博 士 論 文

拡張の合成を容易とするメタレベルアーキテクチャの研究

指導教官　渡部 卓雄 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

田中 哲

2000 年 2 月 17 日

要 旨

ソフトウェアの高機能化・大規模化に伴い、拡張・適応可能なソフトウェアを整合的に、かつ低コストで構成するための方式が望まれている。核となるソフトウェアにモジュールを追加して拡張を行なえるようにすることは、ソフトウェアの核部分を小規模に抑えつつ、全体として多様な機能を提供するのに有効な手段である。

このような方式で拡張を行なう場合、以下の 2 点が問題になる。

1. (単一の) 拡張モジュールを適切に記述可能かどうか。
2. 拡張モジュール同士の衝突の検出/解決が可能かどうか。

上記 1 に対する回答として、メタレベルアーキテクチャという考え方が用いられてきた。この考え方では、ソフトウェアをメタレベルとベースレベルの 2 つに大きく分けて構成し、メタレベルはベースレベルの (広義の) インタプリタとして振舞うようにする。メタレベルアーキテクチャにもとづいて拡張モジュールを (メタレベルの部品として) 構成することにより、拡張方式の柔軟なデザインが可能になる。現在までに、メタレベルアーキテクチャを採用した拡張可能な言語処理系、OS などが提案されている。

しかし、メタレベルアーキテクチャの採用のみでは上記 2 は一般的には解決されない。一般に複数の拡張モジュールを同時に適用した場合には — 個々の拡張モジュールを単独で適用した場合に正しく動作したとしても — 全体が正しく動作するとは限らない。そして、拡張モジュールの開発は核モジュールや他の拡張モジュールとは独立して行なわれることがあるため、そのような複数の拡張モジュール同士の整合性を開発時に検査することも困難である。これは拡張モジュール間の相互作用が静的な形では明確になっていないためである。

本論文で提案するメタレベル構成法では、属性文法の考え方を応用し、拡張モジュール間の相互作用を静的に定義する。具体的には、メタレベルからみたベースレベルを属性付の木とし、拡張モジュールを属性の値を求める手続きの集合とする。そして、拡張モジュール間の相互作用には単一代入の属性を用いる。この構成法では、拡張モジュールを追加した場合に衝突を確実に検出できる。また、この構成法は言語処理系だけでなく、一般的なソフトウェアへ適用可能である。本論文では、提案するメタレベル構成方式の具体的な応用例として、(1)λ 計算に相当する関数型言語と、(2)LR 構文解析器を挙げる。特に (2) は狭義の言語処理系以外への適用例となっており、本方式の適用範囲の広さを示している。

目 次

第 1章

はじめに

1.1　ソフトウェアの拡張とメタレベルアーキテクチャ

大規模なソフトウェアを構成するためにもっとも重要なことは、ソフトウェアを適切に分割して個々の部分だけに注目して開発できるようにすることである。このためには、ソフトウェアをモジュールに分割することが必要であるが、さらに必須ではないモジュールをソフトウェアの外に分離してソフトウェアの基本部分の規模自体を小さくすることも有効である。この場合、分離されたモジュールはそのソフトウェアに対する拡張として後から提供される。図 1.1 はこの様子を表しており、基本部分にさまざまな**拡張モジュール**が**拡張インターフェース**を通して導入されている。

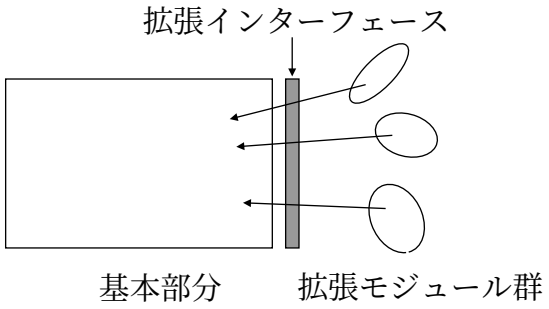


図 1.1: 拡張可能なソフトウェア

このような拡張を行なう場合、問題になるのは次の 2 点である。

- (単一の) 拡張を適切に記述可能かどうか。
- 拡張モジュール同士の**衝突**の検出/解決が可能かどうか。

これらの問題は一般的な形で解決するのは困難であり、どんな場合にでも解決できるという方法はない。したがって、個別の場合毎に対処されることが多いが、

比較的広い範囲に適用可能な形で拡張の問題を解決するアーキテクチャも存在する。**メタレベルアーキテクチャ**は通常のプログラミング言語では記述が面倒な問題を、プログラミング言語を拡張することによって容易に記述したいという動機から生まれたアーキテクチャであり、具体的な問題と問題解決のための記述法のギャップを狭めるためのものである。

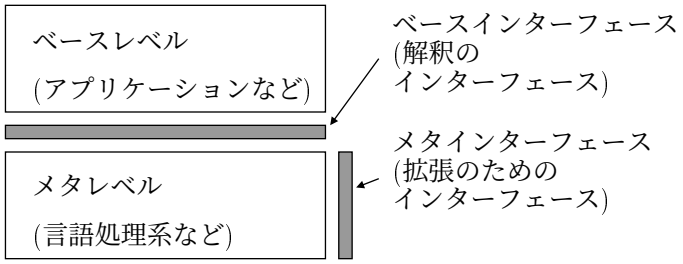


図 1.2: メタレベルアーキテクチャ

メタレベルアーキテクチャとは、プログラムを**ベースレベル**と**メタレベル**の 2 層構造としてとらえ、ベースレベルの意味をメタレベルで定義するという図 1.2 のようなアーキテクチャである。たとえば、拡張可能な言語処理系は、言語処理系自体とその拡張モジュールをメタレベル、その言語で記述されたプログラムをベースレベルとみなせる。このアーキテクチャではベースレベルの記述法をメタレベルで制御できるので、アプリケーションに適した記述法をメタレベルで実現し、ベースレベルで使用することができる。つまり、記述法を問題に適するように特殊化し、それらのギャップを狭めることができる。

メタレベルはベースレベルの意味を定義するものであるが、完全な意味を定義するのは言語処理系を作ることと同じで手間がかかりすぎる。したがって、新しいメタレベルは既存のメタレベルを拡張して作られるのが普通である。ここで、メタレベルを拡張するための界面を**メタインターフェース**と呼び、それに対してを解釈する/されるというベースレベルとメタレベルの間の界面を**ベースインターフェース**と呼ぶ。

メタレベルアーキテクチャには次の利点があり、拡張性の高いプログラムを実現するのに有効である。

- ベースレベルの記述を単純にするために用途に特化した記述法を使える。
- メタレベルはベースレベルの記述を単純にすることに集中できる。

たとえば、プログラム中に同じような記述が散在しているにもかかわらず、言

語仕様の制限[1]によってそれらの記述を関数などにまとめられないことがある。この場合、メタレベルアーキテクチャでは新しい構文を導入して、それらの記述を構文の定義部分にまとめることができ、見通しよく記述することができる。

1.2 拡張モジュールの衝突

しかし、メタレベルアーキテクチャは拡張モジュールの衝突の問題に対する解答を与えていない。これは、衝突の内容はメタインターフェースの設計に依存するが、メタレベルアーキテクチャ自体はメタインターフェース設計の指針を示していないためである。個々のメタインターフェースによって具体的にどのような問題が起こるかは異なるが、いずれにせよ、拡張モジュールの開発は基本部分や他の拡張モジュールとは分離して行なわれるため、独立に開発された複数の拡張モジュールの整合性を開発時に確認することはできない。したがって、衝突の問題は拡張モジュールの開発時に考慮するのではなく、メタインターフェースの設計時に考慮しなければならない。

衝突に関して考慮されていないメタインターフェースでは、言語機構の中で信用できる部分とそうでない部分の見分けがつかないという問題が生ずる。これは拡張モジュールの影響範囲を適切に制御できないのが原因である。その結果、拡張モジュールを開発している時点では実際に拡張モジュールが組合わさってメタレベル全体が構成された時点での挙動を把握することができず、その時点での性質である整合性を考慮することができないことになる。

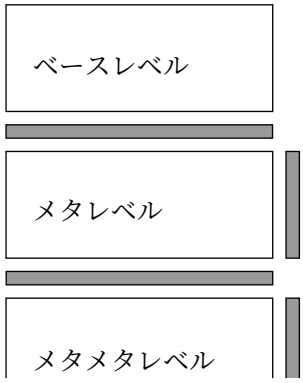


図 1.3: リフレクティブタワー

ここで、まず 3–Lisp[15] を例にとって衝突の問題点を述べる。これは 3–Lisp は**リフレクション** [20] という概念を説明するために作られ、複数の拡張モジュールの整合性を考慮していない (目的としていない) ため、問題点があからさまに現れるためである。ここで、リフレクションとは狭義にはメタレベルアーキテクチャ

[1]たとえば、記述内のパラメタの型が違うなど。

においてメタレベルの記述系がベースレベルと同じものを指す。つまり、ベースレベルが 3–Lisp という言語で記述されると同時に、それを解釈するメタレベルも 3–Lisp で記述されている。つまり、メタレベルは 3–Lisp で記述された 3–Lisp インタプリタ、つまり 3–Lisp の**メタサーキュラーインタプリタ**である。なお、メタレベルを解釈するレベルを**メタメタレベル**と呼び、メタレベルと同様な構造をもつ。一般にメタ ^{n} レベルはメタ $^{n+1}$ レベルによって解釈され、図 1.3 の**リフレクティブタワー**と呼ばれる構造を構成する。3–Lisp のメタインターフェースは**リフレクティブ手続き**で実現される。リフレクティブ手続きは (`lambda reflect` (*args env cont*) *body*) と記述され、通常の手続きと同様に呼び出すことができる。リフレクティブ手続きは呼び出されるとその時点での計算状態 (未評価の引数、環境、継続) をリフレクティブ手続きの仮引数 (*args, env, cont*) に束縛し、1 段階メタなレベル — ベースレベルで呼び出された場合はメタレベル — で *body* を評価する。*body* の評価が終了すると元のレベルの評価が再開される。リフレクティブ手続きは計算状態を引数として受けとるため、これを操作することによって次のことができる。

- 未評価の引数を解析することにより、引数を通常の式以外のものとして解釈をする。これにより、新たな特殊形式が実装できる。
- 環境に対して参照、変更を行なうことにより、任意の変数にアクセスできる。
- 継続を呼ぶことにより、制御の流れを修正できる。この継続はそのリフレクティブ手続きが適用された時点で得た継続でなく、以前に呼び出された時点で得たものでもよい。これにより、新しい制御構造を導入できる。

また、リフレクティブ手続き内でメタレベル内の大域変数を操作することもできる[2]。そして、インタプリタはメタレベル内の大域変数に束縛された手続きとして実現されているため、インタプリタを再定義することも可能である。つまり、拡張モジュールにおいてインタプリタを呼び出す必要があった場合に、インタプリタが再定義されていないという保証は得られない。さらに、プリミティブ (`car` など) が再定義されていないという保証もなく、3–Lisp で頑健な拡張モジュールを記

[2]変数の代入はリフレクティブ手続きで実現できるので、メタメタレベルで環境を操作するとみなせる。

述するのは不可能である。また、拡張モジュールが衝突した場合の挙動の変化は一般には予想できず、衝突を検出することも困難である。

この問題の原因はインタプリタとして表現したメタレベルの任意の変更を許すメタインターフェースを採用したことにある。したがって、より制限的なメタインターフェースを採用することによって、問題を減少させることができる。

たとえば、CLOS MOP(Meta Object Protocol)[10] はオブジェクトの挙動を**メタオブジェクト**によって定義するという図 1.4 のようなメタレベルアーキテクチャを実現するものである。ここで、メタオブジェクトでは CLOS のオブジェクト指向関連の機構のみを変更することができ、Common Lisp 部分は変更することができない。したがって、拡張モジュールを記述する場合にその部分についての挙動が変更されることを考慮する必要がない。さらに、メタオブジェクトは個々のオブジェクトと対応しているため、メタオブジェクトの影響はそのメタオブジェクトに属するオブジェクトに限られ、システム全体には及ばない。つまり、複数のメタオブジェクトを干渉させずに動作させることが可能であり、システムの中で複数の拡張モジュールを同時に動作させることができる。ただし、CLOS MOP では**継承**を利用してメタオブジェクトを定義することが行なわれる。この場合、一つのメタオブジェクトの中に、必ずしも一つのメタオブジェクトで使われるとは考えられていなかった複数の記述が混在することになる。したがって、それらが衝突して動作がおかしくなる場合がある。また、CLOS MOP の衝突解決はオブジェクト指向に依存した設計であり、メタレベルアーキテクチャ一般に適用できるものではない。

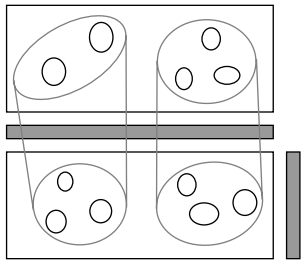


図 1.4: オブジェクト指向メタレベル

1.3 衝突が検出できるメタレベルの構成法

本研究では複数の拡張モジュールが同時に適用された場合の問題について次のような性質を持つ解決法を提案する。

- 確実に衝突の検出ができる。

- メタレベルアーキテクチャ一般に適用できる。

衝突は拡張モジュールを実際に組み合わせた時点で生じるため、開発者以外でも確実に衝突を検出できることが重要である。また、衝突はメタレベルアーキテクチャ一般で問題になるため、個々のメタレベルの設計によらず適用できる解決法が望ましい。衝突検出に必要なことは拡張モジュールの影響範囲を適切に定義し、干渉の可能性を明らかにすることである。しかし、範囲を決めるためにはある程度システムの構造を仮定しなければならない。たとえば、CLOS MOP ではシステムがオブジェクトの集まりでできていることを仮定して、影響範囲をオブジェクト単位に制限することができたのである。本研究では次のような構造を仮定する。

ベースレベルの記述が**抽象構文木**として表現される。

計算機で機械処理を行なう扱うほとんどの記述は木構造をなしており、この仮定が成り立つ。つまり、本研究は大部分のメタレベルアーキテクチャに適用可能である。そして、ベースレベルの解釈は**属性文法**的に抽象構文木の各節に**単一代入**な**属性**を付与することによって行なう。拡張モジュールは属性を評価する**評価規則**の集まりとして定義され、拡張モジュールを追加する場合に起こる衝突は、必要な属性に評価規則が定義されていないか、2 つ以上の評価規則が定義されているか、どちらかの形で現れ、衝突が実際に起きた場合には確実に検出できる。

本論文ではこの解決法を簡単な関数型言語と LR 構文解析器の例によって説明する。LR 構文解析器は狭義の言語処理系以外のメタレベルアーキテクチャにも適用可能であることを示す例であり、また、拡張モジュールの衝突についてはそれぞれについて具体例をあげて説明する。

1.4 論文の構成

以下、2 章で研究背景について述べ、3 章でメタレベルの構成法を定義する。4 章では定義した構成法に基づいて簡単な関数型言語と LR 構文解析器の例を説明し、5 章でこの構成法を支援するライブラリおよび上記の例の Scheme による実装について述べる。6 章ではメタレベルをモジュール化する他の手法との比較を行ない、結論を述べる。

第 2章

研究背景および関連研究

本章ではメタレベルアーキテクチャについて述べ、メタレベルアーキテクチャの観点から AOP, オブジェクト指向、属性文法、デザインパターンについて述べる。

2.1 メタレベルアーキテクチャ

メタレベルアーキテクチャとはソフトウェアを記述系として認識し、なんらかの記述とそれを解釈する部分に分割するという図 2.1 のようなアーキテクチャである。図 2.1 の上部の長方形はベースレベル、下部の長方形はメタレベルであり、上部のベースレベルの意味を下部のメタレベルが定義していることを示している。メタレベルの右の灰色の長方形はメタインターフェースと呼ばれるメタレベルを変更するための界面を示しており、メタレベルの右上の影はメタレベルが固定されたものではなく変更される可能性があることを表している。また、ベースレベルとメタレベルの間の灰色の長方形はメタレベルがベースレベルを解釈するという界面を示しており、ベースインターフェースと呼ばれる。

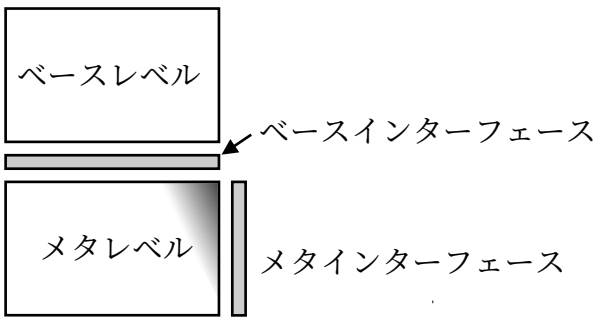


図 2.1: メタレベルアーキテクチャ

メタレベルアーキテクチャとみなせるシステムとして最も典型的な例は言語処理系である。インタプリタ (メタレベル) は記述されたプログラム (ベースレベル) を解釈実行し、コンパイラもプログラムの解釈を行なってオブジェクトコードを生成する。しかし、メタレベルアーキテクチャとみなせるシステムは言語処理系には限らず、なんらかの記述を解釈するシステムはすべてメタレベルアーキテクチャとみなせる。たとえば、アプリケーションは設定ファイルを解釈して挙動を決定する。また、HTML 記述はブラウザによって解釈されて画面に表示するという意味に対応づけられる[1]。さらに、記述が表立って現れないものにもメタレベルアーキテクチャは適用できる。たとえば、オブジェクトコードはプロセッサによって解釈実行される。また、OS におけるプロセスを考えると、プロセスが起動するシステムコールの列をカーネルが意味を解釈しているとみなせる。

メタレベルの変更はメタインターフェースを通して行なわれる。このため、メタレベルを変更する場合に意識しなければならないメタレベルの構造はメタインターフェースに依存する。なお、メタインターフェースはプログラムの内部 (ベースレベルもしくはメタレベル) から利用されるだけでなく、外部から利用される場合もある。

ここで、言語処理系などの拡張とは基本的には記述の解釈法を変更することであり、記述とその解釈を分離するメタレベルアーキテクチャは、そのような拡張に対する統一的な見方を提供しているといえる。つまり、記述の解釈法はメタレベルによって定義され、メタレベルの変更によってその定義の変更が行なわれる。ここで、メタレベルの変更方法はとくに制限されていないため、どのような変更方法もメタインターフェースとして解釈でき、今までに提案されたさまざまな拡張法をメタレベルアーキテクチャとして解釈することができる。

2.2 AOP とメタレベルアーキテクチャ

AOP(*Aspect Oriented Programming*)[11] はプログラムの複数の**側面** (*aspect*) を別々に記述し、それを *Aspect Weaver* と呼ばれるプログラムで機械的に合成することにより最終的なプログラムを得るという考え方である。ここで、個々の側面の記述をアスペクトという。AOP を単純にメタレベルアーキテクチャとして解釈

[1] 音声等に対応づけられることもある。

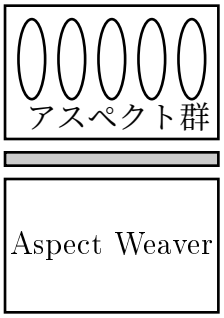


図 2.2: メタレベルアーキテクチャとしての AOP

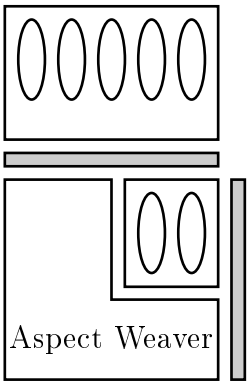


図 2.3: メタレベル的なアスペクトを持つ AOP

すると 図 2.2 のようになる。つまり、アスペクト群を Aspect Weaver が解釈し、プログラム全体の意味を決定するのである。

しかし、個々のアスペクトは (Aspect Weaver に依存して) それぞれで性質が異なり、他のアスペクトの意味に影響を与えるようなメタレベル的な意味を持つこともある。このような意味を考慮すると 図 2.3 のようになる。この場合、他のアスペクトの意味に影響を与えるアスペクト (群) はメタレベルの拡張に対応し、その他のアスペクト (群) がベースレベルに対応するようなメタレベルアーキテクチャとみなせる。このようにみなした場合、AOP は一般のメタレベルアーキテクチャに対してメタレベルやベースレベルを複数のモジュールに分割して記述するという特徴を持つ。ただし、分割したモジュール間で起こる衝突の解決方法はとくに決まっておらず、衝突は個々の実装で対処される。

AOP を支援する Java の拡張として AspectJ[17] がある。AspectJ では Java のプログラムにさまざまなタイミング (*crosscut*) を定義し、そのタイミングを指定して他のアスペクトの処理を割り込ませることによって、プログラムを合成する。

ここで、タイミングには名前、型、クラスで同定したメソッドの起動時/終了時などを指定でき、また実行時にオブジェクト単位でアスペクトの有効、無効を制御することもできる。つまり、割り込まれる側をベースレベル、割り込む側をメタレベルとみなせば、メタレベルのアスペクトは、ベースレベルのメソッド呼び出しの意味を変更ていることになる。また、この機構は Lisp の *advice*[8] に似て

いるが、型情報などを利用して多様なタイミングを指定できる。

この合成法はアプリケーションに依存しない一般的なな方法であり [12]、応用を限定せず広い範囲に適用できるが、継承との組み合わせで衝突が起きることも指摘されている [2]。

2.3 リフレクションとメタレベルアーキテクチャ

リフレクション [20] は狭義にはメタレベルアーキテクチャの中でメタレベルとベースレベルが同じ記述系で記述されるものを指す。また、広義には単にメタレベルの状態を知ること (イントロスペクション) が出来るだけのようなものでもリフレクションと呼ばれることがある。

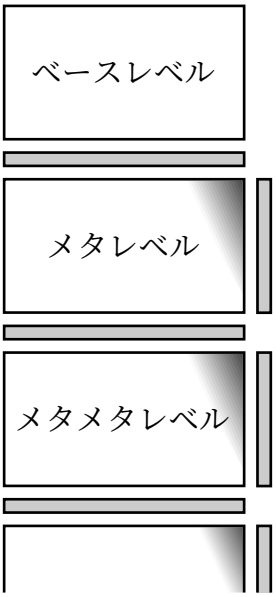


図 2.4: 狭義のリフレクション

狭義のリフレクションではメタレベルのプログラムをベースレベルと同じ記述系で記述する。そして、メタレベルはメタメタレベルによって解釈され、一般にメタ ^{n} レベルはメタ $^{n+1}$ レベルで解釈されると考える。このメタ ^{i} レベルの連なりは図 2.4 のような構造となり、これをリフレクティブタワーと呼ぶ。

このような構造により、メタレベルもベースレベルと同様に拡張できるほか、ベースレベルでのプログラミングと同様な感覚でメタレベルのプログラミングが可能であり、容易にメタレベルのプログラミングができると期待された。しかし、これはベースレベルとメタレベルの記述対象は異なることが多いということを無視した見方である。つまり、メタレベルが常に (ベースレベルの) プログラムを扱うものであるのに対し、ベースレベルがプログラムを扱う状況は少ない。それにもかかわらず、リフレクションでは同じ記述系で両方を記述するため、それぞれに適した言語を最初から用意することができないのである。つまり、メタレベルのためにプログラムを扱う機能に特化した言語を用意することができず、メタレベル

を記述できないほど用途に特化された言語をベースレベルに使用することもできない。このように、リフレクションでは個々の場合に適した言語を使用するというメタレベルアーキテクチャの基本的な利点を捨てがちになる。

また、リフレクションという概念には衝突の解決法は含まれておらず、衝突は個々の実装で対処される。

2.4 オブジェクト指向にもとづくメタレベルの再利用

オブジェクト指向はシステムをオブジェクトの組み合わせで表現するというパラダイムである。ここで、オブジェクトはメッセージの送受信により外部との相互作用を行ない、内部のデータは隠蔽される。このため、オブジェクト内部の変更はメッセージの仕様を変更しない限り外部には波及せず、変更が容易となる。また、継承によりオブジェクトを拡張することができる。このようなことから、オブジェクト指向パラダイムは再利用性や拡張性の高いシステムの構築に有用であり、メタレベルの構成にも有効である [1]。オブジェクト指向的なメタレベルの場合、ベースレベルがオブジェクトの組み合わせで表現されると同時に、メタレベルもオブジェクトによって表現され、図 2.5 のような構成となる。メタレベルを構成するオブジェクトをメタオブジェクトと呼び、ベースレベルを構成するオブジェクトはメタオブジェクト (群) によって意味を定義される。オブジェクト指向にもとづいて構成されたメタレベルでは、既存のオブジェクトの意味を定義するメタオブジェクトとは別に新しいメタオブジェクトを生成することにより、既存のオブジェクトの意味を変更せずに (新しいオブジェクトに対して) 異なる意味づけをすることができる。また、ベースレベルとメタレベルがともにオブジェクトという同じもので表現されることから、狭義のリフレクションとしても解釈できる。

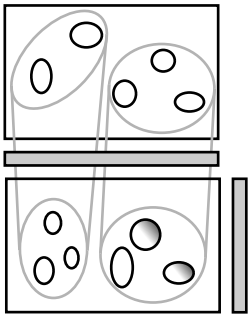


図 2.5: オブジェクト指向的なメタレベル

オブジェクト指向的に構成されたメタレベルには多くの例がある。Smalltalk[5] ではクラスの挙動をメタクラスが制御し、CLOS[10] ではメソッド選択などオブ

ジェクト指向の機能がメタオブジェクトとして表現されており、それをメタオブジェクトプロトコルによってカスタマイズできる。ABCL/R[21] は並行オブジェクト指向言語であり、オブジェクトそれぞれに対応してメタオブジェクトが一つずつ存在するというモデルを持つ。また、[22] ではオブジェクトのグループに対してメタオブジェクトが存在するという *Group-Wide Reflection* というモデルを述べている。AL–1/D[13] は分散システムの記述を支援するオブジェクト指向言語であり、メタレベルをオブジェクト指向的に記述された複数のモデルとして表現している。

2.5 属性文法とメタレベル

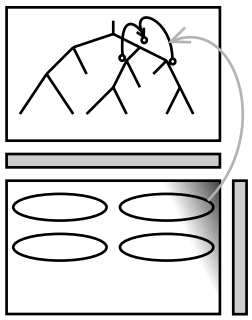


図 2.6: 属性文法的なメタレベル

属性文法 [14] は構文木の各ノード上に属性 (群) を付加し、構文木の構造に沿って属性の値を計算するというパラダイムである。属性間の関係は評価規則によって定義され、属性の値は単一代入によって決定される。木の構造に沿って計算を定義することにより、木の巡回を明示的に記述する必要がなく、言語の仕様や処理系の記述に適している。特にコンパイラの記述によく利用されるが、インタプリタの記述にも利用できる [4]。言語処理系はメタレベルとみなすことが出来るため、属性文法はメタレベルの記法とみなすことが出来る。図 2.6 は、ベースレベルに構文木として記述されたプログラムがあり、ある節の属性がほかの属性から計算され、その計算規則はメタレベル内で定義されていることを示している。

2.6 デザインパターンによる再利用性の高い構造

デザインパターン [3] はいくつかのモジュールで構成される程度の規模の有用な設計であり、多くの種類が提案されている。デザインパターンの多くはソフトウェアの再利用性や柔軟性を高めるための設計であり、ソフトウェアの拡張を容易と

するために使えるものも多い。たとえば、Composite パターンはさまざまな種類の要素を木構造に組み立てるものであり、新しい要素を容易に導入できる。また、衝突に関する扱い個々のパターンに依存する。たとえば、Composite パターンで同一のクラス名の要素を複数導入すればコンパイル時に検出される[2]が、Chain of Responsibility パターンで、同一の要求を処理するハンドラが複数ある場合には単純に最初のハンドラがその処理を行ない、それ以降のハンドラは無視される。

本研究は拡張が容易なメタレベルの設計について考察しており、提案する構成法にはデザインパターンとして提案されている設計と重なる部分がある。たとえば抽象構文木の表現は Composite パターンとして実現できるし、インタプリタをメタレベルとして表現する場合には Interpreter パターンに非常によく似た構成となる。また、抽象構文木上を巡回して計算を行なうにはその計算に関する評価規則を各構文要素毎に記述することになるが、その計算に関する評価規則をまとめてモジュールにすることは Visitor パターンに相当し、各構文要素に対応するモジュールに評価規則を記述することは Composite パターンにおける Leaf クラスや Composite クラスにオペレーションを定義することに相当する。ただし、評価規則自体は自立した実体であり、どのモジュールに属しても同様の効果を持つため、これらの違いは純粋にモジュール分割の単位の違いとなる。

[2]同一の名前を持つクラスを一つのバイナリに入れることはできないので、遅くともリンク時には検出される。

第 3章

拡張を容易とするメタレベルの構成法

本章ではメタレベルの具体的な構成について述べる。そのために、まず構成方針について述べ、その方針にしたがうメタレベルのための抽象構文、属性を定義する。そしてメタレベルとメタレベルモジュールを定義する。

3.1 メタレベルの構成方針

1 章で述べたように、メタレベルは拡張されることを前提としたプログラムである。したがって、一つの目的のためにメタレベル全体が一度に記述されることは稀であり、通常は独立に記述されたモジュールの組み合わせとして構成される。つまり、実際にモジュールを組み合わせる人物が個々のモジュールの内容を深く理解していることは期待できない。したがって、モジュールの組み合わせは簡単に理解できる単純なものでなければならない。

メタレベルの構造として一般的なものは、言語処理系の手続き的な実装をある程度抽象化して拡張の手がかりを公開するというものである。たとえば、コンパイラの場合には、コンパイラが構文解析、意味解析、コード生成などといった要素からなっているものとして抽象化し、構文解析と意味解析の間に処理を挿入可能にして構文木を変換する拡張を可能とする、などということが行なわれる。しかし、このような抽象化によって作られたメタレベルは必ずしも拡張に適したものとはならないことがある。たとえば、上記の構文木を変換する拡張では、意味解析の結果 (型など) を利用して変換を行なおうとすると破綻する。一般に、手続

き的なプログラムを抽象化してメタレベルを作ると、制御の流れが捨象しきれずに残り、拡張の足枷となることが多い[16]。また、手続き的なプログラムは意味が複雑でモジュール間の関係を制御することが困難であるという問題もある。したがって、より宣言的な記述でメタレベルを構成する必要がある。

また、適用範囲の広いメタレベルを設計するためには、個々のアプリケーションに依存しない抽象的な構造にそってメタレベルの構造を構成しなければならない。そこで、本研究ではベースレベルに次の仮定をおき、その構造を手がかりにメタレベルを構成する。

ベースレベルの記述が抽象構文木として表現される。

この仮定は非常に広範囲のメタレベルについて成り立つ。まず、プログラム言語であるかどうかにかかわらず、ある程度複雑な構造を持った記述を定義する場合、文脈自由文法がよく使われる。このような場合には記述は木構造をもつため、すべてこの仮定を満たすことになる。また、より単純な構造しか持たないような記述の場合、その文法を文脈自由文法で記述することは通常容易であるため、この場合も仮定を満たすことになる。この仮定を満たすことが困難なメタレベルは、ベースレベルが木構造として表現することが困難もしくは自然ではないものである。たとえば、画像データは 2 次元の格子構造をなしており、これを解釈するプログラムをメタレベルとみなすことはできるが、画像を木構造として表現するのは不自然である[1]。

この仮定から、ベースレベルプログラムに共通する構造として、そのプログラムがしたがう文法が存在することになる。個々のメタレベルは特定の文法にしたがうベースレベルプログラムを解釈するが、なんらかの文法が存在するということ自体はすべてのメタレベルに共通の性質である。したがって、抽象構文木の文法の一般形に対応するようにメタレベルの構造を設計することによって、上記の仮定のみに依存してメタレベルを構造化することができる。

本章では抽象構文木の一般形について定義した後、その構造にしたがって構造化されたメタレベルを定義する。メタレベルは属性文法によって抽象構文木上の属性を宣言的に定義することでベースレベルの意味を定義する。さらに、直接に

[1] もちろん、恣意的な木構造を与えることはできる。

はベースレベルプログラムに現れない構造にも同様な構造化手法を利用するために、抽象構文木をメタレベルが生成することを可能とする。これは高階属性文法 [19] を拡張したものとなっている。

3.2 抽象構文

抽象構文は、**具象構文** (通常の人間が記述するプログラムをそのまま扱う構文) からプログラムの実行に必要な本質的な木構造だけを扱いやすいように抽出したものである。木構造は代数データ型として定義され、抽象構文木はその型の値として表現される。ここで、型は非終端記号と対応し、構成子は導出規則に対応する。

たとえば、S 式で表現した λ 計算の具象構文は次のようになる。

式 → 識別子
| ‘(’ 式 式 ‘)’
| ‘(’ ‘lambda’ ‘(’ 識別子 ‘)’ 式 ‘)’

ここで、引用符で括ってあるもの (‘(’, ‘)’, ‘lambda’) は終端記号、そうでないもの (識別子、式) は非終端記号である。

この具象構文に対して抽象構文は次のようになる。

式 → ‘変数参照’ 識別子
| ‘関数適用’ 式 式
| ‘関数抽象’ 識別子 式

この抽象構文は具象構文から次のようにして変換されたものである。

- 具象構文の導出規則に現れる終端記号は消去する。
- 導出規則の右辺の先頭にその導出規則を一意に示す終端記号を加える。

このようにして作られた抽象構文は、具象構文の導出規則に現れる非終端記号がその順番通りに抽象構文の導出規則に現れ、具象構文の木構造と同じ構造を抽象構文でも表現できる。また抽象構文は、非終端記号と型、終端記号と構成子を対応付けることによって、代数データ型そのものとなる。

また、抽象構文木の葉は右辺が単一の終端記号であるような導出規則に対応した節となる。つまり、上記の識別子という非終端記号が ‘シンボル’ という終端記号に導出されるとすると、次のようになる。

識別子 $\rightarrow$ ‘シンボル’

たとえば、$(f\ x)$ という記述の抽象構文木は図 3.1 のようになる。各長方形は節を表しており、そのなかはその節がどの導出規則で導出されているかを示す終端記号が書かれる。

なお、必要な場合には非終端記号の右肩に添字をつけて非終端記号の出現を区別することにする。この場合、右肩の添字は非終端記号自体を変えるものではないので、添字の違いは非終端記号の違いとはみなさない。さらに、終端記号は必ず導出規則の右辺の先頭に現れて区別可能であるため、引用符は省略することがある。つまり、たとえば、上記の文法は次のようにも書かれる。

式^0 $\rightarrow$ 変数参照 識別子^1
| 関数適用 式^1 式^2
| 関数抽象 識別子^1 式^2
識別子^0 $\rightarrow$ シンボル

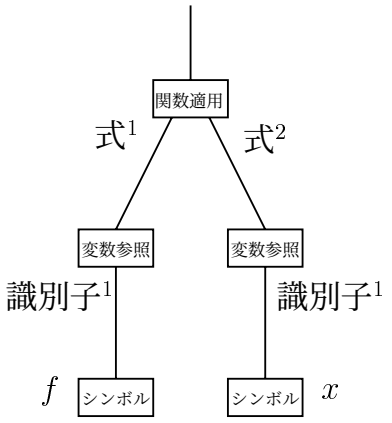


図 3.1: $(f\ x)$ の抽象構文木

3.3 抽象構文の定義

抽象構文の文法 $G = (N, T, P, S)$ は次のように定義される。

- N：非終端記号の集合
- T：終端記号の集合
- P：導出規則の集合

- S：開始記号

ここで導出規則 (P の要素) は次の形をしている。

$$X_0 \to X_1 X_2 \ldots X_n$$

ただし $X_i \in N(i = 0 \vee 1 < i \leq n), X_1 \in T$ であり、X_1 は P 中の導出規則を一意に同定する ($p = p' \longleftrightarrow X_1 = X_1'$ ($p = X_0 \to X_1 X_2 \ldots X_n, p' = X_0' \to X_1' X_2' \ldots X_n' \in P$)) ものとする。これら条件により X_1 は代数データ型における構成子とみなすことができる。また、通常の文脈自由文法では $X_0 \in N, X_i \in T \cup N(0 < i \leq n)$ であるから、抽象構文の文法は文脈自由文法を制限したものになっている。

3.4 属性

ベースレベルプログラムは抽象構文で表現され、メタレベルはその意味を定義する。ここで、ベースレベルプログラムの意味は抽象構文木の節に付随する属性の値として表現する。また、ベースレベルプログラムには含まれるが抽象構文木だけでは表さない詳細な情報 — たとえば識別子の具体的な名前など — は終端記号に付随する属性として扱う。

たとえば、関数型言語では、ある式の意味はその式の値であり、その値はその式に含まれる自由変数の値に依存する。つまり、自由変数の値の定義 (環境) から値への関数として式の意味を表現できる。したがって、関数型言語のプログラムの抽象構文木の各節に意味に対応する関数を属性としてつけることによって、ベースレベルプログラムの意味づけを行なうことができる。ここで、環境を σ とし、識別子の名前が識別子の label という属性に格納されていて、変数参照の構文からは 識別子^1.label と参照できるとすれば、変数参照の意味は $\lambda\sigma.\sigma$ 識別子^1.label と表現できる。変数参照の構文の eval という属性をこの意味として定義することを次のように書く。

変数参照 ： eval := $\lambda\sigma.\sigma$ 識別子^1.label

同様に関数適用と関数抽象の意味も次のように定義できる。

関数適用 ： eval := $\lambda\sigma.$(式^1.eval σ)(式^2.eval σ)

関数参照 ： eval := $\lambda\sigma.\lambda v.$(式^2.eval σ[識別子^1.label $\leftarrow v$])

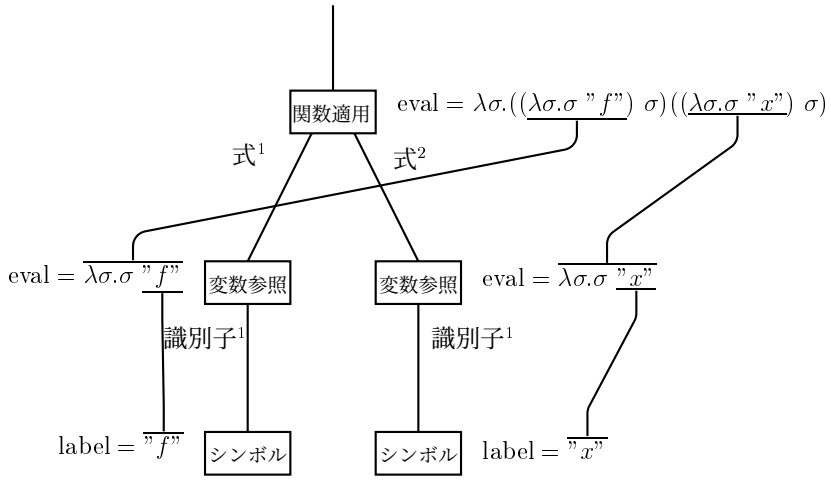


図 3.2: $(f\ x)$ の属性つき抽象構文木

図 3.1 の抽象構文木上でこれらの属性の値を求めると図 3.2 のようになる。

しかし、このように抽象構文木上の属性によってプログラムの意味を表現すると、抽象構文木とは異なる構造をもつ意味を扱うことが相対的に難しくなる。そこで、抽象構文木を属性として計算して既存の抽象構文木に接ぎ木できるようにして、人間が記述したプログラムに直接は現れていない構造も抽象構文木として表現することとする。

3.5 メタレベル

文法 $G = (N, T, P, S)$ にしたがう抽象構文木上の属性は 7 つ組 (A, B, R, N, T, P, S) で定義され、これを**メタレベル**と呼ぶ。ここで、A は属性の集合、B は非終端属性の集合、R は評価規則の集合であり、残りの N, T, P, S は G を表現する。また評価規則 (R の要素) は構成子と評価式の組であり、次のように表記する。

$$X_1 : A_0 := f(A_1, \ldots, A_m)$$

ここで X_1 は P 中の導出規則 p を示す構成子であり、$p = X_0 \to X_1 X_2 \ldots X_n$ とする。評価式 $A_0 := f(A_1, \ldots, A_m)$ の $:=$ は属性の単一代入を表す。A_i は属性の参照であり、次のいずれかの形をとる。以下では属性 (A の要素) と非終端属性 (B の要素) をそれぞれ a および b というメタ変数で表す。

図 3.3: 抽象構文木の節

- a : X_0 の属性 a を示す。
- b : X_0 の非終端属性 b を示す。
- $X_i.a$: ここで $0 < i \leq n$ であり、X_i の属性 a を示す。
- $b.a$: X_0 の非終端属性 b の属性 a を示す。
- X_i : ここで $0 \leq i \leq n$ であり、X_i を根とする構文木自体を示す。ただしこれは右辺でしか使用できない。

この属性参照の記法は通常の属性文法のものとは異なり X_0 の属性を示すときには $X_0.$ を前置しない。これは $X_i \quad (0 < i \leq n)$ という非終端記号自体も X_0 の属性の一種とみなしているためである。

また、f は評価関数であり、メタレベルによって決まる適当な言語によって記述される。ここで、評価関数内では抽象構文木を生成するための構成子を**プリミティブ**として使えるものとする。つまり、一般に抽象構文木の各節は図3.3の形になる。また、評価式の定義時にわかりやさのために適当な式変形を行なったものをもって定義とすることがある。たとえば、λ 記法で評価関数を記述する場合に、eval := $(\lambda l.\lambda\sigma.\sigma\ l)($識別子^1.label$)$ と記述する代わりに β 変換を行なって eval := $\lambda\sigma.\sigma$ 識別子^1.label と記述する。

このメタレベルによってベースレベルプログラムを解釈するということは、ベースレベルプログラムの抽象構文木の根に定義された属性を計算することである。

3.6 モジュール

メタレベルをモジュールの組み合わせで定義するため、**メタレベルモジュール**を定義する。メタレベルモジュール ΔM は 6 つ組 $(\Delta A, \Delta B, \Delta R, \Delta N, \Delta T, \Delta P)$ で定義され、これはメタレベルの開始記号以外の差分である。つまり、ΔA, ΔB, ΔR, ΔN, ΔT, ΔP は属性、非終端属性、評価規則、非終端記号、終端記号、導出規則それぞれの集合に追加する要素の集合である。

また、メタレベルモジュール ΔM_1 と ΔM_2 の合成を $\Delta M_1 + \Delta M_2$ と記述し、次のように定義する。

$$
\begin{aligned}
\Delta M_1 + \Delta M_2 \quad = \quad & (\Delta A_1 \cup \Delta A_2, \Delta B_1 \cup \Delta B_2, \Delta R_1 \cup \Delta R_2, \\
& \Delta N_1 \cup \Delta N_2, \Delta T_1 \cup \Delta T_2, \Delta P_1 \cup \Delta P_2) \\
& \text{where } \Delta M_1 = (\Delta A_1, \Delta B_1, \Delta R_1, \Delta N_1, \Delta T_1, \Delta P_1) \wedge \\
& \Delta M_2 = (\Delta A_2, \Delta B_2, \Delta R_2, \Delta N_2, \Delta T_2, \Delta P_2)
\end{aligned}
$$

ここで、メタレベルモジュールの合成は組の各要素に対する集合の加算で定義されているため、交換律と結合律が成り立つ。

また、メタレベルモジュール ΔM と開始記号 S を組み合わせたメタレベルを $(\Delta M, S)$ と記述し、次のように定義する。

$$
\begin{aligned}
(\Delta M, S) \quad = \quad & (\Delta A, \Delta B, \Delta R, \Delta N, \Delta T, \Delta P, S) \\
& \text{where } \Delta M = (\Delta A, \Delta B, \Delta R, \Delta N, \Delta T, \Delta P)
\end{aligned}
$$

3.7 メタレベルモジュールの衝突

一般に、拡張可能なプログラムにおいて複数の拡張を適用する場合、それらの拡張が衝突して結果の予測が困難になる場合がある。ここで、拡張の衝突とはある拡張が他の拡張に影響されて意図された動作をしないことである。このようなことが起こるのは、一般に拡張を実装する時点ではその拡張が使われる時点でどのような拡張と組合わさって動作するか予想できず、他の拡張の影響を把握しきれないためである。

本研究のメタレベル構成法ではメタレベルモジュール間 (評価規則間) の相互作用は抽象構文木上の属性を通してのみ行なわれるため、衝突は次の 3 つの形で起こる。

1. 必要な属性に評価規則が定義されていない。
2. 単一の属性に複数の評価規則が定義されている。
3. 属性の値が意図通りのものでない。

ここで、メタレベルがベースレベルプログラムの意味を解釈する場合には根の属性から依存関係にしたがって必要な属性を求めるため、上記 1, 2 の衝突が起きなければ意味を一意に決定することができる。ただし、3 の衝突が起きている場合には決定された意味はプログラマの意図通りのものではない。

1 の必要な属性が定義されていない状況は、複数のメタレベルモジュールの組み合わせにより、想定していなかった構文が現れるときに起こる。この場合、その構文上の属性を定義するモジュールを加えることによって衝突を回避できる。

また、2 の属性の定義が 2 つ以上存在する状況 (多重定義) は、組み合わせることができないモジュールを組み合わせたときに起こる。属性の定義は一意でなければならないので、この場合はどちらかのモジュールを取り除いて衝突を回避しなければならない。

3 の属性の値が意図しないものになる状況は、拡張を開発した時点で前提としていたモジュールが入れ換えられたときに起こる。逆にいえば、モジュールの追加はこの衝突の原因にはならず、開発時のモジュールが存在する限りこの衝突は起こらない。これはモジュールの追加では既存の属性の定義を変更できない (上記 2 の衝突がおこる) ためである。

これらの衝突のうち、1, 2 が起きた場合にはベースレベルプログラムの解釈を行なうことが不可能となる。これは必要な属性の定義が存在しなければ計算が続けられず、また、2 つ以上あればどの定義を適用するか判断できないからである。また、3 の衝突はモジュールを入れ換えない限り起きないため、モジュールを追加のみによってメタレベルを構成すればよい。つまり、それぞれの拡張モジュールを開発した時点では意図通りの計算を行なうモジュールが存在したわけであり、それらのモジュールを使ってメタレベルを構成すれば衝突が起きないことが保証

される。

つまり、メタレベルを構成する場合に、使用する拡張が開発時に使っていたモジュールをすべて含むように構成すれば、確実に衝突が検出されて実行を停止することができる。

第 4 章

メタレベル構成法によるメタレベル

本章では 3 章で述べたメタレベルの構成法を用いて構成されたメタレベルについて述べる。4.1 節では単純な関数型言語について、4.2 節では LR 構文解析器について述べる。また、4.3 節ではそれらの合成について述べる。

4.1 単純な関数型言語

本節では λ 計算に相当する単純な関数型言語のメタレベルについて述べる。このメタレベルはベースレベルのプログラムを解釈してその意味を決定する典型的なメタレベルアーキテクチャを構成する。

4.1.1 ベースレベルの構文

ベースレベルの構文は次のとおりである。ここで、E, S は式とシンボルを表す非終端記号とする。

$$
\begin{aligned}
E^0 \quad &\rightarrow \quad \mathrm{Var}\ S^1 \\
&\mid \quad \mathrm{App}\ E^1\ E^2 \\
&\mid \quad \mathrm{Abs}\ S^1\ E^2
\end{aligned}
$$

これらの構文は λ 計算の 3 種類の式 (変数参照、関数適用、関数抽象) に対応している。

変数参照 $E^0 \rightarrow \mathrm{Var}\ S^1$ という構文によって変数参照を表す。S^1 は変数の名前を表すシンボルである。

関数適用 $E^0 \rightarrow \mathrm{App}\ E^1\ E^2$ という構文によって関数適用を表す。E^1 は適用される関数を表す式であり、E^2 は適用する実引数を表す式である。

関数抽象 $E^0 \rightarrow \mathrm{Abs}\ S^1\ E^2$ という構文によって関数抽象を表す。S^1 は仮引数を表すシンボルであり、E^2 は関数のボディを表す式である。

4.1.2 基本的なメタレベル

関数型言語のプログラムの意味は、ある環境においてそのプログラムを評価した結果の値によって決まる。ここで、環境というのはそのプログラムに含まれる自由変数の値を決定するものである。したがって、環境から評価結果への関数を意味とみなすことができる。そこで、ある式の意味を関数で表現したものをその式の根の eval という名前の属性に格納する。また、シンボルはその識別子を label という名前の属性をもつものとする。つまり、識別子の型を L、この言語が扱う値の型を V とすると、label : L、eval : $(L \rightarrow V) \rightarrow V$ となる。

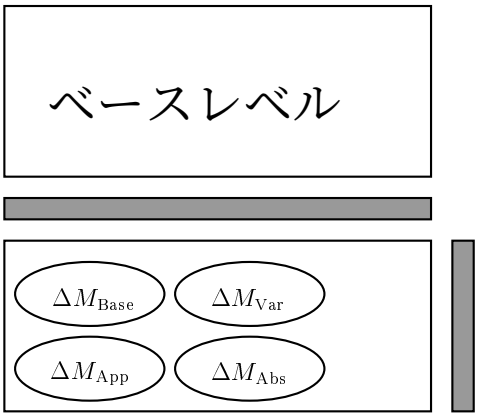


図 4.1: 基本的なメタレベルのモジュール

図 4.1 のように 4 つのモジュール ΔM_{Base},ΔM_{Var},ΔM_{App},ΔM_{Abs} をまとめることにより、この言語の基本的なメタレベルを実現できる。これらをまとめたメタレベルモジュール ΔM_{L} およびメタレベル M_{L} を次のように定義する。

$$\begin{aligned} \Delta M_{\mathrm{L}} &= \Delta M_{\mathrm{Base}} + \Delta M_{\mathrm{Var}} + \Delta M_{\mathrm{App}} + \Delta M_{\mathrm{Abs}} \\ M_{\mathrm{L}} &= (\Delta M_{\mathrm{L}}, E) \end{aligned}$$

ここで、E は式を表す開始記号であり、ΔM_{Base} で定義されている。

ベースモジュール $\Delta M_{\mathbf{Base}}$ 言語の基本的な要素 (シンボル S と属性 eval が存在

すること) を定義するモジュール

$$
\begin{aligned}
\Delta M_{\mathrm{Base}} &= (\{\mathrm{eval}, \mathrm{label}\}, \phi, \{r_{\mathrm{Sym}}\}, \{S\}, \{\mathrm{Sym}\}, \{p_{\mathrm{Sym}}\}) \\
p_{\mathrm{Sym}} &= S^0 \to \mathrm{Sym} \\
r_{\mathrm{Sym}} &= \mathrm{Sym} : \mathrm{label} := \mathrm{Sym.label}
\end{aligned}
$$

ここで、終端記号 Sym は label という属性をもっているものとする。図 4.2 はこのモジュールが定義する属性の様子を示したものである。左側は Sym 節に label という属性が定義されて、その値は終端記号 Sym の label 属性からコピーされることを表現しており、右側は 図 3.2 の構文木の左下の節がこのモジュールが定義する Sym 節の例となっていることを表現している。

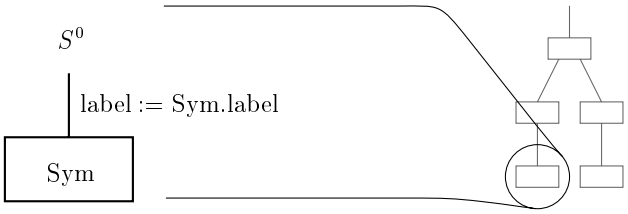


図 4.2: ベースモジュール ΔM_{Base}

変数参照モジュール $\Delta M_{\mathbf{Var}}$ 変数参照の構文を定義するモジュール

$$
\begin{aligned}
\Delta M_{\mathrm{Var}} &= (\phi, \phi, \{r_{\mathrm{Var}}\}, \phi, \{\mathrm{Var}\}, \{p_{\mathrm{Var}}\}) \\
p_{\mathrm{Var}} &= E^0 \to \mathrm{Var}\ S^1 \\
r_{\mathrm{Var}} &= \mathrm{Var} : \mathrm{eval} := \lambda\sigma.\sigma\ S^1.\mathrm{label}
\end{aligned}
$$

図 4.3 はこのモジュールが定義する属性の様子を表現している。

関数適用モジュール $\Delta M_{\mathbf{App}}$ 関数適用の構文を定義するモジュール

$$
\begin{aligned}
\Delta M_{\mathrm{App}} &= (\phi, \phi, \{r_{\mathrm{App}}\}, \phi, \{\mathrm{App}\}, \{p_{\mathrm{App}}\}) \\
p_{\mathrm{App}} &= E^0 \to \mathrm{App}\ E^1\ E^2 \\
r_{\mathrm{App}} &= \mathrm{App} : \mathrm{eval} := \lambda\sigma.(E^1.\mathrm{eval}\ \sigma)(E^2.\mathrm{eval}\ \sigma)
\end{aligned}
$$

図 4.4 はこのモジュールが定義する属性の様子を表現している。

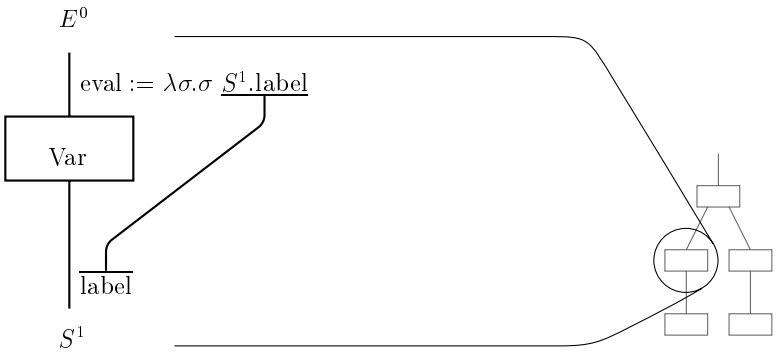


図 4.3: 変数参照モジュール ΔM_{Var}

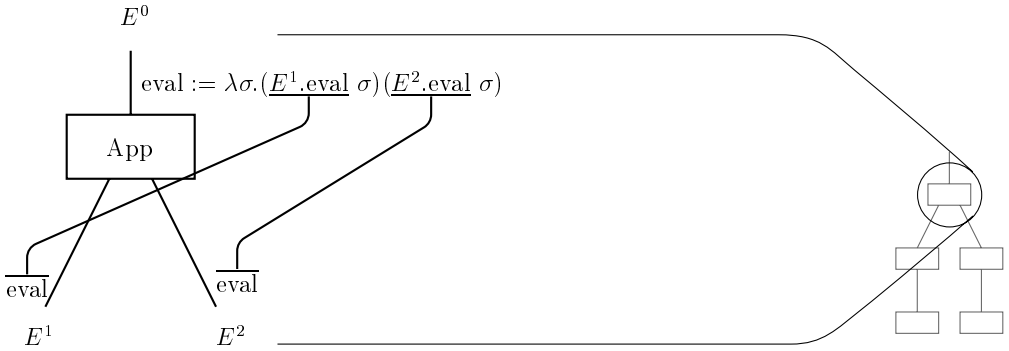


図 4.4: 関数適用モジュール ΔM_{App}

関数抽象モジュール $\Delta M_{\mathbf{Abs}}$ 関数抽象の構文を定義するモジュール

$$
\begin{aligned}
\Delta M_{\text{Abs}} &= (\phi, \phi, \{r_{\text{Abs}}\}, \phi, \{\text{Abs}\}, \{p_{\text{Abs}}\}) \\
p_{\text{Abs}} &= E^0 \to \text{Abs}\ S^1\ E^2 \\
r_{\text{Abs}} &= \text{Abs} : \text{eval} := \lambda\sigma.\lambda v.(E^2.\text{eval}\ \sigma[S^1.\text{label} \leftarrow v])
\end{aligned}
$$

図 4.5 はこのモジュールが定義する属性を表現している。

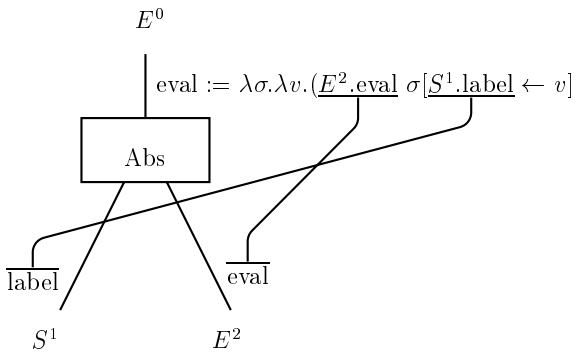


図 4.5: 関数抽象モジュール ΔM_{Abs}

4.1.3 メタレベルの拡張

本節ではメタレベル拡張モジュールについて、任意変数の参照を許す拡張モジュール (ΔM_{AVar})、自由変数を求めるもの (ΔM_{Fv}) と、局所変数を実現するもの (ΔM_{Let}) という 3 つの例をあげて説明する。これらの拡張モジュールを 図 4.6 のように基本的なメタレベルに導入することによってベースレベルで拡張機能が利用できるようになる。なお、ΔM_{Fv} と ΔM_{AVar}、ΔM_{Fv} と ΔM_{Let} は衝突を起こし、同時に利用する場合には制限が生じる。このことについては 4.1.4 節 で述べる。

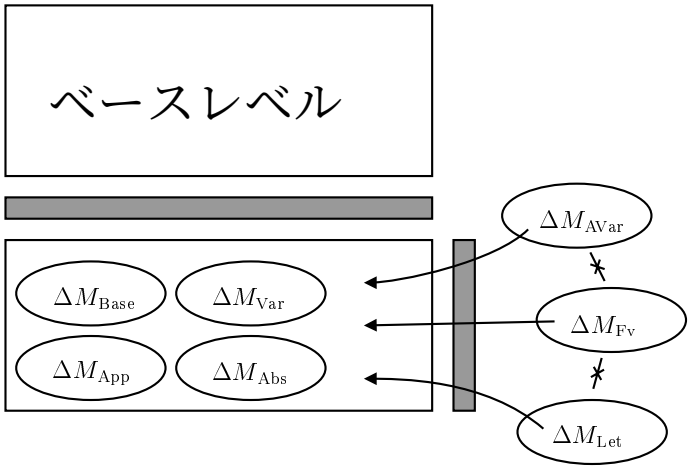


図 4.6: メタレベルへの拡張モジュールの導入

任意変数の参照

任意変数の参照式を引数としてとり、その式を評価した結果を変数とみなして変数参照を行なう構文を考える。このような任意の変数を参照する構文を拡張するモジュールは次のように記述できる。

$$
\begin{aligned}
\Delta M_{\mathrm{AVar}} &= (\phi, \phi, \{r_{\mathrm{AVar}}\}, \phi, \{\mathrm{AVar}\}, \{p_{\mathrm{AVar}}\}) \\
p_{\mathrm{AVar}} &= E^0 \rightarrow \mathrm{AVar}\ E^1 \\
r_{\mathrm{AVar}} &= \mathrm{AVar} : \mathrm{eval} := \lambda\sigma.\sigma\ (E^1.\mathrm{eval}\ \sigma)
\end{aligned}
$$

この拡張を実現するためには環境を自由にアクセスする必要があるため、ベースレベルで実装することは不可能であるが、メタレベルでは環境に自由にアクセス

できるため実現できる。つまり、基本的な λ 計算では実現不可能な機能をモジュールとして拡張している。

また、このモジュールの属性定義を 図 4.7 に図示する。この定義は変数参照モジュールで変数の名前として使っていたシンボルの label 属性を引数の式の評価結果に入れ換えたものである。

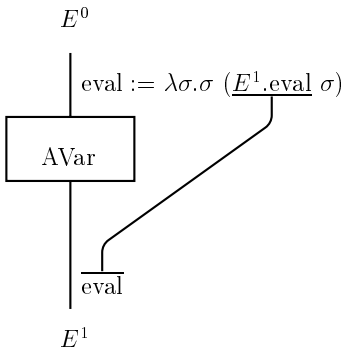


図 4.7: 任意変数参照モジュール ΔM_{AVar}

自由変数を求める

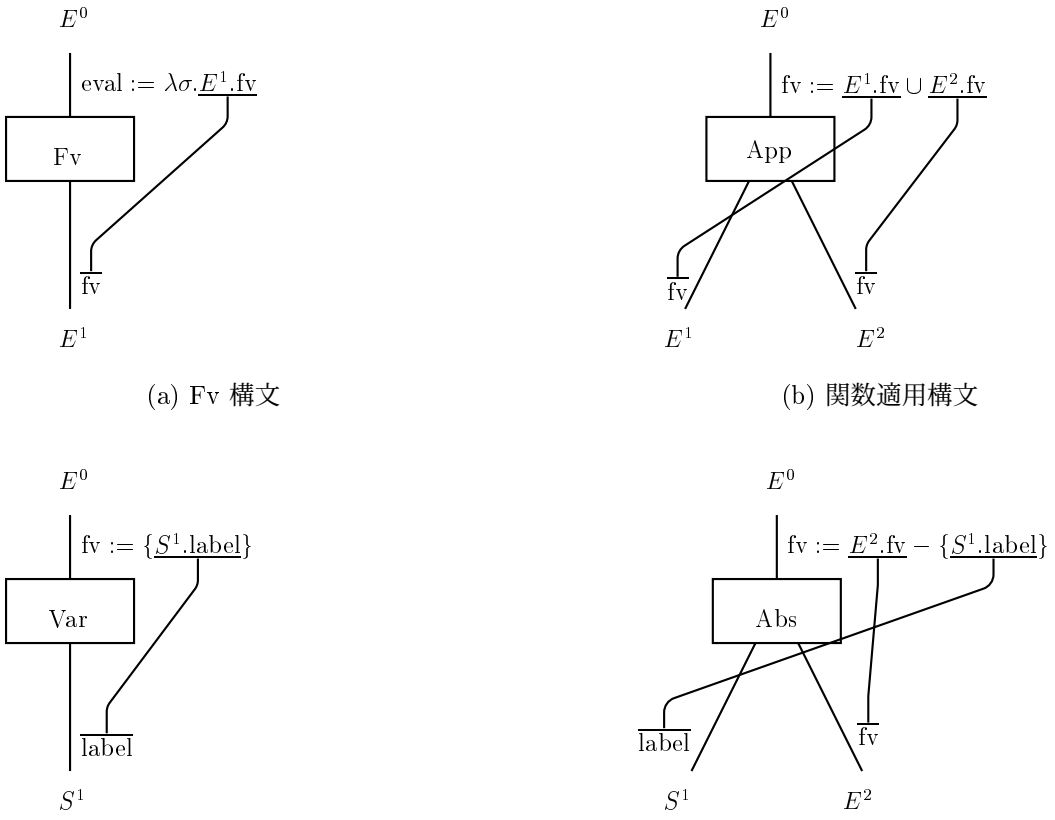
自由変数を求める式を引数としてとり、評価結果としてその式の自由変数の集合を返すという構文を考える。このような自由変数を求める構文を拡張するモジュールは次のように記述できる。

$$
\begin{aligned}
\Delta M_{\text{Fv}} &= (\{\text{fv}\}, \phi, \{r_{\text{Fv}}, r_{\text{FvVar}}, r_{\text{FvApp}}, r_{\text{FvAbs}}\}, \phi, \{\text{Fv}\}, \{p_{\text{Fv}}\}) \\
p_{\text{Fv}} &= E^0 \to \text{Fv}\ E^1 \\
r_{\text{Fv}} &= \text{Fv} : \text{eval} := \lambda\sigma.E^1.\text{fv} \\
r_{\text{FvVar}} &= \text{Var} : \text{fv} := \{S^1.\text{label}\} \\
r_{\text{FvApp}} &= \text{App} : \text{fv} := E^1.\text{fv} \cup E^2.\text{fv} \\
r_{\text{FvAbs}} &= \text{Abs} : \text{fv} := E^2.\text{fv} - \{S^1.\text{label}\}
\end{aligned}
$$

4 つの評価規則のうち $r_{\text{FvVar}}, r_{\text{FvApp}}, r_{\text{FvAbs}}$ が自由変数を求めるものであり、r_{Fv} が求めた自由変数を式の値として利用できるようにしているものである。ここでは他のモジュールで定義した構文 (Var, App, Abs) に対する評価規則 $(r_{\text{FvVar}}, r_{\text{FvApp}}, r_{\text{FvAbs}})$ を追加しているが、これは、構文が定義された時点は考えられていなかった意味

を (既存の意味に悪影響を与えることなく) 追加できるということを示している。また、この例では Fv 式の値が定数となるが、その値を属性に記録しておくことによって、ベースレベルの実行時に Fv 式に到達するたびに再計算が起こることを防いでいる [16]。

また、このモジュールの属性定義を 図 4.8 に図示する。



(a) Fv 構文　(b) 関数適用構文

(c) 変数参照構文　(d) 関数抽象構文

図 4.8: 自由変数を求めるモジュール ΔM_{Fv}

Let

非終端属性を使う拡張として局所変数を定義する Let 式の拡張を考える。Let 式は関数適用と関数抽象に展開可能であるため、ここでは関数適用と関数抽象の

定義を再利用して Let 式を定義する。

$$
\begin{aligned}
\Delta M_{\mathrm{Let}} &= (\phi, \{L\}, \{r_{\mathrm{Let}}, q_{\mathrm{Let}}\}, \phi, \{\mathrm{Let}\}, \{p_{\mathrm{Let}}\}) \\
p_{\mathrm{Let}} &= E^0 \to \mathrm{Let}\ S^1\ E^2\ E^3 \\
r_{\mathrm{Let}} &= \mathrm{Let} : \mathrm{eval} := L.\mathrm{eval} \\
q_{\mathrm{Let}} &= \mathrm{Let} : L := \mathrm{App}\ (\mathrm{Abs}\ S^1\ E^3)\ E^2
\end{aligned}
$$

この定義では、Let 式は非終端属性 L をもち、q_{Let} によって L に Let 式を関数適用と関数抽象に展開した構文木が接ぎ木される。そして、r_{Let} によって L 上の計算結果を Let 式自体の計算結果としている。ここで、この定義の中には関数適用と関数抽象の評価規則は含まれていない。つまり、それらの規則を再利用して Let 式を定義している。

また、このモジュールの属性定義を 図 4.9 に図示する。

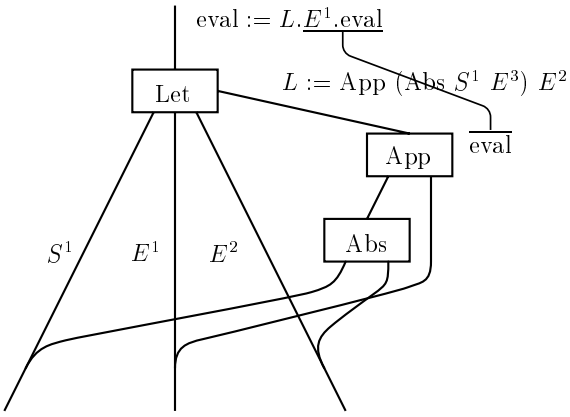


図 4.9: 局所変数モジュール ΔM_{Let}

4.1.4 メタレベルモジュールの衝突

前節のメタレベルモジュール 3 種を同時に適用した場合、必要な属性が定義されていない状況が現れる。

それぞれの構成子について属性がどのモジュールで定義されているかを示したものが表 4.1 である。表から、eval 属性はすべての構成子に対して定義されているが、fv 属性は定義されていない構成子が存在することがわかる。この結果、モジュールをすべて同時に適用した場合、抽象構文木中の AVar 節、Let 節、Fv 節

には属性 fv が定義されず、Fv 節の E^1 内にそれらが存在する場合には自由変数を求めることができない。

この衝突はモジュールを加えたことにより節の種類が増えて、「すべての節に fv 属性が定義されている」という性質が成り立たなくなり、必要な属性が定義されていない場合が生じることが原因である。したがって、必要な属性を定義することによって衝突を解決できる。たとえば、Let と Fv の組み合わせについては次の評価規則をもつモジュールを追加することによって衝突を解決できる。

$$\text{Let} : \text{fv} := L.\text{fv}$$

同様に AVar と Fv の組み合わせについても評価規則を追加すれば解決できるが、AVar に対する自由変数の定義は自明ではないという問題がある。したがって、AVar に対する fv の定義を追加して Fv 内に AVar を許容するか、定義を追加せずにエラーとするかは、メタレベルモジュールの用途に依存する。

表 4.1: メタレベルモジュールの定義する属性

	eval	fv
Var	ΔM_{Var}	ΔM_{Fv}
App	ΔM_{App}	ΔM_{Fv}
Abs	ΔM_{Abs}	ΔM_{Fv}
Fv	ΔM_{Fv}	
Let	ΔM_{Let}	
AVar	ΔM_{AVar}	

4.2 LR 構文解析器

本節では LR 構文解析器をメタレベルで実現することを述べる。構文解析器はトークンの列を入力として受け取り、それを規定された文法にしたがって解釈して構文木を生成する (構文解析する) プログラムである。ここで、文法を BNF などの文法記述言語で記述するとすれば、構文解析器はその言語による文法記述を解釈し、その意味にしたがって入力から構文木を生成するとみなせる。つまり、文法

記述をベースレベルとし、メタレベルをその文法記述にしたがって構文解析するプログラムとみなすことによって、構文解析器は 図 4.10 のようなメタレベルアーキテクチャをなしているとみなせる。

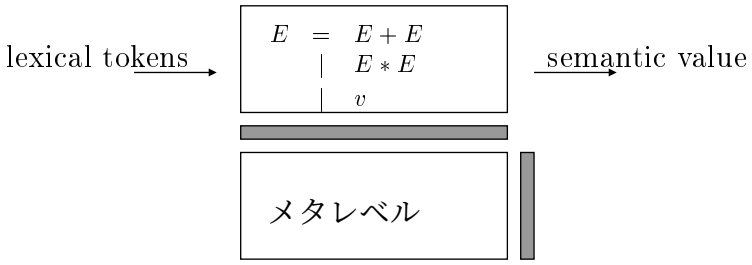


図 4.10: LR 構文解析器のメタレベル

なお、LR 構文解析器生成系は実際に構文解析をするかわりにそのためのプログラムを生成する、つまり文法記述をコンパイルする。しかし、本研究では意味を直接的に定義するために、直接構文解析することを考える。

LR 構文解析の基本的な動作は次の 4 段階にわかれる。

1. 文法記述から item の集合を求める。(ΔM_{Item})
2. item の集合から、LR オートマトンを求める。(ΔM_{At})
3. LR オートマトンの各状態について衝突を解決する。(ΔM_{Res})
4. LR オートマトントークン列を適用して構文解析を行なう[1]。($\Delta M_{\mathrm{Parse}}$)

上記の 4 段階のモジュールに加え、文法記述自体の文法を定義するモジュール ΔM_{BNF} により、LR 構文解析器のメタレベルモジュール ΔM_{LR} およびメタレベル M_{LR} を次のように定義する。

$$\begin{aligned} \Delta M_{\mathrm{LR}} &= \Delta M_{\mathrm{BNF}} + \Delta M_{\mathrm{Item}} + \Delta M_{\mathrm{At}} + \Delta M_{\mathrm{Res}} + \Delta M_{\mathrm{Parse}} \\ M_{\mathrm{LR}} &= (\Delta M_{\mathrm{LR}}, g) \end{aligned}$$

ここで、g は ΔM_{BNF} で定義される開始記号である。

たとえば、次のような文法記述に対して生成される item の集合と (衝突を算術

[1] LR 構文解析器生成系では構文解析器を生成する。

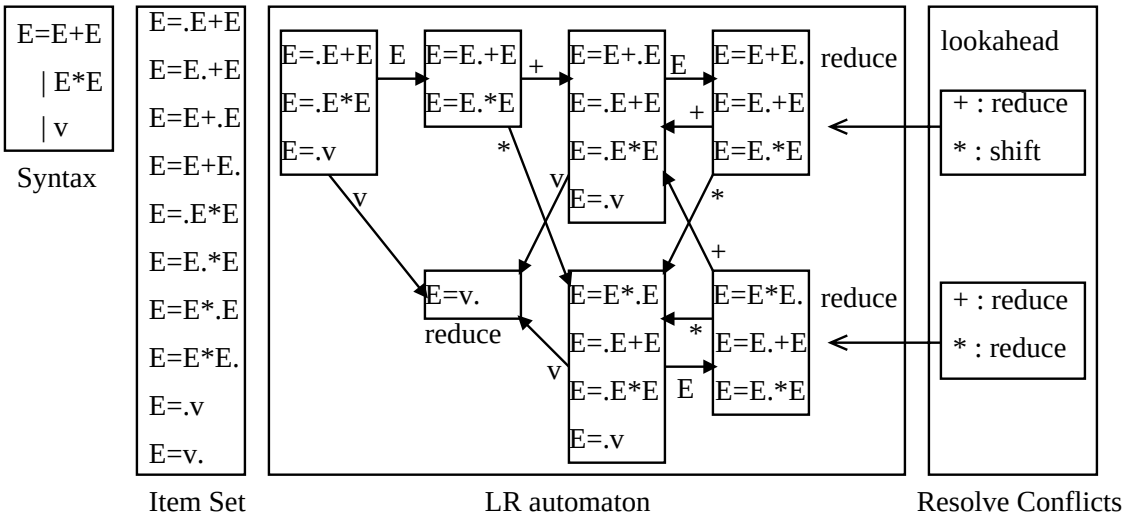


図 4.11: LR 構文解析器の生成例

式に適切なように解決した)LR オートマトンはたとえば 図 4.11 のようになる。

$$
\begin{aligned}
E &= E + E \\
&| \quad E * E \\
&| \quad v
\end{aligned}
$$

4.2.1 評価関数の記法

ここでは 4.1 節 と同様に λ 計算の記法を使って評価関数を記述する。ただし、データ構造として組およびリストを使用する。

組の操作を表現するための記法は次のとおりとする。ここで、e_i は組の要素、t は組を表す。

タプルの生成 $(e_1, \ldots, e_n)$ と記述して n 要素の組を生成を表す。

第 n 要素の取り出し $\#n\ t$ と記述して t の第 n 要素を表す。

第 1 要素の取り出し first t と記述して t の第 1 要素を表す。

第 2 要素の取り出し second t と記述して t の第 2 要素を表す。

リストを操作するためのプリミティブは次のとおりである。ここで、e はリストの要素、l はリストを表す。

空リスト [] と記述して空リストを表す。

リストの生成 $e : l$ と記述して先頭要素が e, それ以降が l のリストを表す。

リストの長さ $|l|$ と記述して l の長さを表す。

先頭要素の取り出し head l と記述して l の先頭要素を表す。

先頭要素を取り除いたリストの取り出し tail l と記述して l から先頭要素を取り除いたリストを表す。

先頭の k 要素の取り出し headk l k と記述して l の先頭の k 要素からなるリストを表す。

先頭の k 要素を取り除いたリストの取り出し tailk l k と記述して l の先頭の k 要素を取り除いたリストを表す。

4.2.2 文法記述の文法を定義する

文法記述は BNF で行なうこととし、その文法をメタレベルモジュールとしたものが次の ΔM_{BNF} である。ここで、メタレベルが扱う文法 (BNF 自体の文法) とベースレベルが扱う文法 (BNF で記述された特定の文法) の混同による混乱を避け

るため、ベースレベルが扱う文法では非終端記号に小文字を使って区別する。

$$
\begin{aligned}
\Delta M_{\mathrm{BNF}} &= (\phi, \phi, \phi, \Delta N_{\mathrm{BNF}}, \Delta T_{\mathrm{BNF}}, \Delta P_{\mathrm{BNF}}) \\
\Delta N_{\mathrm{BNF}} &= \{g, ps, p, ss, s, n, t, a\} \\
\Delta T_{\mathrm{BNF}} &= \{\text{Gram, NonT, Term, Prod0, Prods, Prod, Sym0, Syms}\} \\
\Delta P_{\mathrm{BNF}} &= \{g \rightarrow \text{Gram } ps, \\
&\quad\ ps \rightarrow \text{Prod0}, \\
&\quad\ ps \rightarrow \text{Prods } p\ ps, \\
&\quad\ p \rightarrow \text{Prod } n\ ss\ a, \\
&\quad\ ss \rightarrow \text{Sym0}, \\
&\quad\ ss \rightarrow \text{Syms } s\ ss, \\
&\quad\ s \rightarrow \text{NonT } n, \\
&\quad\ s \rightarrow \text{Term } t\}
\end{aligned}
$$

ここで、g は文法記述の開始記号、ps は構文の並び、p は (アクションが付随した) 構文、ss はシンボルの並び、s はシンボル、n は非終端記号、t は終端記号、a はアクションである。また、n と t はその記号の識別子を label という属性に持つとする。

4.2.3 item の集合を求める

item は導出規則の右辺のどこかにどこまで入力を読んだかを示す印 (.) を挿入したものであり、パーザの入力が構文のどこまで進んだかを示すものである。これは BNF による構文定義の各部と単純に対応がとれるので次の ΔM_{Item} のように定義できる。

$$
\Delta M_{\mathrm{Item}} = (\{\text{items, lhs, left, right}\}, \phi, R, \phi, \phi, \phi)
$$

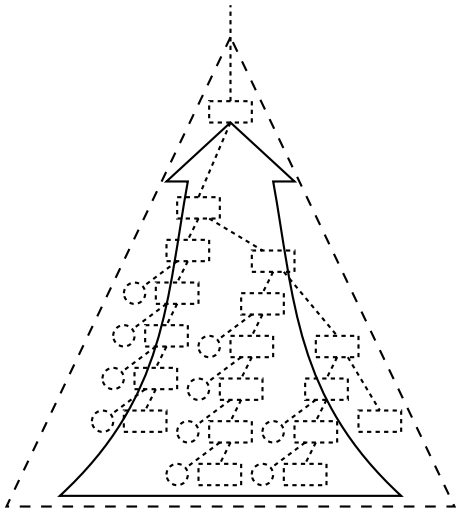


図 4.12: item の集合を求める

ここで、属性 items が item の集合であり、lhs, left, right は補助的な属性である。また、R は次の評価規則からなる集合である。

$$
\begin{aligned}
\text{Gram} &: \text{items} := ps.\text{items} \\
\text{Prods} &: \text{items} := p.\text{items} \cup ps.\text{items} \\
\text{Prod0} &: \text{items} := \phi \\
\text{Prod} &: \text{items} := ss.\text{items} \\
\text{Prod} &: ss.\text{lhs} := n.\text{label} \\
\text{Prod} &: ss.\text{left} := [] \\
\text{Syms} &: ss.\text{lhs} := \text{lhs} \\
\text{Syms} &: ss.\text{left} := \text{left} +\!\!+ [s.\text{label}] \\
\text{Syms} &: \text{right} := [s.\text{label}] +\!\!+ ss.\text{right} \\
\text{Syms} &: \text{items} := ss.\text{items} \cup \{\text{lhs} = \text{left.right}\} \\
\text{Sym0} &: \text{right} := [] \\
\text{Sym0} &: \text{items} := \{\text{lhs} = \text{left.}\}
\end{aligned}
$$

ただし [...] はリスト、$+\!\!+$ はリストの連結演算子である。

このような評価規則により、図 4.12 のように構文定義の各部分で個々の item を

生成し、それらを最上部の Gram 節に集めることができる。

4.2.4 LR オートマトンを求める

LR-オートマトンは状態が item の集合の部分集合に対応するようなプッシュダウンオートマトンである。したがって、初期状態から到達可能なすべての状態について対応する部分集合を求めることによってオートマトンが生成される。このアルゴリズムは subset-construction と呼ばれ、生成される状態数は item の数に対して最大で指数関数的に増加する。このため状態を構文木 (BNF による構文定義) の属性として実現することはできない[2]。したがってこの段階の処理は図 4.13 のように基本的に根の評価規則内で行ない、結果を構文木として動的に生成する。

$$\Delta M_{\mathrm{At}} \quad = \quad (\phi, \Delta B_{\mathrm{At}}, R, \Delta N_{\mathrm{At}}, \phi, \Delta P_{\mathrm{At}})$$

ここで R は次の評価規則からなる[3]。

Gram : A := (items からオートマトンを求める)

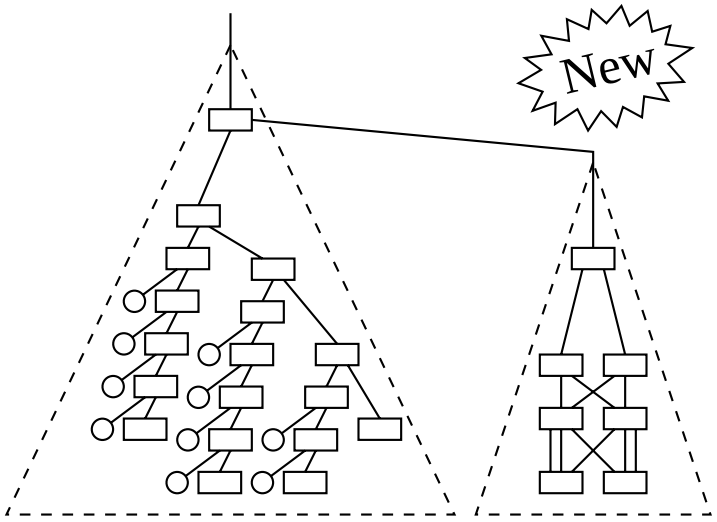


図 4.13: LR オートマトンを求める

[2]属性の数は構文木の大きさに対してたかだか線形にしかならないので、指数関数的に増加するものを属性に直接対応づけることはできない。

[3]この規則は複雑であるため、ここでは内容の定義には触れない。詳細については 付録 A.5, A.4 を参照のこと。

また ΔN_{At}, ΔP_{At} はオートマトンを表現するための非終端記号と導出規則の集合であり、次のように定義される。

$$
\begin{aligned}
\Delta B_{\mathrm{At}} &= \{A\} \\
\Delta N_{\mathrm{At}} &= \{st, bs, b\} \\
\Delta P_{\mathrm{At}} &= \{st \rightarrow \text{State } bs, \\
&\qquad bs \rightarrow \text{Branch0}, \\
&\qquad bs \rightarrow \text{Branches } b\ bs, \\
&\qquad b \rightarrow \text{Shift } s\ st, \\
&\qquad b \rightarrow \text{Reduce } p\}
\end{aligned}
$$

ここで、st は状態、bs は分岐の並び、b は分岐である。分岐は Shift と Reduce があり、一つの状態の分岐に Shift と Reduce が両方とも含まれていれば Shift–Reduce 衝突、Reduce が 2 つ以上含まれていれば Reduce–Reduce 衝突となる。また、オートマトンはサイクルを含む可能性があるが、ここでは属性文法の拡張により構文木にサイクルを含められるという性質を利用して直接構文木として表現している。

4.2.5 衝突を解決する

一般に LR オートマトンにおいて衝突が起こった場合、分岐のいずれかを選んで実行を進めることになる。

このときの選び方で一般的なものには次の規則 [6] がある。

- Shift–Reduce 衝突の場合は Shift を選ぶ。
- Reduce–Reduce 衝突の場合は最長の導出規則に対応するものを選ぶ。

ただし、この方法は必ずしも完全なものではなく、演算子の優先順位を使ったものや、衝突の解決をユーザが指定したコードによって行なうものなど、多種多様な規則がある。しかし、どの解決方法も最終的には構文解析時のある計算状態においてある分岐が選択可能かどうかを判断することに帰着される。

したがって、分岐 b にその分岐を選択できるかどうかを判断するための属性 applicable を定義することによってさまざまな衝突解決法を実現できる。ここで、

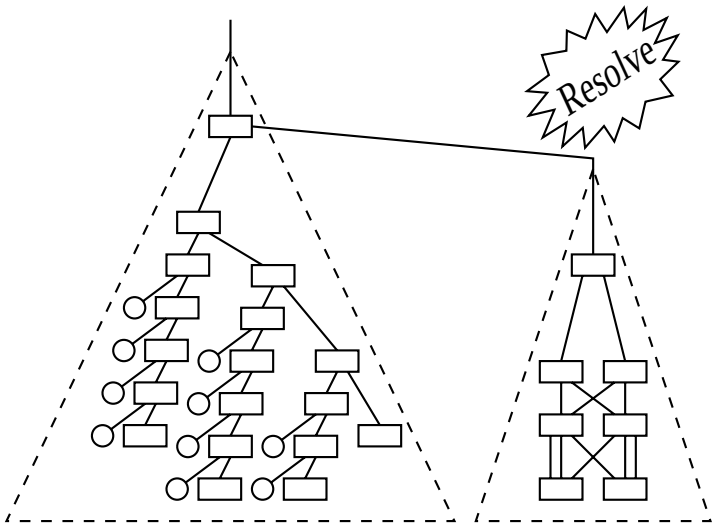


図 4.14: LR オートマトンの衝突を解決する

applicable は引数に計算状態として残りの入力トークン列とスタックをとり、真偽値を返すものとする。次のモジュール ΔM_{Res} はこれを定義するものである。

$$\Delta M_{\mathrm{Res}} \quad = \quad (\{\mathrm{applicable}\}, \phi, R, \phi, \phi, \phi)$$

ここで R は次の評価規則からなる集合である。

$$\begin{aligned}
\mathrm{Shift} \quad &: \quad \mathrm{applicable} := \lambda w.\lambda c.(\mathrm{first}\ (\mathrm{head}\ w)) = s.\mathrm{label} \\
\mathrm{NonT} \quad &: \quad \mathrm{label} := n.\mathrm{label} \\
\mathrm{Term} \quad &: \quad \mathrm{label} := t.\mathrm{label}
\end{aligned}$$

たとえば、もっとも単純な例として LR(0) を考える。この場合 Reduce では先読みを行なわないため、引数の計算状態によらず分岐を選択可能である。この方法は、次のようなメタレベルモジュールで衝突解決規則を定義できる。

$$\Delta M_{\mathrm{LR0Red}} \quad = \quad (\phi, \phi, \{\mathrm{Reduce} : \mathrm{applicable} := \lambda w.\lambda c.\mathrm{true}\}, \phi, \phi, \phi)$$

一般に LR(k) での Reduce では k 個の先読みトークンを使って判断する。したがって、ある Reduce が実行可能な場合の先読みの集合が lookaheads という属性

に定義されているとすれば、次のようなモジュールで衝突解決規則を定義できる。

$$\begin{aligned}
\Delta M_{\mathrm{LRkRed}} &= (\{\text{lookaheads}\}, \phi, \{r_{\mathrm{LRkRed}}\}, \phi, \phi, \phi) \\
r_{\mathrm{LRkRed}} &= \text{Reduce : applicable} := \lambda w.\lambda c.l \in \text{lookaheads} \\
&\qquad \text{where} \quad l = \text{map first (headk } w\ k)
\end{aligned}$$

また、スタックの内容を参照すれば構文解析時に各記号に対応づけられる値 (Semantic Value) を参照でき、その値を使って判断することもできる。

このように衝突の解決のインターフェースと実装を分割してモジュールにすることにより、モジュールの入れ換えによってさまざまな衝突解決の方法を選択することができる。

4.2.6 構文解析を行なう

LR オートマトンをトークン列に適用することによって図 4.15 のように構文解析を行なうことができる。LR オートマトンはプッシュダウンオートマトンであり、トークン列のある場所まで読み込んだ時点の計算状態はオートマトン自体の状態とスタックの対となり、この計算状態の変化として構文解析の意味を定義できる。そこで、オートマトンのそれぞれの状態に対応する節に、スタックを更新して次

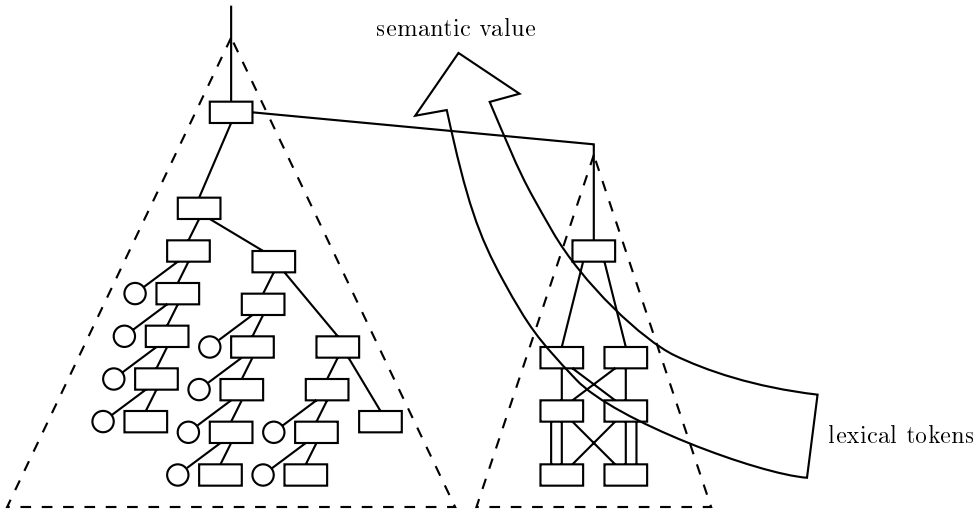


図 4.15: 構文解析を行なう

の状態に遷移することを表現する関数を属性として定義することによって、構文解析の意味を定義する。

$$\Delta M_{\mathrm{Parse}} \quad = \quad (\{\mathrm{parse}\}, \phi, R, \phi, \phi, \phi)$$

ここで、R は次の評価規則からなる。

$$\begin{aligned}
\text{Gram} \quad &: \quad \mathrm{parse} := \lambda w.st.\mathrm{parse}\ (w +\!\!+ [\$])\ [] \\
\text{State} \quad &: \quad \mathrm{parse} := \lambda w.bs.\mathrm{parse}\ w \\
\text{Branch0} \quad &: \quad \mathrm{parse} := \lambda w.(\text{エラー}) \\
\text{Branches} \quad &: \quad \mathrm{parse} := \lambda w.\lambda c. \begin{cases} b.\mathrm{parse}\ w\ c & b.\mathrm{applicable}\ w\ c \\ bs.\mathrm{parse}\ w\ c & \mathrm{otherwise} \end{cases} \\
\text{Shift} \quad &: \quad \mathrm{parse} := \lambda w.\lambda c.s.\mathrm{parse}\ (\mathrm{tail}\ w)\ ((s.\mathrm{parse}, \mathrm{second}\ (\mathrm{head}\ w)) : c) \\
\text{Reduce} \quad &: \quad \mathrm{parse} := \lambda w.\lambda c.(\mathrm{first}\ (\mathrm{head}\ c'))\ ((p.\mathrm{lhs}, v) : w)\ c' \\
&\qquad\qquad \text{where} \\
&\qquad\qquad\quad c' = \mathrm{tailk}\ c\ |p.\mathrm{rhs}| \\
&\qquad\qquad\quad v = p.\mathrm{a}\ (\mathrm{map\ second}\ (\mathrm{headk}\ c\ |p.\mathrm{rhs}|)) \\
\text{Prod} \quad &: \quad \mathrm{lhs} := n.\mathrm{label} \\
\text{Prod} \quad &: \quad \mathrm{rhs} := ss.\mathrm{right}
\end{aligned}$$

4.3 単純な関数型言語と LR 構文解析器の合成

メタレベルモジュールは一般に (名前の衝突を起こさなければ) 合成が可能である。各モジュールそれぞれで完結したモジュール群を合成した場合、開始記号の選び方によって合成された各モジュールと等価なメタレベルを構成することができる。そして、各モジュールの関係を記述するモジュールをさらに加えて合成することにより、各モジュールの機能をすべて利用可能なメタレベルを構成することができる。

単純な関数型言語のメタレベルモジュール ΔM_{L} と LR 構文解析器のメタレベルモジュール ΔM_{LR} は完結したモジュールであり、ΔM_{L} と ΔM_{LR} は名前の衝突

を起こさないため、次のようにしてこれらを合成することができる。

$$\Delta M_{\mathrm{LLR}} = \Delta M_{\mathrm{L}} + \Delta M_{\mathrm{LR}}$$

ここで ΔM_{LLR} を使うと、開始記号の選び方により、M_{L} と M_{LR} と等価なメタレベルを構成できる。具体的には、開始記号として E を選んで構成した $(\Delta M_{\mathrm{LLR}}, E)$ というメタレベルは M_{L} と等価になり、g を選んで構成した $(\Delta M_{\mathrm{LLR}}, g)$ というメタレベルは M_{LR} と等価になる。

そして、次のようなモジュールを ΔM_{LLR} と合成し、式の中に構文解析器を含めることができるようにすると、単純な関数型言語の中から LR 構文解析器の機能を利用できるようになる。

$$\begin{aligned}
\Delta M_{\mathrm{P}} &= (\phi, \phi, \{r_{\mathrm{Parse}}\}, \phi, \{\mathrm{Parse}\}, \{p_{\mathrm{Parse}}\}) \\
p_{\mathrm{Parse}} &= E^0 \to \mathrm{Parse}\ g^1 \\
r_{\mathrm{Parse}} &= \mathrm{Parse} : \mathrm{eval} := \lambda\sigma.\sigma\ g^1.\mathrm{parse}
\end{aligned}$$

また、逆に LR 構文解析器の中から単純な関数型言語を利用するようなモジュールを考えることもできる。

第 5 章

Scheme による実装

本章では Scheme[9] による 3 章、4 章で述べたメタレベルの実装について述べる。

まず 5.1 節 で集合、リストなどを扱う (Scheme 標準ではない) 補助的な関数を導入する。そして、それらの関数を使用し、3 章で述べたメタレベル記述の支援ライブラリについて 5.2 節 で述べる。最後に、4 章 で述べた簡単な関数型言語と LR 構文解析器の実現について 5.3 節 と 5.4 節 で述べる。

なお、本章の記述では繁雑であまり本質的でない部分を省くが、付録 A にはそれらも含んだ実装を載せる。

5.1 補助関数

R5RS[9] に含まれていない集合、リスト、文字列、シンボルを扱う関数として次のものを用意する。

5.1.1 集合を扱うライブラリ

集合を扱う関数として次のような関数を用意する。ここで、集合の要素の等価性は `equal?` で判定されるものとする。また、これらはすべて副作用を持たない関数である。付録 A.6 に実装を示す。

- (`set-empty?` *set*) は *set* が空であるときに `#t`、そうでないときに `#f` を返す。

- (`set-has?` *set elt*) は *set* が *elt* を要素として持つときに `#t`、そうでないときに `#f` を返す。
- (`set-contain?` *set1 set2*) は *set1* が *set2* を包含しているときに `#t`、そうでないときに `#f` を返す。
- (`set-size` *set*) は *set* の要素数を返す。
- (`set` *elt* ...) は *elt* ... を要素とする集合を返す。ここで、特に (`set`) は空集合を返す。
- (`list->set` *list*) は *list* の各要素を要素とする集合を返す。
- (`set-add` *set elt* ...) は *set* に *elt* ... を加えた集合を返す。
- (`set-del` *set elt* ...) は *set* から *elt* ... を取り除いた集合を返す。
- (`set-union` *set* ...) は *set* ... の和集合を返す。
- (`set-subtract` *set set′* ...) は *set* から *set′* ... それぞれの要素を取り除いた集合を返す。
- (`set-intersect` *set set′* ...) は *set set′* ... の共通集合を返す。
- (`set-first` *set*) は *set* から任意の要素をひとつ選んでその要素を返す。
- (`set-rest` *set*) は *set* から (`set-first` *set*) で選ばれる要素を取り除いた集合を返す。

5.1.2 リストを扱うライブラリ

リストを扱う関数として次の関数を用意する。付録 A.8 に実装を示す。

- (`assoc-cps` *obj alist succ-cont fail-cont*) は (`assoc` *obj alist*) と同様の動作をするが、結果の返し方が異なる。*obj* を car とするペアが *alist* 中に見つかった場合、そのペアを引数として *succ-cont* を呼び出してその値を返す。また、見つからなかった場合には引数なしで *fail-cont* を呼び出してその値を返す。

- (`rassoc` *obj alist*) は (`assoc` *obj alist*) と同様な動作をするが、*obj* が比較されるのはペアの car ではなく cdr である。

- (`filter` *proc list*) は *list* の各要素それぞれについて、その要素を引数として *proc* を呼び出し、その結果が `#f` でないものからなるリストを返す。

- (`filter-cps` *proc list cont*) は (`filter` *proc list*) と似た動作をするが、結果の返し方が異なる。`filter-cps` は *proc* が `#f` を返さなかった要素からなるリストと *proc* が `#f` を返した要素からなるリストの 2 つを引数として *cont* を呼び出してその値を返す。

- (`append!` *list* ...) は引数のリストを破壊的に変更して連結し、連結したリストを返す。なお、連結は破壊的に行なわれるため、返り値が空リストでない場合には、引数として与えた最初の (空リストでない) 引数自体が返り値となる。

5.1.3 文字列、シンボルを扱うライブラリ

文字列、シンボルを扱う関数として次の関数を用意する。付録 A.9 に実装を示す。

- (`str-split` *string char*) は *string* を *char* で分割し、分割された部分文字列の要素からなるリストを返す。

- (`str-join` *list char*) は文字列のリストである *list* を *char* を区切りとして連結し、連結された文字列を返す。

- (`sym-split` *symbol char*) は *symbol* の文字列表現に対して `str-split` と同様の処理を行なう。ただし、結果は (文字列ではなく) シンボルのリストとなる。

- (`sym-append` *symbol* ...) は *symbol* ... を文字列として連結し、連結されたシンボルを返す。

5.2 メタレベル記述支援ライブラリ

本節では 3 章 で述べた属性文法的なメタレベルの実装を支援するライブラリについて述べる。このライブラリでは属性つきの抽象構文木を各枝に名前のついた木構造として表現する。つまり、ある節は複数の下位構造を持つことができ、それぞれは名前で同定される。下位構造は抽象構文木の節 (およびその節を根とする部分木) であるか、または属性である。つまり、データ構造としては属性と節は区別されない。以下では節は属性に値として節がはいっている場合と考え、属性は節を包含するものとする。この構造を実現するために、節にはタグつきの連想リストを使用する。タグはその節の構成子を表し、連想リストによって名前のついた属性の集まりを表現する。

また、メタレベルでは属性を参照する必要があるため、その参照を表現する形式が必要である。このライブラリでは $attrpath$ という形式で属性の参照が表現される。$attrpath$ はシンボルのリストであり、その参照が使用される時点で注目している節からの相対的な位置を表現している。ここで、ある節 x に注目しているときに $attrpath$ が記述された場合、次の規則によって参照する対象が定義される。

- もし $attrpath$ が空リストならば、参照する対象は x 自身である。(この場合、x は節である必要はなく、通常の値をもつ属性でもよい。)
- もし $attrpath$ が空リストでないならば、先頭要素を $name$, 先頭要素を取り除いたリストを $attrpath'$ とすると、x 中の $name$ という名前を持つ属性に注目して $attrpath'$ を参照したものが対象である。(この場合、x は節でなければならない。)

特に、その節のタグが $X_0 \to X_1 X_2 \ldots X_n$ という導出規則に対応する構成子 X_1 である場合、3 章 で述べた a, b, $X_i.b$, $b.a$, X_i という属性の参照は 表 5.1 のように表現される。

また、このライブラリは、次の特殊形式[1]によって、構成子と評価規則の定義方法を提供する。このライブラリではメタレベルモジュールは大域的に一つ存在し、新しい構成子や評価規則の定義はモジュールの定義を増やしていくものとする。こ

[1]実際にはマクロとして実現される。

表 5.1: 属性参照の $attrpath$ による表現

属性の参照	$attrpath$	意味
a	(a)	X_0 の属性 a
b	(b)	X_0 の非終端属性 b
$X_i.b$	$(X_i\ a)$	X_i の属性 a
$b.a$	$(b\ a)$	X_0 の非終端属性 b の属性 a を
X_i	(X_i)	X_i を根とする構文木

のため、複数のモジュールは共存できないが、名前の衝突が起きない限りモジュールは安全に合成できるため、これは本質的な問題とはならない。

`(define-tag (`$tagname\ attrpath_1 \ldots attrpath_n$`) (`$attrpath\ proc$`)...)` は $tagname$ という構成子 (関数) を定義する。$tagname$ は n 個の引数をとり、i 番目の実引数の値は生成する節の $attrpath_i$ という属性の初期値に使われる。

($attrpath\ proc$)... はこの構成子に附属する評価規則を定義するものであり、($attrpath\ proc$) という記述それぞれは $tagname$ を定義した後に `(define-attr` $tagname\ attrpath\ proc$) として評価規則を定義したのと同じ効果を持つ。

また、同時に $tagname$`-lazy` という構成子 (マクロ) も定義される。$tagname$`-lazy` も $tagname$ と同様の引数をとるが、引数の評価は属性が最初に参照された時点で行なわれる。

`(define-attr` $tagname\ attrpath\ proc$`)` は $tagname$ という構成子に附属して、$attrpath$ という属性を定義する評価規則を定義する。ここで、評価規則はその評価規則が附属している節 (X_0) を定義することはできないので、$attrpath$ は空リストであってはならない。$proc$ は関数として表現された属性の内容の定義であり、この関数は節を受け取って引数のない関数を返さなければならない。返された引数のない関数は属性が最初に参照された時点で呼び出され、結果の値が属性の値となる。通常、$proc$ は `(lambda (`$node$`) (lambda () ...))` という形で記述される。

また、*proc* では受けとった節を使って属性や節を参照することができる。具体的には `node-evaluate` という関数を用い、(`node-evaluate` *node attrpath*) とすることによって参照する。ただし、簡単のために これを $*node*:*attr*:... と略記できることとする。ここで、*attr*:... は *attrpath* 中のシンボルを : で区切って連結したものである。

これらを定義する準備として、まず属性を保持するセルと節の実装について 5.2.1 節 と 5.2.2 節 で述べる。そして、5.2.3 節 でこれらの実装について述べる。

5.2.1 セル

属性を保持するために**セル**を定義する。属性は単一代入によって定義され、参照時に自動的に値が求まるため、メタレベルモジュールからみると参照の透明性が保たれている。しかし、内部的には次のような状態を遷移する。

1. セルが生成される。
2. セルの値を求める評価規則が設定される。
3. 値が要求され、規則の評価が始まる。
4. 規則の評価が終わり、セルの値が決まる。

これらの状態遷移は次の関数によって起きる。

- (`cell-alloc`) はセルを生成して返す。
- (`cell-set-thunk!` *cell thunk*) は *cell* に *thunk* を評価規則として設定する。ここで *thunk* は引数のない関数で、呼び出されるとセルの値を返すものである。
- (`cell-ref` *cell*) は必要ならば評価規則を評価した上でセルの値を参照する。

状態には 4 種類あり、セルが生成された直後は `uninitialized` であり、評価規則が設定されると `unevaluated` になり、値が要求されると `evaluating` になり、値が求まると `evaluated` となる。

また、セルは unevaluated, evaluated 状態ではそれぞれ評価規則、値を保持しなければならない。そこで、セルの実装にはペアを使い、cdr 部に状態、car 部に評価規則か値を保持する。状態の定義および述語は次のように定義される。

```
(define cell-state-uninitialized 'uninitialized)
(define cell-state-unevaluated 'unevaluated)
(define cell-state-evaluating 'evaluating)
(define cell-state-evaluated '())

(define (cell-uninitialized? cell)
  (eq? (cdr cell) cell-state-uninitialized))

(define (cell-unevaluated? cell)
  (eq? (cdr cell) cell-state-unevaluated))

(define (cell-evaluating? cell)
  (eq? (cdr cell) cell-state-evaluating))

(define (cell-evaluated? cell)
  (eq? (cdr cell) cell-state-evaluated))
```

それぞれの状態を表す値は区別可能であれば何でも良く、ここでは最初の 3 つの状態にはそれぞれの状態を表すシンボルを用い、最後の 1 つには空リストを使っている[2]。

cell-alloc と cell-set-thunk! は次のように定義される。

```
(define (cell-alloc)
  (cons #f cell-state-uninitialized)); car 部の #f は使われない

(define (cell-set-thunk! cell thunk)
  (if (cell-uninitialized? cell); 適切でない状態ならエラー
    (begin
      (set-car! cell thunk)
      (set-cdr! cell cell-state-unevaluated))
    (error "cell is not uninitialized" cell)))
```

cell-alloc は cell-state-uninitialized 状態のセルを生成し、cell-set-thunk! はセルの状態が uninitialized 状態であることを確認した上でセルの状態を unevaluated 状態に遷移させる。

また、(cell-ref *cell*) は次のように定義される。

[2]状態を保持するのに cdr 部を使い、また、evaluated 状態を表すのに空リストを使うのは、値が求まった時点での表現が (*val*) と単純になるためである。

```
(define (cell-ref cell)
  (cond
    ((cell-uninitialized? cell)
      (error "uninitialized cell referenced" cell))
    ((cell-unevaluated? cell)
      (let ((thunk (car cell)))
        (set-cdr! cell cell-state-evaluating)
        (set-car! cell (thunk))
        (set-cdr! cell cell-state-evaluated)
        (car cell)))
    ((cell-evaluating? cell)
      (error "evaluating cell referenced" cell))
    ((cell-evaluated? cell)
      (car cell))
    (else
      (error "unknown cell status" cell))))
```

cell-ref の動作はセルの状態によって次のように変化する。

- uninitialized 状態の場合には、評価規則が存在せず、値を求めることが不可能なためエラーとする。
- unevaluated 状態の場合には、評価規則を起動して値を求め、その結果を返す。ただし、評価規則を起動する前にセルを evaluating 状態に遷移させ、評価規則が終了した後に evaluated 状態に遷移させて求めた値を記録する。
- evaluating 状態の場合には、評価規則が起動中であるということであり、同一の属性に対して評価規則を複数回起動することはできないため、エラーとする。ここで評価規則を複数回起動することができないのは、属性が単一代入されるものであるため、値を計算するのは一回しか許されないためである。また、このことは、あるセルの評価規則が直接・間接を問わずそのセル自身の値に依存してはならないということも意味する。
- evaluated 状態の場合には、記録されている値を返す。

5.2.2 節

抽象構文木の節は任意の数の属性を持つ。そこで、構成子をタグとして持ち、属性を属性名とセルの連想リストとして実装する。節を扱う関数は次の通りである。

- (node-alloc *tag*) は *tag* という構成子の節を生成して返す。
- (node->tag *node*) は *node* の構成子を返す。
- (node->attrs *node*) は *node* の属性の連想リストを返す。
- (node-attr-cell *node attr*) は *node* の属性 *attr* を表現するセルを返す。
- (node-evaluate *node attrpath*) は *node* の属性を参照してその値を返す。
- (make-node *tag args* ...) は *tag* という構成子の節を (node-alloc で) 生成し、*tag* の定義で与えられた属性を *args* ... で初期化する。

節の実装にはリストを使い、先頭要素に構成子、それ以降に連想リストを保持する。ただし、節の生成時には連想リストは空であり、node-attr-cell によってセルが要求された時点で必要に応じて属性名とセルの対が破壊的に追加される。追加されたセルは cell-state-uninitialized 状態であり、値を要求する前に評価規則を設定しなければならない。

節を扱う関数は次のように定義される。

```
(define (node-alloc tag)
  (cons tag '()))

(define node->tag car)
(define node->attrs cdr)

(define (node-attr-cell node attr)
  (let ((p (assq attr (node->attrs node))))
    (if p
      (cdr p)
      (let ((p (cons attr (cell-alloc))))
        (append! node (list p))
        (cdr p)))))

(define (node-attr+-cell node path)
  (let loop ((node node) (attr (car path)) (path (cdr path)))
    (let ((cell (node-attr-cell node attr)))
      (if (null? path)
        cell
        (loop (cell-ref cell) (car path) (cdr path))))))

(define (node-evaluate node path)
  (if (null? path)
```

```
    node
    (cell-ref (node-attr+-cell node path))))

(define (tag-attr-defs tagname . args)
  (let ((p (assq tagname taginfo)))
    (append
      (apply
        (cadr p)
        (map (lambda (val) (lambda (node) (lambda () val))) args))
      (cddr p))))

(define (make-node-internal tag-attr-defs tag . args)
  (let ((node (node-alloc tag)))
    (let loop ((attr-defs (apply tag-attr-defs tag args)))
      (if (null? attr-defs)
        node
        (let ((attr-def (car attr-defs)))
          (cell-set-thunk!
            (node-attr+-cell node (car attr-def))
            ((cdr attr-def) node))
          (loop (cdr attr-defs)))))))

(define (make-node tag . args)
  (apply make-node-internal tag-attr-defs tag args))
```

node-alloc, node->tag, node->attrs は節 (構成子と連想リストの対) をペアを使って実現する関数である。

node-attr-cell は節からセルを取り出す関数である。このとき、節に要求された名前のセルが存在しない場合、append!を使って破壊的に連想リストを拡張し、新しいセルを要求された名前で追加する。この新しいセルは cell-alloc で生成されるため、uninitialized 状態となる。

node-attr+-cell は節 node から path にしたがって再帰的に節をたどり、セルを取り出す関数である。ここで、node 自身はセルではないため node を値として持つセルを返すことはできない。したがって、path の長さは 1 以上でなければならない。この関数は再帰的に節をたどるため、途中の節はセルを cell-ref によって参照され、evaluated 状態となるが、最終的に得られるセルに対しては cell-ref は呼び出されず、セルの状態は変化しない。

node-evaluate は節 node から path にしたがって再帰的に節をたどり、属性を取り出す関数である。この関数は node-attr+-cell を呼び出して、結果のセルを cell-ref で参照することによって実現されている。ただし、path が空リスト

である場合には、node-attr+-cell は呼び出されず、即座に node 自身が返り値となる。

make-node とその補助関数 make-node-internal, tag-attr-defs は node-alloc を使って節を生成し、構成子の定義にしたがってセルを付加して評価規則を設定する。ここで、tag-attr-defs は make-node の引数から属性の定義用の関数に変換する関数である。この関数は taginfo という大域変数を参照しているが、これは define-tag と define-attr によって保守される変数であり、構成子の情報を保持している。

5.2.3 メタレベル定義

define-tag, define-attr は次のように定義される。これらは、make-node などのラッパーの定義へ展開されるマクロである。また、define-tag 内の評価規則の定義は省略記法であり、tagname が補完されて define-attr に展開される。

```
(define taginfo '())

(define (define-tag-func form env)
  (let* ((tagname (caadr form))
         (tagname-lazy-func (sym-append tagname '-lazy-func))
         (tagname-lazy (sym-append tagname '-lazy))
         (attrpaths (cdadr form))
         (attrdefs (cddr form))
         (attrpathsyms (map (lambda (syms) (sym-join syms #\:)) attrpaths)))
    `(begin
      (define (,tagname ,@attrpathsyms)
        (make-node ',tagname ,@attrpathsyms))
      (define (,tagname-lazy-func form env)
        (cons 'make-node-lazy (cons '',tagname (cdr form))))
      (define ,tagname-lazy
        (procedure->memoizing-macro ,tagname-lazy-func))
      (define taginfo
        (cons
          (list ',tagname (lambda vals (map cons ',attrpaths vals)))
          taginfo))
      ,@(map (lambda (attrdef) `(define-attr ,tagname . ,attrdef)) attrdefs)
      )))

(define define-tag (procedure->memoizing-macro define-tag-func))

(define (define-attr-func form env)
```

```
  (let ((tagname (cadr form))
        (attrpath (caddr form))
        (proc (cadddr form)))
    `(let ((p (assq ',tagname taginfo)))
       (append! p (list (cons ',attrpath ,proc))))))

(define define-attr (procedure->memoizing-macro define-attr-func))
```

大域変数 taginfo は構成子の情報を保持するものである。taginfo には define-tag によって新しい構成子が導入されたことが記録され、define-attr によって新しい評価規則が導入されたことが記録される。

define-tag は構成子の名前の関数をその構成子の節を生成する関数として定義する。define-attr は評価規則を taginfo に記録する。記録された評価規則は make-node によって (tag-attr-defs を通して) 使われて、実際の節中のセルに設定され、そのセルが参照された時点で実行される。

5.3 単純な関数型言語

4.1 節 で述べた単純な関数型言語は次のようにして定義される。ただし、このライブラリでは宣言なしで属性を使用できるため属性の存在を宣言するベースモジュールは省略する。また、4.1 節 では述べなかった定数を記述する構文も定義する。

5.3.1 変数参照モジュール

変数参照は次のようにして定義される。ここで、env-ref は環境と識別子を引数としてとり、その識別子に対応する値を返す関数とする。

```
(define-tag (lambda-var (lbl))
  ((eval)
    (lambda (node)
      (lambda ()
        (let ((lbl $node:lbl))
          (lambda (env)
            (env-ref env lbl)))))))
```

この定義は lambda-var という構成子を定義し、その構成子に属する eval という

属性を定義する。5.2節で述べた通り、2行目以降の `((eval) ...)` は `define-attr` による属性定義の省略記法であり、省略せずに `define-attr` を用いると次のように記述される。

```
(define-tag (lambda-var (lbl)))

(define-attr lambda-var (eval)
  (lambda (node)
    (lambda ()
      (let ((lbl $node:lbl))
        (lambda (env)
          (env-ref env lbl))))))
```

この定義では最初の `define-tag` によって `lambda-var` という構成子が定義される。この構成子は引数を 1 つとり、構成子を呼び出して節を生成したときに、その引数を初期値として `lbl` という属性が作られる。なお、`(lbl)` というのは *attrpath* であり、一般には (長さ 1 以上の) 任意の *attrpath* が利用できるが、ここでは長さ 1 のものを使うことにより、`lambda-var` で生成する節自体についた属性を定義している。

また、次の `define-attr` によって `lambda-var` という構成子に属する `(eval)` という属性が定義される。この属性は初期値を持たず、値がはじめて要求された時点で評価が行なわれて値が求められる。その値を求める部分は `(let ((lbl $node:lbl)) (lambda (env) (env-ref env lbl)))` である。値が要求された時点で `$node:lbl` により、変数の名前 (Var 節の `lbl` という属性) が取得され、

環境が与えられた時点 (ベースレベルの実行された時点) で、与えられた環境 `env` からその名前の変数の値を取得する。

5.3.2 関数適用モジュール

関数適用は次のようにして定義される。

```
(define-tag (lambda-app (fun) (arg))
  ((eval)
    (lambda (node)
      (lambda ()
        (let ((fun-eval $node:fun:eval)
```

```
      (arg-eval $node:arg:eval))
  (lambda (env)
    ((fun-eval env)
     (arg-eval env))))))))
```

この定義では、lambda-app という 2 引数の構成子が定義される。それらの引数は関数適用の関数部分と引数部分を表す節でなければならず、与えられた引数は fun と arg という属性に格納される。lambda-app の eval 属性はそれらの節の eval 属性を$node:fun:eval と $node:arg:eval として取得し、環境が与えられ、ベースレベル実行が行なわれると、それらに環境を渡して実行し、得られた結果を関数適用する。

5.3.3 関数抽象モジュール

関数抽象は次のようにして定義される。ここで、env-extend は環境と識別子と値を引数としてとり、その識別子からその値への束縛を拡張した環境を返す関数とする。

```
(define-tag (lambda-abs (var) (body))
  ((eval)
    (lambda (node)
      (lambda ()
        (let ((var $node:var)
              (body-eval $node:body:eval))
          (lambda (env)
            (lambda (val)
              (body-eval (env-extend env var val)))))))))
```

この評価規則の定義では、lambda-abs という 2 引数の構成子が定義される。それらの引数は関数適抽象の仮引数と本体でなければならず、与えられた引数は var と body という属性に格納される。eval 属性は節が生成されると $node:var と $node:body:eval によって仮引数と本体のインタプリタ関数を取得し、ベースレベル実行が行なわれると、与えられた環境によって無名関数を生成する。そして、無名関数に実引数が渡されて呼ばれたときには仮引数と実引数で環境を拡張してインタプリタ関数を呼び出すことにより、関数本体を実行する。

5.3.4 定数モジュール

定数構文は次のようにして定義される。

```
(define-tag (lambda-con (val))
  ((eval)
    (lambda (node)
      (lambda ()
        (let ((val $node:val))
          (lambda (env)
            val))))))
```

この定義は変数参照とよく似ているが、(ベースレベルの実行時に) 環境を参照せずに節の生成時に与えられた値を直接返すことによって定数を実現している。

5.3.5 任意変数の参照モジュール

任意変数の参照は次のようにして定義される。

```
(define-tag (lambda-anyvar (lbl-exp))
  ((eval)
    (lambda (node)
      (lambda ()
        (let ((lbl-exp-eval $node:lbl-exp:eval))
          (lambda (env)
            (env-ref env (lbl-exp-eval env))))))))
```

この定義では `lambda-anyvar` という 1 引数の構成子が定義される。その引数は評価すると変数の名前を返すような式を表現する節でなければならない。また、`eval` 属性によるベースレベルの実行時には `$node:lbl-exp:eval` を呼び出すことによって変数の名前が取得され、その名前で変数参照が行なわれる。

5.3.6 Let モジュール

Let は次のようにして定義される。

```
(define-tag (lambda-let (var) (exp) (body))
  ((expanded)
    (lambda (node)
```

```
      (lambda ()
        (let ((var $node:var)
              (exp $node:exp)
              (body $node:body))
          (lambda-app (lambda-abs var body) exp)))))
  ((eval)
    (lambda (node)
      (lambda ()
        $node:expanded:eval))))
```

この定義では lambda-let という 3 引数の構成子が定義される。その引数は変数の名前、評価すると束縛される値となる式、本体の式でなければならず、節が生成されるとそれぞれ var, exp, body という属性に格納される。

この定義では expanded と eval という 2 つの属性を定義している。expanded は 4.1 節 で述べた非終端属性 L に対応するものであり、lambda-app と lambda-abs という構成子を使って抽象構文木を生成する定義となっている。そして、eval は expanded の eval をそのまま使うよう定義されている。

5.3.7 自由変数モジュール

自由変数を求める構文は次のようにして定義される。

```
(define-tag (lambda-fv (exp))
  ((fv)
    (lambda (node)
      (lambda ()
        (set))))
  ((eval)
    (lambda (node)
      (lambda ()
        (let ((fv $node:exp:fv))
          (lambda (env) fv))))))

(define-attr lambda-con (fv)
  (lambda (node)
    (lambda ()
      (set))))

(define-attr lambda-var (fv)
  (lambda (node)
    (lambda ()
      (set $node:lbl))))
```

```
(define-attr lambda-app (fv)
  (lambda (node)
    (lambda () (set-union $node:fun:fv $node:arg:fv))))

(define-attr lambda-abs (fv)
  (lambda (node)
    (lambda () (set-del $node:body:fv $node:var))))
```

この定義では、まず lambda-fv という 1 引数の構成子が定義される。その引数は式でなければならず、その上には fv という属性が定義されていなければならない。また、eval という属性が、ベースレベルの実行時にその fv 属性を参照して返すものとして定義される。

そして、後の 4 つの define-attr によって、fv 属性が lambda-con, lambda-var, lambda-app, lambda-abs に対して定義される。これらは参照されるとそれぞれの自由変数の定義にしたがって変数の名前の集合を求めるものである。

5.3.8 衝突解決

4.1.4 節 で述べたように、自由変数を求めるモジュールと Let は衝突を起こす。この衝突は評価規則が足りないことによって起こるものであり、次のような評価規則を補完することによって解決できる。

```
(define-attr lambda-let (fv)
  (lambda (node)
    (lambda () $node:expanded:fv)))
```

この定義は lambda-let に fv 属性を定義するものであり、その内容は expanded 上の fv 属性をそのまま使うというものである。

5.4 LR 構文解析器

本節では 4.2 節で述べた LR 構文解析器の実装について述べる。

5.4.1 文法記述の文法定義を行なうモジュール

構成子を次のように定義する。

```
(define-tag (gram (start) (ps)))
(define-tag (prods (p) (ps)))
(define-tag (prod0))
(define-tag (prod (n) (ss) (a)))
(define-tag (syms (s) (ss)))
(define-tag (sym0))
(define-tag (nont (label)))
(define-tag (term (label)))
```

これらの定義は 4.2.2 節 で述べた ΔM_{BNF} に対応するものである。

また、branches と branch0 はリスト状の構造を構成するので、便宜のため、任意の長さのリストを生成する関数として branch-list などを定義する。

```
(define (branch-list . args)
  (if (null? args)
    (branch0)
    (branches (car args) (apply branch-list (cdr args)))))
```

これらは、list 関数とよく似た関数であり、リストと同様な構造を作るが、cons の代わりに branches という 2 引数の構成子を使い、終端の空リストの代わりに branch0 という引数のない構成子を使うものである。

5.4.2 item の集合を求めるモジュール

item の集合を求める規則を次のようにして定義する。

```
(define-attr gram (items)
  (lambda (node)
    (lambda () $node:ps:items)))

(define-attr prods (items)
  (lambda (node)
    (lambda () (set-union $node:p:items $node:ps:items))))

(define-attr prod0 (items)
  (lambda (node)
    (lambda () (set))))

(define-attr prod (items)
  (lambda (node)
    (lambda () $node:ss:items)))
```

```
(define-attr prod (ss lhs)
  (lambda (node)
    (lambda () $node:n)))

(define-attr prod (ss left)
  (lambda (node)
    (lambda () '())))

(define-attr syms (ss lhs)
  (lambda (node)
    (lambda () $node:lhs)))

(define-attr syms (ss left)
  (lambda (node)
    (lambda () (append $node:left (list $node:s:label)))))

(define-attr syms (right)
  (lambda (node)
    (lambda () (append (list $node:s:label) $node:ss:right))))

(define-attr syms (items)
  (lambda (node)
    (lambda ()
      (set-add
        $node:ss:items
        (item $node:lhs $node:left $node:right)))))

(define-attr sym0 (right)
  (lambda (node)
    (lambda () '())))

(define-attr sym0 (items)
  (lambda (node)
    (lambda () (set (item $node:lhs $node:left '())))))
```

これらの定義は 4.2.3 節 で述べた ΔM_{Item} に対応するものである。

ここで、item を実際に生成しているのは `syms` および `sym0` の `items` 属性であり、他の評価規則はそれらを生成するための情報を生成し、また、生成した item をまとめて一つの集合とするためのものである。item 生成には item の左辺 (`lhs`) および右辺の . の左右 (`left`, `right`) の情報が必要であるが、item は抽象構文木上の . に対応する節[3]で生成するため、. よりも `lhs` および `left` は継承属性によって求め、. よりも `right` は合成属性によって求めている。

[3]正確には . は節と節の間にあると考えられるが、ここでは . の右の節が対応すると考える。つまり、. が右端にある場合には対応する節は `sym0` である。

5.4.3 LR オートマトンを求めるモジュール

LR オートマトンは次のようにして定義される。ここで、make-automaton は開始記号と item の集合から LR オートマトンを実際に生成する関数[4]であり、ここでは生成した LR オートマトンを節として表現しなおしている。

```
(define-tag (state (bs)))
(define-tag (branches (b) (bs)))
(define-tag (branch0))
(define-tag (shift (s) (st)))
(define-tag (reduce (a)))
(define-tag (accept))

(define-attr gram (automaton)
  (lambda (node)
    (lambda ()
      (let* ((ret (make-automaton $node:start $node:items))
             (items-state-alist (car ret))
             (transitions (cadr ret))
             (reduce-list (caddr ret))
             (start-state (cadddr ret))
             (end-item (car (cddddr ret)))
             (size (length transitions))
             (state-vec (make-vector size))
             (states (lambda (i) (vector-ref state-vec i))))
        (do ((i 0 (+ i 1)))
            ((= i size))
          (vector-set! state-vec i
            (state-lazy
              (apply
                branch-list
                (append
                  (map
                    (lambda (trans)
                      (shift
                        (car trans)
                        (vector-ref state-vec (cdr trans))))
                    (cdr (assoc i transitions)))
                  (map
                    (lambda (item)
                      (if (equal? item end-item)
                        (accept)
                        (reduce item)))
                    (cdr (assoc i reduce-list)))
                  )
                ))))
```

[4]詳細は付録 A.5 を参照。

```
        (states start-state)))))
```

これらの定義は 4.2.4 節 で述べた ΔM_{At} に対応するものである。

オートマトンは一般にサイクルを含み得るグラフ構造であり、木構造にはならない可能性がある。したがって抽象構文「木」として直接表現することはできない。ここではこの問題を抽象構文「グラフ」として表現することによって解決している。つまり、オートマトンを直接表現するのである[5]。また、このグラフ構造を実現するために、遅延評価によって状態 (に対応する節) を生成している。このように生成されたオートマトンが非終端属性の automaton に接ぎ木される。

5.4.4 衝突を解決するモジュール

衝突解決は次のようにして定義される。

```
(define-attr shift (applicable)
  (lambda (node)
    (lambda ()
      (lambda (w)
        (lambda (c)
          (eq? (caar w) $node:s:label))))))

(define-attr reduce (applicable)
  (lambda (node)
    (lambda ()
      (lambda (w)
        (lambda (c)
          (set-has? $node:lookaheads (map car (headk w k))))))))
```

これらの定義は 4.2.5 節 で述べた ΔM_{Res}, $\Delta M_{\mathrm{LRkRed}}$ に対応するものであり、shift および reduce の applicable 属性を定義している。このモジュールは個々の選択肢が適用可能かどうかを判断するものであり、複数の選択肢が適用可能である場合にどの選択肢を適用するかを判断するものではない。その判断は構文解析を行なうモジュールに依存する。5.4.5 節で述べる構文解析では選択肢は branches と branch0 によるリスト構造によって一列に並んでおり、最初の選択肢から順に判断が行なわれ、最初に適用可能と判断された選択肢が選択される。

[5] 単純な関数型言語の拡張の Let の例でも、DAG を使って構造を表現している。

5.4.5 構文解析を行なうモジュール

構文解析は次のようにして定義される。

```
(define-attr gram (parse)
  (lambda (node)
    (lambda ()
      (lambda (w)
        (($node:st:parse (append w 'end)) '()))))))

(define-attr state (parse)
  (lambda (node)
    (lambda ()
      (lambda (w)
        ($node:bs:parse w)))))

(define-attr branch0 (parse)
  (lambda (node)
    (lambda ()
      (lambda (w)
        (error "parse error" w)))))

(define-attr branches (parse)
  (lambda (node)
    (lambda ()
      (lambda (w)
        (lambda (c)
          (if (($node:b:applicable w) c)
            (($node:b:parse w) c)
            (($node:bs:parse w) c)))))))

(define-attr shift (parse)
  (lambda (node)
    (lambda ()
      (lambda (w)
        (lambda (c)
          (($node:s:parse
            (cdr w))
              (cons (cons $node:s:parse $node:t:value) c)))))))

(define-attr reduce (parse)
  (lambda (node)
    (lambda ()
      (lambda (w)
        (lambda (c)
          (let ((c (list-tail c (length $node:p:rhs))))
            (((caar c) (cons $node:p:lhs w)) (cdr c)))))))))

(define-attr prod (lhs)
```

```
  (lambda (node)
    (lambda ()
      $node:n:label)))

(define-attr prod (rhs)
  (lambda (node)
    (lambda ()
      $node:ss:right)))
```

これらの定義は 4.2.6 節 で述べた $\Delta M_{\mathrm{Parse}}$ に対応するものである。

構文解析は LR オートマトンを動作させることによって行なう。LR オートマトンは実行時に (有限の) 状態とスタックを保持して入力のトークン列を読み、状態とスタックを変化させていくことによって動作する。この定義では、状態の変化はどの節の parse 属性が実行されているかということによって表現する。つまり、LR オートマトンは抽象構文グラフとして表現されており、各状態に対応する節が存在する。したがって、各状態に対応する parse 属性を作ることができ、この属性の値の関数が末尾呼び出しによって次の状態の parse 属性を呼び出すことによって状態遷移を表現している。スタックおよび入力はそれらの関数の引数として保持され、入力は Shift 時に消費され、スタックは状態遷移時に更新される。

第 6 章

結論

本研究ではメタレベルを属性文法的に記述されたメタレベルモジュールの組み合わせによって構成するという構成法を提案し、特に複数の拡張の衝突について考察した。本章ではメタレベルをモジュール化する他の手法との比較について考察し、まとめを行なう。

6.1 OOP によるメタレベルとの比較

本研究のメタレベルの構成法はメタレベルをメタレベルモジュールに分割/合成して記述するものである。この記述法は、OOP によってメタレベルを記述し、継承によってメタレベル記述を合成することに似ている。ここで、特に mix-in のように複数のクラスを合成する場合に似ているが、単一継承の場合でもサブクラスにはスーパークラスのメソッド等が合成されている。

本研究のメタレベルと OOP によるメタレベルには表 6.1 のような対応があるが、次の点が異なる。

表 6.1: OOP によるメタレベルとの対応

本研究	OOP
メタレベルモジュール	クラス
評価規則	メソッド

- メソッドは何回でも呼び出すことが可能で副作用を持てるが、評価規則はそれが定義する属性ごとに一回しか呼び出せず、属性の値を決定する以外のことができない。

  この性質は自由度を下げるものであるが、そのかわりに評価規則間の関係が明瞭になっている。また、この性質は衝突を確実に検出するために必要な性質である。

- 同じ属性に対する評価規則を持つメタレベルモジュールを合成した場合にはその属性は使用不能になるが、同じ名前のメソッドをもつクラスを合成した場合にはどちらかが優先される[1]。また、優先されたメソッドは優先されなかったメソッドを利用することができる。なお、この属性が使用不能になるという性質は多相性を禁止するものではなく、異なる構文に対するものであれば同じ名前の属性に異なる評価規則があってもよい。これは OOP では複数のクラスが同名のメソッドを独立に (実装を継承せずに) 実装している場合に対応する。

  この性質も自由度を下げるものであるが、上記の属性を複数回参照した場合の性質と同様に衝突を確実に検出できるという性質を実現するために必要な性質である。

- OOP では情報隠蔽が重要視されるが、本研究のメタレベルでは情報隠蔽の方法は提供していない。

  このことは明らかな欠点ではあるが、属性名に名前空間を導入して情報隠蔽を行なうことは容易である。ただし、OOP ではクラス間の関係 (スーパークラス/サブクラスなど) が存在してその関係を利用して隠蔽の度合を変えられるのに対し、メタレベルモジュール間に明確な関係が定義されていないため、public[2] と private[3] 以外の隠蔽方法[4] を自然に定義することは困難である。

[1] スーパークラスとサブクラスに同じ名前のメソッドがあればサブクラスのものが優先される。また、多重継承で、複数のスーパークラスに同じ名前のメソッドがあれば、クラス間に優先順位を与えることにより優先順位が決定される。

[2] どのモジュールからでも自由に使用できるもの。

[3] その属性を定義したモジュールないだけでしか使えないもの。

[4] サブクラスからのみ使えるもの (protected) など。

- クラスにはパラメタをつけられる場合があるが、本研究ではメタレベルモジュールに対するパラメタは提供していない。パラメタは再利用性を向上させるために有用であるが、評価規則の同一性を自明に定義できなくするため、単純に導入することはできない。これは今後の課題である。

6.2 モジュール化された属性文法

[14] では属性文法をモジュール化する方針を次の 3 つに分類している。

Nonterminal = Module 非終端記号毎にモジュールにまとめる。

Attribute = Module 属性毎にモジュールにまとめる。

Semantic aspect = Module 意味的なまとまりでモジュールにまとめる。

これらはモジュールをどのような単位でまとめるかということを示しており、本研究のメタレベルモジュールは意味的なまとまりでモジュール化するものに対応する。

メタレベルにおける拡張可能性を考えた場合、非終端記号や属性毎にモジュールを作ることは適切ではない。これは新しい非終端記号を導入する場合もあれば、新しい属性を導入する場合もあり、どちらかの見方のみでモジュール化することはできないからである。たとえば、4.1.3 節で述べた自由変数を求める拡張では既存の非終端記号 (E) に新しい属性 (fv) を導入しているため、非終端記号毎のモジュール化では既存のモジュールの修正なしには拡張できない。また、既存の属性 (eval) に Fv に対する新しい定義を追加しなければならないので、属性毎のモジュール化でも既存のモジュールの修正なしには拡張できない。

6.3 拡張可能なプリプロセッサ

プリプロセッサはプログラムを処理するプログラムである。ここで、(変換する対象の) プログラムの意味を解釈して変換すると考えれば、拡張可能なプリプロセッサはメタレベルアーキテクチャとしてみなすことができる。ここで、拡張可能な

Java プリプロセッサに EPP[7] がある。EPP ではさまざまな言語機構を plug-in として実装し、EPP 本体を拡張することができ、衝突を検出するフレームワークを持つ。

EPP はプリプロセッサであるから、その処理はソースコードからソースコードへの変換として定義される。その変換は複数のパスからなり、plug-in それらのパスに変換処理を追加することや、パス自体を追加することができる。ここで、plug-in が新しい構文を追加した場合には、その構文は変換によって消去される。EPP における衝突を検出するフレームワークはこのような変換による構文の種類の変化に注目して衝突を考察している。つまり、複数の変換によって変換された結果が正しく (拡張されていない) Java のコードになることを確認するのである。このような変化の系列を考えなければならないのはプリプロセッサでは変換の順序が重要な意味を持つためである。それに対して、本研究の拡張ではメタレベルモジュールの合成において交換律と結合律が成り立つため、拡張の順序に影響されない。

6.4 まとめ

本研究で提案した構成法は次のような利点を持つ。

- メタレベルアーキテクチャが適用可能な応用のほとんどに適用できる。
- メタレベルはモジュール化され、自由に分割、再利用できる。
- メタレベルモジュールの衝突を確実に検出できる。

独立に開発された複数の拡張を組み合わせて使用した場合、個々の拡張自体は正しく動いたとしても、拡張が衝突して適切な動作をしないことがある。この問題を避けるためには個々の拡張の影響範囲を明確に定義し、衝突を確実に検出できなければならない。しかし、影響範囲を定義するためには、メタレベルに構造を定義し、個々の部分が影響範囲に含まれるかどうかを決定できなければならない。そして、すべてのメタレベルに共通する構造というものは存在しないため、メタレベル一般に適用できる構造を定義することは困難であり、これまでに設計されたメタレベルはそのメタレベルが利用されるアプリケーションに依存して構造を設計されていた。それに対し、本研究ではベースレベルの文法構造をメタレベル

の構造として採用し、その構造上にメタレベルをモジュール化して定義することを提案する。この構造はほとんどのメタレベルアーキテクチャで成り立つものであるため、メタレベルアーキテクチャが適用可能な応用のほとんどに適用できる。また、この構造はベースレベルプログラムというユーザの目に見える形で示されるため、把握しやすいという利点もある。

メタレベルは構文木内の各節の属性を評価することによって、ベースレベルの解釈を行なう。そして、メタレベルのモジュールは構文規則と評価規則の集合として表現され、メタレベル全体はそれらの集合の単純な和集合となる。つまり、構文規則や評価規則は最終的なメタレベルに含まれるのならば、どのモジュールで定義されていても同じ意味を持つ。このため、メタレベルの分割の仕方には制限がなく、拡張を行なう際に既存のモジュールを変更しなくても新しいモジュールを定義することによって同じ効果が得られる。この性質によって既存のモジュールを利用する他のモジュールに対する互換性を保ちつつ自由に拡張することができる。

属性の値は評価規則によって決まるが、単一の属性に対して複数の評価規則が存在した場合、もしくは評価規則がひとつも存在しなかった場合には属性の値を一意に評価することができない。これらの場合はメタレベルで衝突が起きている。提案した構成法ではモジュール間 (評価規則間) の情報の受渡しには属性のみが使用されるため、このような衝突が起きず、かつその属性の値が意図通りのものであれば衝突は起きていないことが保証される。これは属性の値を一意に評価することができない場合は計算を続行できないためエラーとなるため、機械的な検出が可能なためである。また、属性の値が意図通りのものであることを保証するには、その属性の値を意図通りに定義するモジュールを使用すれば良い。ここで、あるモジュール a の開発時に他のモジュール b が定義する属性に依存していたとすれば、b が定義する属性は意図通りのものであるといえる。したがって、あるメタレベルに a が使われる時には、b (および開発時に b が依存していたモジュール) も使えば、意図通りの値が求まることを保証できる。ここで、開発時には使われていなかったモジュールが使われて問題の属性の計算過程に干渉する場合、その属性の評価規則が複数存在することになり衝突が検出される。つまり、モジュールを追加する限りでは衝突は確実に検出可能であり、計算が正常に終了した場合

には衝突は起きなかったことが保証される。

本論文ではこれらのことを次の 2 つの例によって述べた。

- 簡単な関数型言語

  この例では λ 計算を元にした簡単な関数型言語とそれに対するいくつかの拡張を述べている。この言語の基本的なメタレベルは λ 計算の 3 つの構文に対応するモジュールからできており、それぞれの拡張モジュールもその拡張を起動するための構文を持っている。しかし、ある拡張 (自由変数を求めるもの) はその構文上の属性に対する評価規則だけでなく、他の構文上の属性に対するものも持っており、メタレベルのモジュールが自由に規則を追加できることを示している。また、複数の拡張によって衝突が起こることも述べられており、新しいモジュールを導入して衝突を解決することにも触れている。さらに、拡張が既存の定義を再利用して定義できることも述べている。

- LR 構文解析器

  この例は提案した構成法が狭義の言語処理系以外にも適用できることを示している。また、ソースコード (BNF によるアクションつきの文法記述) とは直接対応しない構造を扱うために、構文木を属性として生成することについて述べている。

また、今後の課題としては、さらに多くの拡張の衝突について検討し、また言語処理系とはさらに離れた例への適用を予定している。たとえば、(オブジェクト指向言語での) クラス階層や、XML[18] など、拡張可能な木構造はさまざまなものがあり、それらを対象にするメタレベルも考えられる。

謝辞

本研究を行なうに当たり, 終始御指導を賜わった渡部卓雄助教授に深謝致します.

最後に, 本論文をまとめるに当たって御協力いただいた言語設計学講座の諸兄に厚く御礼申し上げます.

参考文献

[1] P. Cointe. Reflective languages and metalevel architectures. *ACM Computing Surveys*, Vol. 28, No. 4es, pp. 151–151, December 1996.

[2] Gianpaolo Cugola, Carlo Ghezzi, and Mattia Monga. Language support for evolvable software: An initial assessment of aspect-oriented programming. In *Proceedings of International Workshop on Principles of Software Evolution*, pp. 98–102, July 1999.

[3] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, Reading, MA, USA, 1995.

[4] Harald Ganzinger. Transforming denotational semantics into practical attribute grammars. In *Proceedings of Semantics-directed compiler generation*, LNCS 94, pp. 1–69, January 1980. Lecture Notes in Computer Science 94.

[5] Adele Goldberg and David Robson. *Smalltalk–80: The Language and its Implementation*. Addison-Wesley, Reading, MA, USA, 1983.

[6] Dick Grune and Ceriel J. H. Jacobs. *PARSING TECHNIQUES A Practical Guide*. Ellis-Horwood, 1990.

[7] 一杉裕志. 拡張可能 java プリプロセッサ epp の型チェック機構フレームワーク. In *Systems for Programming and Applications '99*, March 1999.

[8] Stephen H. Kaisler. *INTERLISP: the language and its usage*. Wiley-Interscience, New York, NY, USA, 1986.

[9] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, Vol. 33, No. 9, pp. 26–76, September 1998. With H. Abelson, N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, and M. Wand.

[10] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[11] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect–oriented programming. In *Proceedings ECOOP '97*, LNCS 1241, pp. 220–242. Springer Verlag, June 1997.

[12] Cristina Lopes and Gregor Kiczales. Recent developments in AspectJ™. In *Proceedings of the Aspect Oriented Programming Workshop at ECOOP '98*, LNCS 1543, pp. 398–401. Springer Verlag, October 1998.

[13] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. Metalevel decomposition in AL-1/D. *Lecture Notes in Computer Science*, Vol. 742, pp. 110–127, 1993.

[14] Jukka Paakki. Attribute grammar paradigms — A high-level methodology in language implementation. *ACM Computing Surveys*, Vol. 27, No. 2, pp. 196–255, June 1995.

[15] Brian Cantwell Smith. Reflection and semantics in Lisp. In *ACM POPL*, pp. 23–35, January 1984.

[16] 田中哲, 渡部卓雄. 制御の流れを明示しない拡張可能な言語処理系記述. 日本ソフトウェア科学会第 16 回全国大会論文集, pp. 261–264. 日本ソフトウェア科学会, September 1999.

[17] The AspectJ team. AspectJ™Web Site. http://www.aspectj.org/.

[18] The World Wide Web Consortium. Extensible markup language (xml). http://www.w3.org/XML/.

[19] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. *ACM SIGPLAN Notices*, Vol. 24, No. 7, pp. 131–145, July 1989.

[20] 渡部卓雄. リフレクション. コンピュータソフトウェア, Vol. 11, No. 3, pp. 5–14, May 1994.

[21] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), San Diego CA.*, pp. 306–315, September 1988. (Revised version in Chapter 3 of [23].).

[22] Takuo Watanabe and Akinori Yonezawa. An actor-based metalevel architecture for group-wide reflection. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Proceedings of REX School/Workshop on Foundations of Object-Oriented Languages (REX/FOOL), Noordwijkerhout, the Netherlands, May, 1990*, No. 489 in Lecture Notes in Computer Science, pp. 405–425. Springer Verlag, 1991.

[23] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System.* The MIT Press, 1990.

本研究に関する発表論文

[1] 田中 哲, 渡部 卓雄: “拡張可能な言語システムのためのフレームワーク”, ソフトウェア工学の基礎 III 日本ソフトウェア科学会 FOSE'96, pp.154–161(1996 年 12 月).

[2] 田中 哲, 渡部 卓雄: “拡張可能な言語システムの効率的な実装への試み”, WOOC '97 オンライン論文集 10 頁 (1997 年 3 月).

[3] Akira Tanaka, Takuo Watanabe: “An Extensible LR Parser Generator — A Case Study of Composable Metalevel Extensions —”, Proceedings of International Workshop on Principles of Software Evolution, pp.84–88(1999 年 7 月).

[4] 田中 哲, 渡部 卓雄: “メタレベルの拡張に適したメタレベルの構成法”, 情報処理学会論文誌 (投稿中) (2000 年 3 月)

[5] 田中 哲, 渡部 卓雄: “メタレベル記述の再利用を考慮した自己反映的プログラミング言語”, 情報処理学会第 51 回全国大会論文集 (5), pp.73–74(1995 年 9 月).

[6] 田中 哲, 渡部 卓雄: “再利用可能な部品から構成された言語処理系”, 電子情報通信学会技術研究報告 Vol.96 No.81 [ソフトウェアサイエンス], pp.25–32(1996 年 6 月).

[7] 田中 哲, 渡部 卓雄: “言語のメタレベルアーキテクチャにおけるモジュール化手法”, 日本ソフトウェア科学会第 13 回全国大会論文集, pp.229–232(1996 年 9 月).

[8] 田中 哲, 渡部 卓雄: “一般的な拡張機構を持つ言語とその実現法”, 日本ソフトウェア科学会第 14 回全国大会論文集, pp.213–216(1997 年 9 月).

[9] 田中 哲, 渡部 卓雄: “制御の流れを明示しない拡張可能な言語処理系記述”, 日本ソフトウェア科学会第 16 回全国大会論文集, pp.261–264(1999 年 9 月).

Appendix A

Scheme による実装

A.1 セルを扱うライブラリ

このライブラリは単一代入な変数を実現するものであり、メタレベルを扱うライブラリから利用される。

```
;;; cell operations:
;
; (cell-alloc): cell
; (cell-alloc-evaluated): cell
; (cell-set-thunk! cell thunk): unspecified
; (cell-ref cell): value
; (cell-set-value! cell): unspecified

(define cell-state-uninitialized 'uninitialized)
(define cell-state-unevaluated 'unevaluated)
(define cell-state-evaluating 'evaluating)
(define cell-state-evaluated '())

(define (cell-alloc)
  (cons #f cell-state-uninitialized))

(define (cell-alloc-unevaluated thunk)
  (cons thunk cell-state-unevaluated))

(define (cell-alloc-evaluated val)
  (cons val cell-state-evaluated))

(define (cell-uninitialized? cell)
  (eq? (cdr cell) cell-state-uninitialized))

(define (cell-unevaluated? cell)
  (eq? (cdr cell) cell-state-unevaluated))

(define (cell-evaluating? cell)
  (eq? (cdr cell) cell-state-evaluating))

(define (cell-evaluated? cell)
  (eq? (cdr cell) cell-state-evaluated))

(define (cell-set-thunk! cell thunk)
  (if (cell-uninitialized? cell)
    (begin
      (set-car! cell thunk)
      (set-cdr! cell cell-state-unevaluated))
    (error "cell is not uninitialized" cell)))

(define (cell-set-value! cell val)
  (set-car! cell val)
  (set-cdr! cell cell-state-evaluated))

(define (cell-ref cell)
  (cond
    ((cell-uninitialized? cell)
      (error "uninitialized cell referenced" cell))
    ((cell-unevaluated? cell)
      (let ((thunk (car cell)))
        (set-cdr! cell cell-state-evaluating)
        (set-car! cell (thunk))
        (set-cdr! cell cell-state-evaluated)
        (car cell)))
    ((cell-evaluating? cell)
      (error "evaluating cell referenced" cell))
    ((cell-evaluated? cell)
      (car cell))
    (else (error "unknown cell status" cell))))
```

A.2 メタレベルを扱うライブラリ

このライブラリは本論文で述べたメタレベルの構成を支援するライブラリである。

```
;;; attribute graph

(define proxy-list (vector 1))

(define (proxy-alloc! val)
  (if (= (vector-length proxy-list) (vector-ref proxy-list 0))
    (set! proxy-list
      (apply vector
        (append
          (vector->list proxy-list)
          (vector->list
            (make-vector (vector-length proxy-list)))))))
  (let ((id (vector-ref proxy-list 0)))
    (vector-set! proxy-list id val)
    (vector-set! proxy-list 0 (+ id 1))
    id))

(define (val->proxy val)
  (let loop ((i (- (vector-ref proxy-list 0) 1)))
    (if (zero? i)
      (proxy-alloc! val)
      (if (eq? val (vector-ref proxy-list i))
        i
        (loop (- i 1))))))

(define (proxy->val id)
  (vector-ref proxy-list id))

;;; node manipulation (tag independent)

(define (node-alloc tag)
  (list tag #f))

(define node->tag car)
(define node->grafted cadr)
(define node->attrs cddr)

(define (node-grafted! node)
  (set-car! (cdr node) #t))

(define (node? node) ; xxx
  (and
    (pair? node)
    (assq (car node) taginfo)))

(define (node-attr-cell node attr)
  (let ((p (assq attr (node->attrs node))))
    (if p
      (cdr p)
      (let ((p (cons attr (cell-alloc))))
        (append! node (list p))
        (cdr p)))))

(define (node-proxy-alloc n) (cons 'proxy (val->proxy n)))
(define (node-proxy-ref p) (proxy->val (cdr p)))
(define (node-proxy? o) (and (pair? o) (eq? 'proxy (car o))))
```

```
(define (node-cell-ref cell)
  (if (cell-unevaluated? cell)
    (let ((v (cell-ref cell)))
      (cond
        ((node-proxy? v) (node-proxy-ref v))
        ((node? v)
          (if (node->grafted v)
            (cell-set-value! cell (node-proxy-alloc v))
            (node-grafted! v))
          v)
        (else
          v)))
    (let ((v (cell-ref cell)))
      (if (node-proxy? v)
        (node-proxy-ref v)
        v))))

(define (node-attr+-cell node path)
  (let loop ((node node) (attr (car path)) (path (cdr path)))
    (let ((cell (node-attr-cell node attr)))
      (if (null? path)
        cell
        (loop (node-cell-ref cell) (car path) (cdr path))))))

(define (node-evaluate node path)
  (if (null? path)
    node
    (node-cell-ref (node-attr+-cell node path))))

;;; node manipulation (tag dependent)

(define (make-node-internal tag-attr-defs tag . args)
  (let ((node (node-alloc tag)))
    (let loop ((attr-defs (apply tag-attr-defs tag args)))
      (if (null? attr-defs)
        node
        (let ((attr-def (car attr-defs)))
          (cell-set-thunk!
            (node-attr+-cell node (car attr-def))
            ((cdr attr-def) node))
          (loop (cdr attr-defs)))))))

(define (make-node tag . args)
  (apply make-node-internal tag-attr-defs tag args))

(define (make-thunk form) `(lambda () ,form))

; (make-node-lazy tagname args)
(define (make-node-lazy-func form env)
  (let ((tagname-exp (cadr form))
        (arg-exps (cddr form)))
    `(make-node-lazy-internal
      ,tagname-exp
      ,@(map make-thunk arg-exps))))
(define make-node-lazy
  (procedure->memoizing-macro make-node-lazy-func))

(define (make-node-lazy-internal tag . args)
  (apply make-node-internal tag-attr-defs-lazy tag args))

;;; tag manipulation

(define taginfo '())

; (define-tag (tagname attrpath...) attrdef...)
(define (define-tag-func form env)
  (let* ((tagname (caadr form))
         (tagname-lazy-func (sym-append tagname '-lazy-func))
         (tagname-lazy (sym-append tagname '-lazy))
         (attrpaths (cdadr form))
         (attrdefs (cddr form))
         (attrpathsyms
           (map (lambda (syms) (sym-join syms #\:)) attrpaths)))
    `(begin
      (define (,tagname ,@attrpathsyms)
        (make-node ',tagname ,@attrpathsyms))
      (define (,tagname-lazy-func form env)
        (cons 'make-node-lazy (cons '',tagname (cdr form))))
      (define ,tagname-lazy
        (procedure->memoizing-macro ,tagname-lazy-func))
      (define taginfo
        (cons
          (list ',tagname (lambda vals (map cons ',attrpaths vals)))
          taginfo))
      ,@(map
        (lambda (attrdef) `(define-attr ,tagname . ,attrdef))
        attrdefs)
      )))
(define define-tag (procedure->memoizing-macro define-tag-func))

; (define-attr tagname attrpath proc)
(define (define-attr-func form env)
  (let ((tagname (cadr form))
        (attrpath (caddr form))
        (proc (cadddr form)))
    `(let ((p (assq ',tagname taginfo)))
      (append! p (list (cons ',attrpath ,(unsugar proc)))))))
(define define-attr (procedure->memoizing-macro define-attr-func))

(define (tag-attr-defs tagname . args)
  (let ((p (assq tagname taginfo)))
    (append
      (apply
        (cadr p)
        (map (lambda (val) (lambda (node) (lambda () val))) args))
      (cddr p))))

(define (tag-attr-defs-lazy tagname . thunks)
  (let ((p (assq tagname taginfo)))
    (append
      (apply
        (cadr p)
        (map (lambda (thunk) (lambda (node) thunk)) thunks))
      (cddr p))))

;;;
(define (unsugar o)
  (cond
    ((symbol? o)
      (let ((s (symbol->string o)))
        (if (char=? (string-ref s 0) #\$)
          (let ((l (str-split (substring s 1 (string-length s))
                              #\:)))
            `(node-evaluate
              ,(string->symbol (car l))
              ',(map string->symbol (cdr l))))
          o)))
    ((pair? o)
      (case (car o)
        ((quote) o)
        (else (map unsugar o))))
    (else o)
  ))
```

A.3 関数型言語のメタレベル

このメタレベルモジュールは 4.1 節で述べた簡単な関数型言語を実現するものである。

```
;;; example (lambda calclus)

;; constant module
(define-tag (lambda-con (val))
  ((eval)
    (lambda (node)
      (lambda ()
        (let ((val $node:val))
          (lambda (env)
            val))))))

;; variable module
(define-tag (lambda-var (lbl))
  ((eval)
    (lambda (node)
      (lambda ()
        (let ((lbl $node:lbl))
          (lambda (env)
            (env-ref env lbl)))))))

;; application module
(define-tag (lambda-app (fun) (arg))
  ((eval)
    (lambda (node)
      (lambda ()
        (let ((fun-eval $node:fun:eval)
              (arg-eval $node:arg:eval))
          (lambda (env)
            ((fun-eval env)
             (arg-eval env))))))))

;; abstraction module
(define-tag (lambda-abs (var) (body))
  ((eval)
    (lambda (node)
      (lambda ()
        (let ((var $node:var)
              (body-eval $node:body:eval))
          (lambda (env)
            (lambda (val)
              (body-eval (env-extend env var val)))))))))

;; dynamic variable module
(define-tag (lambda-anyvar (lbl-exp))
  ((eval)
    (lambda (node)
      (lambda ()
        (let ((lbl-exp-eval $node:lbl-exp:eval))
          (lambda (env)
            (env-ref env (lbl-exp-eval env)))))))

;; let module
(define-tag (lambda-let (var) (exp) (body))
  ((expanded)
    (lambda (node)
      (lambda ()
        (let ((var $node:var)
              (exp $node:exp)
              (body $node:body))
```

```
          (lambda-app (lambda-abs var body) exp)))))
  ((eval)
    (lambda (node)
      (lambda ()
        (let ((expanded-eval $node:expanded:eval))
          expanded-eval)))))

;; free variable module
(define-tag (lambda-fv (exp))
  ((fv)
    (lambda (node)
      (lambda ()
        (set))))
  ((eval)
    (lambda (node)
      (lambda ()
        (let ((fv $node:exp:fv))
          (lambda (env) fv))))))

(define-attr lambda-con (fv)
  (lambda (node)
    (lambda ()
      (set))))

(define-attr lambda-var (fv)
  (lambda (node)
    (lambda ()
      (set $node:lbl))))

(define-attr lambda-app (fv)
  (lambda (node)
    (lambda () (set-union $node:fun:fv $node:arg:fv))))

(define-attr lambda-abs (fv)
  (lambda (node)
    (lambda () (set-del $node:body:fv $node:var))))

(define-attr lambda-let (fv)
  (lambda (node)
    (lambda () $node:expanded:fv)))
```

A.4 LR パーザのメタレベル

このメタレベルモジュールは 4.2 節で述べた LR パーザを実現するものである。

```
(define (define-list-func form env)
  (let ((name (cadr form))
        (pair (caddr form))
        (term (cadddr form)))
    `(define ,name
      (letrec ((f
        (lambda args
          (if (null? args)
              ,term
              (,pair (car args) (apply f (cdr args)))))))
       f))))
(define define-list (procedure->memoizing-macro define-list-func))

;;;

;;; example LR parser generator

(define-tag (gram (start) (ps)))
(define-tag (prods (p) (ps)))
(define-tag (prod0))
(define-tag (prod (n) (ss) (a)))
(define-tag (syms (s) (ss)))
(define-tag (sym0))
(define-tag (nont (label)))
(define-tag (term (label)))

(define-list prod-list prods (prod0))
(define-list sym-list syms (sym0))

(define-tag (state (bs)))
(define-tag (branches (b) (bs)))
(define-tag (branch0))
(define-tag (shift (s) (st)))
(define-tag (reduce (a)))
(define-tag (accept))

(define-list branch-list branches (branch0))

;;;

(define-attr gram (items)
  (lambda (node)
    (lambda () $node:ps:items)))

(define-attr prods (items)
  (lambda (node)
    (lambda () (set-union $node:p:items $node:ps:items))))

(define-attr prod0 (items)
  (lambda (node)
    (lambda () (set))))

(define-attr prod (items)
  (lambda (node)
    (lambda () $node:ss:items)))

(define-attr prod (ss lhs)
  (lambda (node)
    (lambda () $node:n)))

(define-attr prod (ss left)
  (lambda (node)
    (lambda () '())))

(define-attr syms (ss lhs)
  (lambda (node)
    (lambda () $node:lhs)))

(define-attr syms (ss left)
  (lambda (node)
    (lambda () (append $node:left (list $node:s:label)))))

(define-attr syms (right)
  (lambda (node)
    (lambda () (append (list $node:s:label) $node:ss:right))))

(define-attr syms (items)
  (lambda (node)
    (lambda ()
      (set-add $node:ss:items (item $node:lhs $node:left $node:right)))))

(define-attr sym0 (right)
  (lambda (node)
    (lambda () '())))

(define-attr sym0 (items)
  (lambda (node)
    (lambda () (set (item $node:lhs $node:left '())))))

(define-attr gram (automaton)
  (lambda (node)
    (lambda ()
      (let* ((ret (make-automaton $node:start $node:items))
             (items-state-alist (car ret))
             (transitions (cadr ret))
             (reduce-list (caddr ret))
             (start-state (cadddr ret))
             (end-item (car (cddddr ret)))
             (size (length transitions))
             (state-vec (make-vector size))
             (states (lambda (i) (vector-ref state-vec i))))
        (do ((i 0 (+ i 1)))
            ((= i size))
          (vector-set! state-vec i
            (state-lazy
              (apply
                branch-list
                (append
                  (map
                    (lambda (trans)
                      (shift
                        (car trans)
                        (vector-ref state-vec (cdr trans))))
                    (cdr (assoc i transitions)))
                  (map
                    (lambda (item)
                      (if (equal? item end-item)
                        (accept)
                        (reduce item)))
                    (cdr (assoc i reduce-list)))
                  )
                ))))
        (states start-state)))))

\begin{schemecode}
(define-attr shift (applicable)
  (lambda (node)
    (lambda ()
      (lambda (w)
        (lambda (c)
          (eq? (car w) $node:s:label))))))

(define-attr reduce (applicable)
  (lambda (node)
    (lambda ()
      (lambda (w)
        (lambda (c)
          (set-has? $node:lookaheads (headk w 1)))))))

(define-attr gram (parse)
  (lambda (node)
    (lambda ()
      (lambda (w)
        (($node:st:parse (append w 'end)) '())))))

(define-attr state (parse)
  (lambda (node)
    (lambda ()
      (lambda (w)
        ($node:bs:parse w)))))

(define-attr branch0 (parse)
  (lambda (node)
    (lambda ()
      (lambda (w)
        (error "parse error" w)))))

(define-attr branches (parse)
  (lambda (node)
    (lambda ()
```

```
      (lambda (w)
        (lambda (c)
          (if (($node:b:applicable w) c)
              (($node:b:parse w) c)
              (($node:bs:parse w) c)))))))

(define-attr shift (parse)
  (lambda (node)
    (lambda ()
      (lambda (w)
        (lambda (c)
          (($node:s:parse
            (cdr w))
            (cons (cons $node:s:parse $node:t:value) c)))))))

(define-attr reduce (parse)
  (lambda (node)
    (lambda ()
      (lambda (w)
        (lambda (c)
          (let ((c (list-tail c (length $node:p:rhs))))
            (((caar c) (cons $node:p:lhs w)) (cdr c))))))))

(define-attr prod (lhs)
  (lambda (node)
    (lambda ()
      $node:n:label)))

(define-attr prod (rhs)
  (lambda (node)
    (lambda ()
      $node:ss:right)))
```

A.5　パーザの補助関数

このライブラリは subset construction により LR オートマトンを実際に生成するなど、LR パーザのメタレベルモジュールで使われる補助関数を提供する。

```
(define (item lhs left right) (list lhs '= left right))
(define (item->lhs i) (car i))
(define (item->left i) (caddr i))
(define (item->right i) (cadddr i))

(define (first-item? i) (null? (item->left i)))
(define (last-item? i) (null? (item->right i)))
(define (next-symbol i) (car (item->right i)))
(define (next-item i)
  (let ((r (item->right i)))
    (item
      (item->lhs i)
      (append (item->left i) (list (car r)))
      (cdr r))))

;;

(define (make-firstsym-alist prods)
  (let loop ((prods prods) (al '()))
    (if (null? prods)
      (let ((completed (null? al)))
        (let loop ((al al))
          (if completed
            al
            (begin
              (set! completed #t)
              (loop
                (map
                  (lambda (p)
                    (let* ((old-len (set-size (cdr p)))
                           (new-set
                             (apply
                               set-union
                               (cdr p)
                               (map
                                 (lambda (s)
                                   (let ((p (assq s al)))
                                     (if p (cdr p) (set))))
                                 (cdr p))))
                           (new-len (set-size new-set)))
                      (if (not (= old-len new-len))
                        (set! completed #f))
                      (cons
                        (car p)
                        new-set)))
                  al))))))
      (let* ((prod (car prods))
             (lhs (item->lhs prod)))
        (filter-cps
          (lambda (p) (eq? lhs (item->lhs p)))
          prods
          (lambda (prods rest)
            (loop
              rest
              (cons
                (cons
                  lhs
                  (apply
                    set
                    (apply
                      append
                      (map
                        (lambda (p)
                          (if (last-item? p)
                            '()
                            (list (next-symbol p))))
                        prods))))
                al))))))))

(define (make-firstitems-alist item-set)
  (if (set-empty? item-set)
    '()
    (let* ((i (set-first item-set))
           (lhs (item->lhs i)))
      (filter-cps
        (lambda (i) (eq? (item->lhs i) lhs))
        (set-rest item-set)
        (lambda (items rest)
          (cons
            (cons lhs (cons i items))
            (make-firstitems-alist rest)))))))

(define (automaton-state-closure
         firstsym-alist firstitems-alist items)
  (apply
    set-union
    items
    (map
      (lambda (s)
        (let ((is (assq s firstitems-alist)))
          (if is
            (cdr is)
            (set))))
      (apply
        set-union
        (map
          (lambda (i)
            (if (last-item? i)
              (set)
              (let ((ss (assq (next-symbol i) firstsym-alist)))
                (if ss
                  (cdr ss)
                  (set)))))
          items)))))

(define (automaton-start-items
         firstsym-alist firstitems-alist beg-item)
  (automaton-state-closure
    firstsym-alist
    firstitems-alist
    (set beg-item)))

(define (automaton-next-items
         firstsym-alist firstitems-alist items sym)
  (automaton-state-closure
    firstsym-alist
    firstitems-alist
    (map
      next-item
      (filter
        (lambda (i)
          (and
            (not (last-item? i))
            (eq? (next-symbol i) sym)))
        items))))

(define (automaton-next-symbols item-set)
  (let loop ((item-set item-set) (s (set)))
    (if (set-empty? item-set)
      s
      (let ((i (set-first item-set)))
        (loop
          (set-rest item-set)
          (if (last-item? i)
            s
            (set-add s (next-symbol i))))))))

;;

(define (alloc-state-gen)
  (let ((last-state -1))
    (lambda ()
      (set! last-state (+ 1 last-state))
      last-state)))

(define (make-automaton start-symbol item-set)
  (let* ((top-symbol 'top)
         (beg-item (item top-symbol '() (list start-symbol)))
         (end-item (item top-symbol (list start-symbol) '()))
         (item-set (set-add item-set beg-item end-item))
         (firstitem-set (filter first-item? item-set))
         (firstsym-alist (make-firstsym-alist firstitem-set))
         (firstitems-alist (make-firstitems-alist firstitem-set))
         (alloc-state (alloc-state-gen))
         (start-state (alloc-state))
         (start-items (automaton-start-items
                        firstsym-alist firstitems-alist beg-item))
         )
    (let state-loop ((state-alist
                       (list (cons start-items start-state)))
                     (fresh-states (set start-state))
                     (transitions '()))
      (if (set-empty? fresh-states)
        (list
          state-alist
          transitions
```

```
          (map
            (lambda (p) (cons (cdr p) (filter last-item? (car p))))
            state-alist)
          start-state
          end-item)
        (let* ((state (set-first fresh-states))
               (items (car (rassoc state state-alist))))
          (let symbol-loop ((state-alist state-alist)
                            (fresh-states (set-rest fresh-states))
                            (symbols (automaton-next-symbols items))
                            (trans '())
                            )
            (if (set-empty? symbols)
              (state-loop
                state-alist
                fresh-states
                (cons (cons state trans) transitions))
              (let* ((sym (set-first symbols))
                     (next-items
                       (automaton-next-items
                         firstsym-alist
                         firstitems-alist
                         items
                         sym))
                     (p (assoc next-items state-alist))
                     )
                (let ((next-state (if p (cdr p) (alloc-state))))
                  (symbol-loop
                    (if p
                      state-alist
                      (cons
        (cons next-items next-state)
state-alist))
                    (if p
      fresh-states
      (set-add fresh-states next-state))
                    (set-rest symbols)
                    (cons (cons sym next-state) trans)))
                ))))))))
```

A.6 集合を扱うライブラリ

このライブラリは自由変数や item の集合などに使われる集合を扱うものである。

```
;;; set package
;
; (set-empty? SET)
; (set-has? SET ELT)
; (set-contain? SET1 SET2)
; (set-size SET)
;
; (set ELT...)
; (list->set LIST)
; (set-add SET ELT...)
; (set-del SET ELT...)
;
; (set-union SET...)
; (set-subtract SET SET...)
; (set-intersect SET SET...)
;
; (set-first SET)
; (set-rest SET)

;; comparing system

(define (boolean-cmp a b lt eq gt)
  (if a
    (if b (eq) (gt))
    (if b (lt) (eq))))

(define (symbol-cmp a b lt eq gt)
  (if (eq? a b)
    (eq)
    (string-cmp (symbol->string a) (symbol->string b) lt eq gt)))

(define (char-cmp a b lt eq gt)
  (number-cmp (char->integer a) (char->integer b) lt eq gt))

(define (vector-cmp a b lt eq gt)
  (list-cmp (vector->list a) (vector->list b) lt eq gt))

(define (pair-cmp a b lt eq gt)
  (elt-cmp (car a) (car b)
    lt
    (lambda () (elt-cmp (cdr a) (cdr b) lt eq gt))
    gt))

(define (number-cmp a b lt eq gt)
  (if (< a b)
    (lt)
    (if (> a b)
      (gt)
      (eq))))

(define (string-cmp a b lt eq gt)
  (if (string<? a b)
    (lt)
    (if (string>? a b)
      (gt)
      (eq))))

(define (record-cmp a b lt eq gt)
  (string-cmp (record-name a) (record-name b)
    lt
    (lambda () (list-cmp (record-fields-ref a)
                         (record-fields-ref b)
                         lt eq gt))
    gt))

(define (elt-cmp a b lt eq gt)
  (define (type-string obj)
    (cond
      ((null? obj)      0)
      ((boolean? obj)   1)
      ((number? obj)    2)
      ((symbol? obj)    3)
      ((char? obj)      4)
      ((pair? obj)      5)
      ((string? obj)    6)
      ((vector? obj)    7)
      ((record? obj)    8)
      (else (error "elt-cmp:unknown type" obj))))
  (if (eq? a b)
    (eq)
    (number-cmp (type-string a) (type-string b)
      lt
      (lambda ()
        (cond
          ((boolean? a) (boolean-cmp a b lt eq gt))
          ((symbol? a) (symbol-cmp a b lt eq gt))
          ((char? a) (char-cmp a b lt eq gt))
          ((vector? a) (vector-cmp a b lt eq gt))
          ((pair? a) (pair-cmp a b lt eq gt))
          ((number? a) (number-cmp a b lt eq gt))
          ((string? a) (string-cmp a b lt eq gt))
          ((null? a) (eq))
          ((record? a) (record-cmp a b lt eq gt))
          (else (error "elt-cmp:type-string:unknown type" a))))
      gt)))

(define (elt-eq a b)
  (elt-cmp a b (lambda () #f) (lambda () #t) (lambda () #f)))

;; set api

(define (set . elts)
  (list->set elts))

(define (list->set elts)
  (if (null? elts)
    '()
    (let ((med (car elts)))
      (let loop ((smaller '()) (bigger '()) (rest (cdr elts)))
        (if (null? rest)
          (append!
            (list->set smaller)
            (list med)
            (list->set bigger))
          (elt-cmp (car rest) med
            (lambda ()
              (loop (cons (car rest) smaller) bigger (cdr rest)))
            (lambda ()
              (loop smaller bigger (cdr rest)))
            (lambda ()
              (loop smaller (cons (car rest) bigger) (cdr rest)))))))))

(define (set-add s . elts)
  (set-union s (list->set elts)))

(define (set-del s . elts)
  (set-subtract s (list->set elts)))

(define (set-union . sets)
  (define (union s1 s2)
    (cond
      ((null? s1) s2)
      ((null? s2) s1)
      (else
        (elt-cmp (car s1) (car s2)
          (lambda () (cons (car s1) (union (cdr s1) s2)))
          (lambda () (cons (car s1) (union (cdr s1) (cdr s2))))
          (lambda () (cons (car s2) (union s1 (cdr s2))))))))
  (let loop ((s '()) (sets sets))
    (if (null? sets)
      s
      (loop (union s (car sets)) (cdr sets)))))

(define (set-subtract s . sets)
  (define (subtract s1 s2)
    (cond
      ((null? s1) '())
      ((null? s2) s1)
      (else
        (elt-cmp (car s1) (car s2)
          (lambda () (cons (car s1) (subtract (cdr s1) s2)))
          (lambda () (subtract (cdr s1) (cdr s2)))
          (lambda () (subtract s1 (cdr s2)))))))
  (let loop ((s s) (sets sets))
    (if (null? sets)
      s
      (loop (subtract s (car sets)) (cdr sets)))))
```

```
(define (set-intersect s . sets)
  (define (intersect s1 s2)
    (cond
      ((null? s1) s2)
      ((null? s2) s1)
      (else
        (elt-cmp (car s1) (car s2)
          (lambda () (intersect (cdr s1) s2))
          (lambda () (cons (car s1) (intersect (cdr s1) (cdr s2))))
          (lambda () (intersect s1 (cdr s2)))))))
  (let loop ((s s) (sets sets))
    (if (null? sets)
      s
      (loop (intersect s (car sets)) (cdr sets)))))

(define set-empty? null?)

(define (set-has? s e)
  (if (null? s)
    #f
    (elt-cmp (car s) e
      (lambda () (set-has? (cdr s) e))
      (lambda () #t)
      (lambda () #f))))

(define (set-contain? s1 s2)
  (cond
    ((null? s2) #t)
    ((null? s1) #f)
    (else
      (elt-cmp (car s1) (car s2)
        (lambda () (set-contain? (cdr s1) s2))
        (lambda () (set-contain? (cdr s1) (cdr s2)))
        (lambda () #f)))))

(define set-size length)

(define (set-first s) (car s))
(define (set-rest s) (cdr s))
```

A.7　環境を扱うライブラリ

　このライブラリは関数型言語で使われる環境を実現するものである。

```
;;; environment

(define (env-empty)
  (lambda (var) (error "undefined variable." var)))
(define (env-extend env var val)
  (lambda (v) (if (eq? v var) val (env v))))
(define (env-ref env var) (env var))
```

A.8　リストを扱うライブラリ

　このライブラリは R5RS[9] にないいくつかのリストに関する補助的な関数を提供するものである。

```
(define (assoc-cps e l succ fail)
  (let ((p (assoc e l)))
    (if p
      (succ p)
      (fail))))

(define (filter f l)
  (if (null? l)
    '()
    (if (f (car l))
      (cons (car l) (filter f (cdr l)))
      (filter f (cdr l)))))

(define (filter-cps f l k)
  (if (null? l)
    (k '() '())
    (if (f (car l))
      (filter-cps f (cdr l)
        (lambda (tl fl) (k (cons (car l) tl) fl)))
      (filter-cps f (cdr l)
        (lambda (tl fl) (k tl (cons (car l) fl)))))))

(define (last l)
  (if (null? l)
    '()
    (let loop ((l l))
      (if (null? (cdr l))
        l
        (loop (cdr l))))))

(define (rassoc a l)
  (if (null? l)
    #f
    (if (equal? a (cdar l))
      (car l)
      (rassoc a (cdr l)))))

(define (append! . args)
  (if (null? args)
    '()
    (if (null? (cdr args))
      (car args)
      (let ((anchor (list 'dummy)))
        (let loop ((lastcell anchor) (args args))
          (set-cdr! lastcell (car args))
          (if (null? (cdr args))
            (cdr anchor)
            (loop (last lastcell) (cdr args)))))))))
```

A.9　文字列、シンボルを扱うライブラリ

　このライブラリは R5RS[9] にないいくつかの文字列とシンボルに関する補助的な関数を提供するものである。

```
(define (str-split str ch)
  (let end ((i (string-length str)) (l '()))
    (let beg ((j i))
      (cond
        ((= j 0) (cons (substring str j i) l))
        ((char=? (string-ref str (- j 1)) ch)
          (end (- j 1) (cons (substring str j i) l)))
        (else
          (beg (- j 1)))))))

(define (str-join lst ch)
  (define (f ch l)
    (if (null? l) '() (cons ch (cons (car l) (f ch (cdr l))))))
  (apply string-append
    (car lst)
    (f (string ch) (cdr lst))))

(define (sym-split sym ch)
  (map string->symbol (str-split (symbol->string sym) ch)))

(define (sym-join lst ch)
  (string->symbol (str-join (map symbol->string lst) ch)))

(define (sym-append . syms)
  (string->symbol (apply string-append (map symbol->string syms))))
```

索 引