックに残っていたら除去される.

続いて removeatom 命令により 2 つのアトムを本膜から削除し，newatom 命令によりアトム a_2 と c_2 を本膜の中に生成する．この例では，ルール左辺にアトム a_2 があるにもかかわらず再作成しているが，3.7.1 節の最適化器を用いると再利用が可能になる.

2 つのアトムはまだ単独で存在しているだけなので，newlink 命令で相互接続する.

---

newlink [*a1, p1, a2, p2, mem*]

膜 *mem* にあるアトム *a1* の第 *p1* 引数とアトム *a2* の第 *p2* 引数の間に新しくリンクを張る.

作られたアトムは他のルールが書き換える可能性があるため，enqueueatom でアトムスタックに積む.

この段階でルール左辺のアトムは本膜からは削除されたが，メモリから解放されたわけではない．そのため最後に freeatom 命令でメモリから解放する．現在は Java のガーベジコレクションに頼っているが，LMNtal 処理系が明示的にメモリ管理を行うことも考えられる.

最後に，制御命令 proceed によって命令列の実行を成功終了させる.

3.6.3 中間命令列：膜とガードを含む例

本節では，アトムとリンクのほかに膜とガードを含むルールのコンパイル例を説明する.

```
{a(X), $n[X], $p} :- int($n) | a(X), $n[X], {$p}
```

これは膜の中にアトム a からつながる整数 $n があったとき，それらを膜から出すというルールである．これを中間命令列にしたものが図 3.14 である.

アトム主導ヘッド命令列

この例ではアクティブアトムは a_1 のみなので，branch による分岐先は 1 通りとなる．アトム a_1 であることを func で確かめた後，getmem 命令によりアトムが所属する膜を取得する.

```
01 --atommatch:
02    spec          [2, 5]
03    branch        [[
04      spec        [2, 4],
05      func        [1, a_1],
06      getmem      [2, 1, 0, null],
07      lock        [2],
08      getparent   [3, 2],
09      eqmem       [0, 3],
10      jump        [L102, [0, 2], [1], []] ]]
11 --memmatch:
12    spec          [1, 3]
13    anymem        [1, 0, 0, null]
14    findatom      [2, 1, a_1]
15    jump          [L102, [0, 1], [2], []]
16 --guard:L102:
17    spec          [3, 4]
18    derefatom     [3, 2, 0]
19    isint         [3]
20    norules       [1]
21    jump          [L103, [0, 1], [2, 3], []]
22 --body:L103:
23    spec          [4, 7]
24    commit        [null, 0]
25    dequeueatom   [2]
26    removeatom    [2, 1, a_1]
27    dequeueatom   [3]
28    removeatom    [3, 1]
29    removemem     [1, 0]
30    removeproxies [1]
31    newmem        [4, 0, 0]
32    movecells     [4, 1]
33    copyatom      [5, 0, 3]
34    newatom       [6, 0, a_1]
35    newlink       [6, 0, 5, 0, 0]
36    enqueueatom   [6]
37    freemem       [1]
38    freeatom      [2]
39    freeatom      [3]
40    proceed       []
```

図 3.14 {a(X), $n[X], $p} :- int($n) | a(X), $n[X], {$p} の中間命令列

---

getmem [*dstmem, srcatom, type, name*]

アトム *srcatom* の所属膜への参照を *dstmem* に代入する．この時点では膜のロックは試みない．所属膜が *type* で表せる型でないか，所属膜の名前が *name* でない場合は失敗する*2．

その後に lock 命令により指定した膜のロックを試みる．

---

lock [*srcmem*]

膜 *srcmem* のロックを試みる．後続の命令列が失敗して本命令までバックトラックが起きたときは，ロックした膜をアンロックしてバックトラックを続ける．

アトムの所属膜が判明した後は，その膜がルールで指定した膜，つまり本膜の子膜であることを確かめる必要がある．中間命令列では，アトムの所属膜の親膜が本膜であるか否か検査することでこれを確認する．まず，getparent 命令でその膜の親膜を取得する．続いて eqmem により，代入された膜が (変数 0 が指す) 本膜であることを確認する．

以上の検査が終了するとガード命令列に移行する．

膜主導ヘッド命令列

膜主導命令列によるマッチング検査では，本膜から順に外側から内側へ必要なプロセスを取得していく．まず，anymem 命令により任意の膜を取得する．

---

anymem [*dstmem, srcmem, type, name*]

[繰返し命令] 膜 *srcmem* の子膜のうち，型 *type* を持ち，名前が *name* である膜に対して次々とロックを試みる．そして，ロックが成功した子膜への参照を *dstmem* にロードする．後続の命令列が失敗すると本命令までバックトラックが起き，先に取得した膜をアンロックした後に別の膜を取得して，後続の命令列を再度実行する．

---

anymem 命令は，マッチングに成功するか，膜 *srcmem* の中に該当する子膜がなくなるまで繰返し実行される．続く findatom 命令では，取得した膜の中のアトム a_1 の取得を試みる．

以上の検査が終了するとガード命令列に移行するが，ガード命令列が失敗して再度 13, 14 行目の繰返し命令に制御が戻る可能性もある．例えばアトム a_1 のリンク先が整数でなかった場合，14 行目の findatom により別のアトム a_1 を取得する．また膜内のアトム a_1 全てについて検査が失敗したら，13 行目の anymem で別の膜を新たに取得する．

---

*2 本命令を含むいくつかの命令に現れる膜の型と名前は，現言語仕様にはない拡張機能である．そのためこの例では，*type* をデフォルトの型である 0 とし，*name* を null としている．

ガード命令列

まずアトム a_1 のリンク先アトムを derefatom 命令で取得する．通常のアトムの場合，リンク先のアトムが特定のファンクタをもつことを確認するのに deref と func を用いるが，この例ではリンク先が整数アトムであることだけが確認できればよい．そのため a_1 のリンク先アトムへの参照を derefatom で取得し，続いて isint で取得したアトムが整数であることを確認する．その後，この例のルール左辺の膜はルール文脈をもたないため，norules 命令によって膜内にルールが存在しないことを確認する．

以上のようにガード命令列の多くは失敗する可能性をもち，失敗した場合，アトム主導命令列もしくは膜主導命令列の最後に成功した繰返し命令に制御が戻る．ただしこの例のアトム主導命令列には繰返し命令がないため，ガード命令に失敗した場合はルール適用が失敗する．すべての命令検査に成功するとボディ命令列に移行する．

ボディ命令列

まず左辺のプロセスを削除するため，25〜29 行目のようにアトムをアトムスタックから除き，所属膜からアトムを削除する．膜についても removemem 命令により膜の親膜から削除する．続く removeproxies も左辺のプロセスを削除する命令であるが，これについては 3.6.4 節で説明する．

続いて右辺のプロセスを生成する．まず，newmem 命令で新たに膜を生成する．

---

newmem [*dstmem, srcmem, type*]
　膜 *srcmem* の中に型 *type* を持つ膜を生成し，*dstmem* にその参照をロードする．

続いて movecells 命令により左辺の膜内のプロセスを右辺の膜に移動させる．

---

movecells [*dstmem, srcmem*]
　膜 *srcmem* にある全てのアトムと子膜を膜 *dstmem* に移動する．

---

movecells 命令はアトムと子膜の個数に比例する時間がかかる．しかしこの例のルールでは定数個のアトムを外に出すだけで良く，アトムと子膜の個数に比例する時間をかける必要はない．この問題は，3.7.1 節で紹介する膜の再利用を用いて解決する．

その後 33〜36 行目のように右辺で新たにアトムを生成し，アトムスタックに積みなおす．ただし整数アトムに関しては copyatom 命令を用いる．この例では，右辺で生成するものが整数アトムということしか分からないため，ファンクタを指定して新規に作成することはできない．そのため本命令を用いて左辺のアトムを複製している．

右辺のプロセス生成が終了すると，37〜39 行目でプロセスをメモリから解放する．膜の解放には freemem 命令を使用するが，freeatom 命令と同様，現処理系は Java のガーベジコレクションに頼っている．

最後に proceed 命令が呼ばれ，命令列の実行を成功終了させる．

3.6.4 自由リンク管理アトムの扱い

3.4.1 節で述べた自由リンク管理アトムの中間命令列における扱いを説明する．

まず，膜を貫くリンクのうち，ソースプログラム中に明示されているもの (明示的な自由リンク) については，構文解析の段階で自由リンク管理アトムを挿入することによって，残りの処理は通常のリンクのつなぎかえと同様に扱うことができる．

例 3.6.1 ルール

```
{a(X)} :- {{b(X)}}
```

は，コンパイル時に

```
$out(X1,X), {$in(X1,X2),a(X2)} :-
    $out(X3,X), {$in(X3,X4),$out(X5,X4), {$in(X5,X6),b(X6)}}
```

と展開される． □

一方，リンク束にマッチするリンク群は実行時にしか特定できないため，コンパイル時に他のリンクのつなぎかえと同様に扱うことはできない．特に，下記の例のようにプロセス文脈がルール左辺に複数個出現する場合，各リンク束にマッチしたリンクどうしが相互接続される場合があり，右辺ではこれについて自由リンク管理アトムを取り除かなくてはならない．

例 3.6.2 二つのセルを融合するルール

```
{$p[|*X]},{$q[|*Y]} :- {$p[|*X],$q[|*Y]}
```

を考える．ボディ命令列では，左辺プロセスの除去前と右辺プロセスの構築後に，右辺のプロセス文脈が出現する膜とその先祖膜全てに対して自由リンク管理アトム操作命令 (removeproxies と insertproxies，付録参照) を実行する．これによって，$p[|*X] と $q[|*Y] とを相互接続するリンクに対して自由リンク管理アトムを削除する． □

しかし，局所的な書換え処理だけでは冗長な自由リンク管理アトムが残ることがある．

例 3.6.3

```
a(A), b(B), {{c(A,B), (c(X,Y):-X=Y)}}
```

を実行すると

```
a(A), b(A), {{c(X,Y):-X=Y}}
```

となるが，自由リンク管理アトムを除去しないと，リンク A が膜を合計 4 回通過した形で残る． □

そこで，次の除去ルールをシステムルールセット (3.8.4 節) に追加している．

```
$out(X1,X), $out(Y1,Y), {$in(X1,Z),$in(Y1,Z),$p} :-
   X=Y, {$p}.
$out(X1,Z), $out(Y1,Z), {$in(X1,X),$in(Y1,Y),$p[X,Y|*V]} :-
   {$p[U,U|*V]}.
```

これらのルールは LMNtal の操作的意味論の (R4) と (R5) をそれぞれ実現するものである．

3.7 最適化

LMNtal コンパイラは中間命令列を生成した後，それを最適化器に通すことで実行効率の改善を試みる．本処理系は操作的意味論に沿った実現に重きを置いているため，最適化器も言語仕様上の簡約動作を再現できる範囲内で実行効率の改善を図る．本節では最適化機能のうち，アトムと膜の再利用，命令列の並び替え，命令列のグループ化を紹介する．

3.7.1 アトムと膜の再利用

ルール適用では左辺にマッチしたプロセスを右辺のプロセスに書き換えるが，`a, b :- a, c` のように左辺と右辺に共通部分 (この場合アトム `a`) を持つルールで，共通部分を除去して再び同じプロセスを生成するのは無駄である．そこで最適化器はヘッド命令列とボディ命令列を比較して，共通するアトムやそれに関連するリンクを再利用するように命令列を書き換える．これによって時間効率も空間効率も改善できる．

膜の再利用に関しても同様に，ヘッド部で取得済みの膜をボディ部で再利用可能かどうかを判断する．

3.7.2 命令列の並び替え

現在の処理系では，命令列の実行中に失敗が起こった際，直前に実行した繰返し命令 (`findatom`, `anymem`) にバックトラックする．そのため，ルール中に 2 つの繰返し命令があり，各繰返し命令にマッチする可能性のあるプロセスが n 個ある場合，最悪の場合 n^2 回の繰返しが起きる．

例 3.7.1 ルール `a(X), b :- X>5 | ok` の中間命令列のうち，膜主導テストのヘッド部とガード部を併合したものは図 3.15 のようになる (ボディ部への `jump` 命令はマッチングには関係ないので省略する)．

この命令列では，アトム `a_1`, `b_0` を取得した後，`allocatom` で整数アトム `5` を生成し，`igt` で `a_1` の第 1 引数が `5` より大きい整数アトムであることを確認する (各命令の詳細は付録参照)．`5` 以下の整数アトムを取得すると `igt` が失敗して，直前の繰返し命令，すな

```
spec      [1, 5]
findatom  [1, 0, a_1]
findatom  [2, 0, b_0]
allocatom [3, 5_1]
derefatom [4, 1, 0]
isint     [4]
igt       [4, 3]
```

図 3.15 a(X), b :- X>5 | ok の中間命令列

```
spec      [1, 5]
findatom  [1, 0, a_1]
derefatom [4, 1, 0]
isint     [4]
allocatom [3, 5_1]
igt       [4, 3]
findatom  [2, 0, b_0]
```

図 3.16 a(X), b :- X>5 | ok の並び替え後の命令列

わち b_0 を取得する findatom にバックトラックする．しかし，別の b_0 の取得に成功しても igt は再び失敗する．これは膜内に b_0 が存在する限り続き，b_0 がなくなったらようやく a_1 を取得する findatom にバックトラックできる．b_0 の取得だけでなく，allocatom による定数アトム 5_1 の生成も毎回無駄に行われる．

これらの無駄は，ヘッド部の検査の完了後にガード部の検査を行うために起こる．この例では，a_1 を取得したらすぐに derefatom でその接続先を取得し，整数であること (isint) と 5 より大きいこと (igt) を確認したあと，b_0 を取得すべきである． □

最適化器はこの問題を解決するため，ある変数を使用する命令があるとき，可能な限りその命令を命令列の後方へ動かす並び替えを行う．命令列は SSA 形式であるため，変数を定義している命令は，使用している命令に対して一意に定まる．よって，使用している命令は，対応する定義命令との順序関係を崩さない限り，命令列の後方へ移動しても実行時の動作に支障をきたすことはない．これに基づいて最適化器に実装した並び替え機能を使って図 3.15 の命令列を並び替えると図 3.16 のようになる．

3.7.3 命令列のグループ化

図 3.16 の命令列は，isint と igt での失敗時には無駄なバックトラックを行わなくなった．しかしこの命令列では，b_0 を取得する findatom で失敗したとき，a_1 を取得する findatom にバックトラックする．b_0 が膜内にないときは，a_1 を取得し直しても無意味

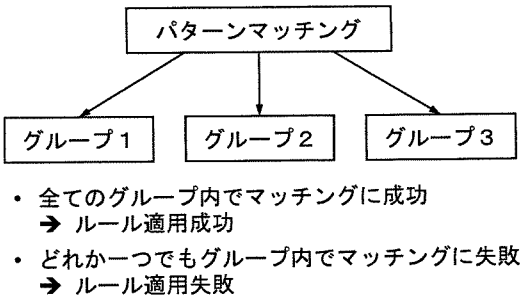


図 3.17　マッチングのグループ分け

であるにもかかわらず，この命令列では膜内のすべての a_1 を取得するまで失敗を確定できない．

この問題は，リンクやガードによる依存関係を持たない 2 つのアトムを，まとめて一連のパターンマッチングで取得しようとしていることにより生じている．そこで，互いに依存しない複数のプロセスとのマッチングを行うルールにおいて，マッチング作業を独立したプロセスごとに区切ることとした (図 3.17)．

マッチングのグループ分けは命令列のグループ化によって行う．各命令の変数は単一代入であるため，依存関係は容易にわかる．つまり，同じ変数を引数に持つ命令，およびその変数を定義している命令は同じグループとする．命令列全体に対し依存関係を調べることで命令列のグループ化を行う．グループ化を支援する group 命令を新たに定義した．

`group [`*instructions*`]`

[制御命令] 命令列 *instructions* を実行し，成功なら次の命令へ行き，失敗ならルール適用失敗とする．マッチングのグループ化が可能なときは，命令列は group 命令がいくつか並んだ形になる．すべてのグループでマッチングに成功したらルール適用は成功となり，どれか 1 つでも失敗するとルール適用は失敗となる．

ルール `a(X), b :- X>5 | ok` の命令列に，並び替えと合わせてグループ化の最適化も施した命令列は図 3.18 のようになる．group 命令の引数命令列の最後には proceed 命令が挿入されており，この命令はそれ以前にある繰返し命令へのバックトラックを終了させる機能も持つ．これにより，b_0 と a_1 の取得がそれぞれ独立した処理として扱われ，a_1 の取得に失敗したとき，b_0 が残っていてもすぐにルール適用の失敗を検知できる．

3.7.4　トランスレータ

中間言語に基づく言語実装では，中間言語コードから下位言語コードへのコンパイル時変換は，広い意味での最適化と考えることができる．

本処理系のトランスレータは，中間命令列から次のような Java ソースコードへ変換す

```
spec   [1, 5]
group  [[
  spec      [1, 5],
  findatom  [2, 0, b_0],
  proceed   [] ]],
group  [[
  spec      [1, 5],
  findatom  [1, 0, a_1],
  derefatom [4, 1, 0],
  isint     [4],
  allocatom [3, 5_1],
  igt       [4, 3],
  proceed   [] ]]
```

図 3.18 `a(X), b :- X>5 | ok` のグループ化後の命令列

る．まず，ルールセット 1 つを 1 つのクラスとし，命令列 1 つを 1 つのメソッドとして表現する．`jump` 命令や `branch` 命令のように引数に命令列をもつ制御命令による制御構造は，これで自然に表現できる．複数回成功する繰返し命令は，抽象機械における実装と同様，繰返し子 (iterator) を用いたループを用いて表現することにした．

命令列に対応するメソッドは，成功か失敗かを表す `boolean` 値を返す．命令列が成功した場合は，`true` を返して命令列の実行全体を終了する．途中で失敗した場合は `false` を返すが，命令列の呼出し元ではこの返り値を判定してバックトラックが起きた場合の処理を行う．例えば，複数回成功する命令の場合は次の値を取得して当該命令列を実行し直し，ロックを取得する命令の場合は取得したロックを解放する．

3.8 拡張言語機能

本節では，LMNtal の基本言語モデルには含まれないが実際的なプログラミングには必要ないくつかの拡張言語機能について，その簡単な解説と実装方法を説明する．各機能の詳しい仕様や使い方は文献 [5] を参照してほしい．

3.8.1 モジュールシステム

実際的アプリケーションを記述するためには，モジュールの宣言・読み込み機能が不可欠である．

モジュール m を定義するには，(i) モジュール m に含めたいルール群および (ii) モジュール名を表す `module(`m`)` というプロセスからなるセルを定義し，それを m`.lmn` ファイルに保存してライブラリパス (デフォルトでは `./lmntal_lib` だが実行時にコマンドライン引数 `-I` で追加可能) が通ったディレクトリに置けばよい．

モジュール m は，$m.$ で始まる名前を持つアトムに言及するとその膜の中に読み込まれる．モジュールをロードする膜の中に m.use と書くのが LMNtal プログラミングの標準スタイルとなっている．

モジュールからのルール群の読み込みは，ルール右辺においてモジュール読み込みアトム (bool.use 等) が記述された膜に対し loadmodule 命令を発行することで実現している．この命令はモジュール内のルール群をその膜に読み込む．

モジュールは通常は Java のバイトコードとして読み込まれるため，あらかじめトランスレータによってコンパイルしておく必要がある．モジュールのコンパイルには --library オプションを用いる．ただし解釈実行時には --use-source-library オプションを指定することによりコンパイルせずに用いることもできるようにした．

入出力，ソケット通信，GUI 等の機能を提供するモジュールが，標準ライブラリとして処理系にあらかじめ用意されている．

3.8.2 他言語インターフェース

LMNtal の他言語インタフェースは，アトム名の中に Java のインラインコードを書く機能を提供することで実現している．インラインコードを含むアトムをインラインアトムと呼ぶ．

インラインコードは，当該インラインアトムを生成するルールの中間命令列の実行の最終段階で実行される．この仕組みにより，入出力，ソケット通信，コマンドライン引数へのアクセスなど OS とのやりとりを担うライブラリが実現されている．また，インラインコードは誰でも書くことができるので LMNtal をグルー言語として使うこともできる．

インラインコードの生成・コンパイル

LMNtal コードから中間命令列へのコンパイル中にインラインアトムが現れたら逐一それを記録し，ソースファイルごとにまとめる．それを元に Java ソースファイルを生成し，LMNtal ソースファイルよりもクラスファイルの日付が古かったらコンパイルプロセスを起動する．

生成される Java コードは図 3.19 の形式をしている．インライン宣言アトムはクラス宣言のためのアトムなので，中身はそのまま宣言領域に展開され，インライン実行アトムは run メソッドの中に展開される．実行時にインライン実行アトムが生成されると，そのアトムと対応するインラインコード番号を引数として run メソッドが呼ばれ，その番号で識別されるインラインコードが実行される．

また，コンパイラはインラインコードを実行するための中間命令 inline を出力する．

例 3.8.1 命令

```
inline  [5, 'lmntal_lib/io.lmn', 7]
```

```
import runtime.*;
/* ここにインライン宣言コードが展開される */
public class SomeInlineCode implements InlineCode {
  public void run(Atom me, int codeID) {
    AbstractMembrane mem = me.getMem(); // インラインアトムが所属する膜を取得
    switch(codeID) {
    case <N>: {
      /* コード ID <N> に対応するインラインコード */
      break; }
    }
  }
}
```

図 3.19　インライン機能使用時に生成される Java コードの形式

は，変数 5 が指すインラインアトムに対応する lmntal_lib/io.lmn 内の 7 番目のインラインコードを実行する．□

インラインコードの実行

処理系はコンパイル時に記録しておいたインラインアトムの出現情報を元にしてクラスファイルを動的に読み込み，インラインアトムに関連づける．

インライン実行アトムを生成するルール右辺の実行の最後で，前述の run メソッドが呼ばれてインラインコードが実行される．引数として自分自身を表す Atom me が渡される．me を通じて所属膜への参照も取得できるので，自分自身やそれにつながるアトムをインラインコードから自由に操作することができる．

3.8.3　アトム集団機能

アトム集団機能は現処理系には実装されていない．そのため他言語インタフェース機能を用い，代替機能を nlmem (nonlinear membrane) ライブラリとして提供している．本ライブラリを用いることで，アトム集団が提供する，自由リンクを持つプロセス文脈の複製，廃棄が可能になる．これらの機能は，自由リンク管理アトムの Java オブジェクトをインラインコード内で直接操作することで実現している．

3.8.4　システムルールセット

システムルールセットは，処理系によってすべての膜内に配置される組込みルール群である．

算術演算および自由リンク管理アトムの簡約規則があらかじめ組み込まれており，ユーザが新たなルールをシステムルールセットに追加する機能も用意されている．処理系は，これらのルール群をあらかじめコンパイルして保持しておき，各膜に所属するルールセットの適用検査時に，膜主導テストの一環として適用検査を行う．

算術演算は，ガードがもつ四則演算機能を用いたルールとして提供されている．

例 3.8.2　整数の加算ルールは

```
H='+'($a,$b) :- $c=$a+$b | H=$c.
```

と定義されている．□

これによって，算術式の形を持つグラフ構造は自律的に簡約されてゆく．ただし現処理系はシステムルールセットを膜主導テストで実行するため，効率が必要な場合はガードの四則演算機能を使用すべきである．算術演算ルール適用の最適化は今後の課題である．

3.9　プログラム実行環境

本節では，プログラムの挙動を調べるために本処理系が提供しているランダマイズ機能，トレース機能，プロファイリング機能について概要を述べる．これ以外の重要な機能として実行状態可視化機能があるが，独立性の高い技術課題であるため別論文に譲る．

3.9.1　ランダマイズ機能

LMNtal の言語仕様は，書換え可能なプロセスや適用可能なルールが複数存在する場合の適用順序を規定しておらず，処理系が自由な順序で適用することができる．本処理系では，標準の実行モードでは，LMNtal のソースコード中の順序に基づいて選択するプロセスやルール適用の順序が決まるため，非決定的なプログラムからも毎回同一の結果が得られる．

しかし，非決定的プログラムから毎回異なる計算結果を得たいこともあろう．また，非決定的プログラムが複数の計算結果をもつかどうかは一般に検証によって調べる必要があるが，それに先だって計算結果が複数通りあるかどうかをシミュレーションによって調べる機能は重要である．そこで本処理系では，膜内のルール選択や，ルールにマッチするアトムや膜の選択をランダムにするシャッフルモードを用意している．

シャッフルモードを指定する実行時オプション `-s` にはいくつかのレベルがあり，それぞれ

`-s1`　膜主導テストのみを行う
`-s2`　アトムや膜の選択をランダムにする
`-s3`　さらに膜内のルールの選択もランダムにする

という機能をもつ.

-s2, -s3 における選択のランダム性は，RandomIterator を用いて実現している．なお，-s3 におけるルール単位のランダム性を実現するためには，ルール 1 つ 1 つを別個のルールセットにコンパイルする必要がある．実行時スケジューリングの最小単位がルールセットであるためである．このため，-s3 オプションの指定はコンパイル時にも必要である.

3.9.2 トレース機能

実行時オプション -t を指定することで，トレースモードでの実行が可能となる．トレースモードでは 1 回のルール適用ごとに，実行状態と，どのルールが適用されたかを出力する．この機能は制御命令 commit (付録参照) により実現されている.

3.9.3 プロファイリング機能

LMNtal は書換え言語であるため，実行のボトルネックとなっている部分を，プログラムに時間計測コードを挿入して探し出すことは難しい．そこでプロファイリング機能を提供することにした．実行時オプションとしてプロファイルの詳細度を指定することができ，詳細度を上げることにより，ルール毎，テスト実行方式毎，スレッド毎にプロファイルを取ることができる.

プロファイリングは，処理系内部の各処理でタイムスタンプを作成し，ルールオブジェクト内に実行時間や実行回数を保存することで実現している．詳細度の設定値が低いときは余計なプロファイルを取らないように工夫している．トランスレータ (3.7.4 節) 使用時は，オプションごとに計測処理の Java プログラムを出力することにより対応している.

3.10 関連研究

LMNtal は，階層グラフ書換えモデルのプログラミング言語への展開の初めての試みである．したがって実装技術の開発も，記号の表現や中間コードへのコンパイルなどの記号処理言語一般にあてはまる概念や技術を別にすれば，プロジェクト当初から独自に研究開発を進めてきた.

LMNtal にある程度近い言語として Constraint Handling Rules (CHR) [3] がある．実際，LMNtal と CHR は小さな共通部分集合をもち，それは膜をもたない LMNtal (Flat LMNtal) および関数記号と propagation rule をもたない CHR にほぼ相当する.

ただし，CHR は制約プログラミング言語として発展し，LMNtal はグラフ書換え言語として発展してきたため，プログラム例にはかなりの隔たりがある．さらに CHR の変数は Prolog の論理変数と同様「値に具体化する」という概念をもつのに対し，LMNtal の変数はアトム間の相互接続が目的であって，値に具体化するという概念をもたない．このことから，CHR の実装技術が LMNtal の実装技術にそのまま活かせるわけではない．LMNtal

の実装技術の開発で鍵となったのは階層化の扱い，特に異なる階層に属するルールの非同期実行や，階層をまたがるグラフ構造の操作であった．これに対して CHR の実装で重要なのはルールマッチングの最適化 [4] である．この技術の LMNtal への導入可能性は，将来の研究課題である．

LMNtal の祖先である並行論理型言語および並行制約言語 [11] も関連言語に挙げられる．しかし LMNtal は (1) 膜による階層構造をもつこと，(2) ルール左辺に多重集合が指定できること，(3) 論理変数の具体化の概念をもたないこと，などの理由から，具体的な実装方法はほぼゼロから検討しなければならなかった．だが，静的型体系もしくは構文的制限を導入して論理変数の用法を限定した Moded Flat GHC[11] や Janus[10] のプログラムの多くは，LMNtal プログラムとしてほぼそのまま実行可能である．

他の重要な関連言語としては Interaction Nets [7] がある．実際，Interaction Nets で書かれた書換え規則は膜を持たない LMNtal プログラムとして実行可能であり，その意味で本 LMNtal 処理系は Interaction Nets のスーパーセットに実用機能を付加して実装したものとも言える．

3.11 まとめと今後の課題

本章では，階層グラフ書換えに基づく並行言語モデル LMNtal の設計と並行して開発改良を進めてきた LMNtal 処理系について詳細な紹介を行った．

記号処理言語処理系の設計における鍵は実行時処理系と中間命令列の仕様であるが，LMNtal は並行性と階層性をもつのでその設計と実装は自明ではなく，本章で紹介した実装方式に至るまでに何回もの見直しが必要となった．しかし結果として，階層グラフ書換えや安全な非同期実行の実装方式が確立し，並列分散実装への展開のベースを構築することができるとともに，非同期処理を必要とする他言語インタフェースの活用が進むこととなった．

5 万行規模の処理系の共同開発は，定期的な会合に加えて，CVS によるコードと履歴の共有，Eclipse 環境の利用，および Wiki を活用した技術的コミュニケーションに支えられてきた．これらのツールのおかげで，技術的な難しさに直面して打開策の模索に時間を要した期間があった以外は，全体として円滑に共同作業を進めることができた．

処理系開発においては，記号処理言語の既存実装技術から容易に発想できる技法の適用はあえて後回しにして，LMNtal 特有の実装上の問題の同定と解決に努めてきた．現在得られている性能は高性能計算用途以外では実用の域に達しているが，静的型体系を導入して処理系と連携させることは重要な研究課題である．また非同期処理を外すことで単純高速化した処理系を，本処理系とは別に設計実装中であり，本処理系のバックエンドエンジンとしても活用する予定である．

参考文献

[1] Drewes, F., Hoffmann, B. and Plump, D. : Hierarchical Graph Transformation, *J. Comput. Syst. Sci.*, Vol. 64, No. 2 (2002), pp. 249–283.

[2] Engels, G. and Schürr, A. : Encapsulated Hierarchical Graphs, Graph Types, and Meta Types, *Electronic Notes in Theor. Comput. Sci.*, Vol. 1 (1995), pp. 75–84.

[3] Frühwirth, T. : Theory and Practice of Constraint Handling Rules, *J. Logic Programming*, Vol. 37, No. 1–3 (1998), pp. 95–138.

[4] Holzbaur, C., Garcia de la Banda, M., Stuckey, P. J. and Duck, G. J. : Optimizing Compilation of Constraint Handling Rules, *Theory and Practice of Logic Programming*, Vol. 5, No. 4–5 (2005), pp. 503–53.

[5] 乾敦行，工藤晋太郎，原耕司，水野謙，加藤紀夫，上田和紀：階層グラフ書換えモデルに基づく統合プログラミング言語 LMNtal，コンピュータソフトウェア，Vol. 25, No. 1 (2008)，pp. 124–150.

[6] 工藤晋太郎，加藤紀夫，上田和紀：LMNtal 処理系におけるグラフ構造の操作機能の設計と実装，情報科学技術レターズ，2005，pp. 9–12.

[7] Lafont, Y. : Interaction Nets, in *Proc. POPL'90*, ACM, 1990, pp. 95–108.

[8] 水野謙, 永田貴彦, 加藤紀夫, 上田和紀：LMNtal ルールコンパイラにおける内部命令の設計，情報処理学会第 66 回全国大会論文集，2005，pp. 203–204.

[9] 水野謙，加藤紀夫，原耕司，上田和紀：階層グラフ書換え言語 LMNtal 処理系における非同期実行の実現，日本ソフトウェア科学会第 22 回大会講演論文集，2005，pp. 213-221.

[10] Saraswat, V. A., Kahn, K. and Levy, J. : Janus: A Step Towards Distributed Constraint Programming, in *Proc. 1990 North American Conf. on Logic Programming*, MIT Press, 1990, pp. 431–446.

[11] Ueda, K. : Experiences with Strong Moding in Concurrent Logic/Constraint Programming, in *Proc. Int. Workshop on Parallel Symbolic Languages and Systems (PSLS'95)*, LNCS 1068, Springer-Verlag, 1996, pp. 134–153.

[12] 上田和紀，加藤紀夫：言語モデル LMNtal，コンピュータソフトウェア，Vol. 21, No. 2 (2004)，pp. 44–60.

[13] Ueda, K. and Kato, N. : LMNtal: A Language Model with Links and Membranes, in *Proc. Fifth Int. Workshop on Membrane Computing (WMC 2004)*, LNCS 3365, Springer, 2005, pp. 110–125.

[14] Warren, D. H. D. : An Abstract Prolog Instruction Set, Technical Note 309, SRI International, Menlo Park, CA, 1983.

[15] 矢島伸吾，永田貴彦，加藤紀夫，上田和紀：LMNtal プロトタイプ処理系の設計と実

装，日本ソフトウェア科学会第 20 回記念大会論文集，2003，pp. 21–25.

付録 A

代表的な中間命令

本付録では，第 3 章の本文中で詳説しなかった命令の中で重要なものを紹介する．各項目の最初の段落が機能および使用上の条件に関する説明で，第 2 段落以降は補足説明である．

A.1 制御命令

`commit [rulename, lineno]`

本命令はボディ命令列で spec の次に呼ばれ，トレースおよびデバッグモードでの実行時，実行中のルールが所属するルールセットの ID とルール名 *rulename* (ルール名が指定されていない場合は `null` を指定) を出力する．*lineno* は，デバッグモードにおいて当該ルールがブレークポイントに設定されているかどうかの検査に用いる．

本命令の目的は二つある．一つは，ルールの適用成功と一対一に対応する場所に置いてトレーサやデバッガの呼出しを行うことで，もう一つは，ヘッドおよびガード命令列とボディ命令列の境目に置いてコンパイラが中間命令列最適化の際の指標として利用することである．

A.2 ヘッド命令列の命令

`func [srcatom, funcref]`

アトム *srcatom* がファンクタ *funcref* を持つかどうかを検査する．持っていなければ本命令は失敗である．

`testmem [srcmem, srcatom]`

アトム *srcatom* が膜 *srcmem* に属しているかどうかを検査する．属していなけれ

ば失敗である.

`eqatom [atom1, atom2]`

atom1 と *atom2* が同じアトムを参照しているかどうかを検査する.

`neqatom [atom1, atom2]`

atom1 と *atom2* が異なるアトムを参照しているかどうかを検査する.

たとえば `a`, `a :- b` のように, ヘッドに同じファンクタを持つアトムが 2 個以上あるルールでは, 1 個目の `a` と 2 個目の `a` は別のアトムにマッチしなければならない. このため, 左辺に同名のアトムが複数個あるときは, 1 度取得したアトムを他の命令が再度取得しないようにする.

`lockmem [dstmem, freelinkatom, name]`

Inside proxy である *freelinkatom* が所属する名前 *name* の膜に対してロックの取得を試みる. ロックに成功した膜は *dstmem* にロードする.

膜を貫くリンクを外側からたどって到達した膜への参照を取得するために用いる命令である. たとえば, ルール

```
a(X), {b(X)} :- c(X), {d(X)}
```

では, まず `findatom` 命令で `a_1` を取得し, 接続先の `b_1` を取得する前に, `lockmem` 命令によってその所属膜を取得する. `anymem` 命令で膜を先に取得するのではなく, アトムを取得してからリンクをたどって膜を取得するのは, 一般にこの方が効率が良いためである.

A.3 ガード命令列の命令

`allocatom [dstatom, funcref]`

ファンクタ *funcref* を持つ所属膜を持たないアトムを生成し, その参照を *dstatom* にロードする.

ガード命令列において, 制約条件の検査のため一時的に必要になる定数アトム, たとえばルール `a($x) :- $x>0 | a($x-1)` のガードに出現する定数アトム `0` を生成するために用いる.

`isunary [atom]`

アトム *atom* が 1 価アトムであるかどうかを検査する．同様の命令に `isint`, `isfloat`, `isstring` があり，それぞれ *atom* が整数アトム，浮動小数点アトム，文字列アトムであることを確認する．

`ilt [intatom1, intatom2]`

整数アトム *intatom1* と 整数アトム *intatom2* が関係 $intatom1 < intatom2$ を満たすかどうかを検査する．引数が整数アトムでなければならないため，`isint` 命令の後で発行される．

同様の命令に `ile`, `igt`, `ige` などがある．浮動小数点型にも `flt`, `fle`, `fgt`, `fge` などがある．

`iadd [dstintatom, intatom1, intatom2]`

整数アトム用の加算命令．$intatom1 + intatom2$ の計算結果の整数アトムを生成し，*dstintatom* にロードする．

同様の命令に `isub`, `imul`, `idiv`, `imod` などがある．`idiv`, `imod` は 0 で除算を行うと失敗が起こる．浮動小数点型の四則演算命令 `fadd`, `fsub`, `fmul`, `fdiv` なども用意されている．

`natoms [srcmem, count]`

膜 *srcmem* にあるアトムの個数が *count* 個であるかどうかを検査する．

プロセス文脈を含まない膜とのマッチングを試みる際に使用する．たとえばルール `{a,b} :- ok` にはプロセス文脈が含まれないので，膜内のアトムの数は 2 個でなければならない．従って，`findatom` 命令で `a`, `b` を取得した後，`natoms` 命令を実行することでその膜に他のアトムがないことを確認する．

`nmems [srcmem, count]`

膜 *srcmem* にある子膜の個数が *count* 個であるかどうかを検査する．`natoms` 命令と使い方は同じで，こちらは子膜の数に制約をかける．

A.4 ボディ命令列の命令

`removeatom [srcatom, srcmem, funcref]`

膜 *srcmem* 内のファンクタ *funcref* を持つアトム *srcatom* を膜内から除去する．

`newatom [dstatom, srcmem, funcref]`

膜 *srcmem* にファンクタ *funcref* を持つアトムを生成し，生成したアトムへの参照を *dstatom* にロードする．

`enqueueatom [srcatom]`

アトム *srcatom* を所属膜のアトムスタックに積む．

`getparent [dstmem, srcmem]`

膜 *srcmem* の親膜への参照を *dstmem* にロードする． 親膜がない場合は失敗する．

`derefatom [dstatom, srcatom, pos]`

アトム *srcatom* の第 *pos* 引数のリンク先のアトムへの参照を *dstatom* にロードする．

`copyatom [dstatom, mem, srcatom]`

アトム *srcatom* と同じ名前のアトムを膜 *mem* に生成し，*dstatom* に代入して返す．

`relink [a1, p1, a2, p2, mem]`

膜 *mem* にあるアトム *a1* の第 *p1* 引数と，アトム *a2* の第 *p2* 引数のリンク先の引数とをリンクで接続する．

ルール左辺でマッチしたアトムの接続先を右辺で別のアトムにつなぎかえるのに用いる．たとえばルール `a(X,Y) :- b(Y,X)` では，`a` の第 1 引数のリンクを右辺で `b` の第 2 引数につなぎかえるが，このつなぎかえに `relink` 命令を用いる．

`removeproxies [srcmem]`

本命令は `removemem` の直後に同じ膜に対して呼ばれ，*srcmem* が持つ自由リンク管理アトムをスターアトム (以下`$star`) に置き換える．ただし，削除前の *srcmem* の階層を通過して，その子膜の内部と *srcmem* の外側とを接続しているだけの自由リンク管理アトムは自動削除する．

本命令はプロセス文脈を移動させて膜の階層構造を変更する準備である．通常，膜間の自由リンクは`$in` と`$out` により階層関係を表現しているが，`$star` に置き

換えることで，上下どちらの階層に向かうかの情報をいったん消去している．

`insertproxies [parentmem, childmem]`

本命令は `newmem` が全て終わった後で呼ばれ，互いに親子関係にある膜である *parentmem* と *childmem* に対し，`$star` を適切に変化させる．*childmem* にある各`$star` に着目し，`$star` の第 1 引数の接続関係を以下のように直す．

- `$star` のリンク先も *childmem* にある`$star` であれば，両`$star` の他方の引数どうしを短絡させる．
- `$star` のリンク先が *parentmem* にある自由リンク管理アトムであれば，それぞれ`$in`，`$out` に変更する．
- `$star` のリンク先がそれ以外の膜へのリンク (*parentmem* を通過しているリンク) のとき，自由リンクが *parentmem* を通過していることを明示するために，*parentmem* に自由リンク管理アトムを挿入する．

以上の処理により，*parentmem* と *childmem* 間の自由リンクに対し，階層関係を持つ適切な自由リンク管理アトムが配置される．
